软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3726−3743 [doi: 10.13328/j.cnki.jos.007260] [CSTR: 32375.14.jos.007260]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于有限状态机引导的网络协议模糊测试方法*

袁 斌[1,2,3,4,5,8,9,10], 任家俊[1,2,3,4,5], 陈群锦明[1,2,3,4,5], 张 驰[1,2,3,4,5], 邹德清[1,2,3,4,5,8], 金 海[1,2,6,7]

[1](大数据技术与系统国家地方联合工程研究中心 (华中科技大学), 湖北 武汉 430074)

[2](服务计算技术与系统教育部重点实验室 (华中科技大学), 湖北 武汉 430074)

[3](湖北省大数据安全工程技术研究中心 (华中科技大学), 湖北 武汉 430074)

[4](分布式系统安全湖北省重点实验室 (华中科技大学), 湖北 武汉 430074)

[5](华中科技大学 网络空间安全学院, 湖北 武汉 430074)

[6](集群与网格计算湖北省重点实验室 (华中科技大学), 湖北 武汉 430074)

[7](华中科技大学 计算机科学与技术学院, 湖北 武汉 430074)

[8](金银湖实验室, 湖北 武汉 430040)

[9](嵩山实验室, 河南 郑州 452470)

[10](深圳华中科技大学研究院, 广东 深圳 518057)

通信作者: 邹德清, E-mail: deqingzou@hust.edu.cn

摘　要: 模糊测试技术能够自动化挖掘软件当中的漏洞, 然而目前针对网络协议的模糊测试工具对于协议实现内部状态空间探索有限, 导致覆盖率较低. 有限状态机技术能够对网络协议实现进行全方位建模, 以深入了解网络协议实现的系统行为和内部状态空间. 将有限状态机技术和模糊测试技术相结合, 提出一种基于有限状态机引导的网络协议模糊测试方法. 以广泛使用的 TLS 协议为研究对象, 利用有限状态机学习来对于 TLS 协议实现进行建模, 用来反映协议内部状态空间及其系统行为. 随后, 基于有限状态机对于 TLS 协议模糊测试进行引导, 使模糊测试的深度更深、覆盖代码更广. 为此, 实现一个原型系统 SNETFuzzer, 并且通过一系列对比实验发现 SNETFuzzer 在覆盖率等重要指标中优于已有工作. SNETFuzzer 在实验中成功发现多个漏洞, 其中包含两个新漏洞, 证明了其实用性和有效性.

关键词: 软件测试; 模糊测试; 网络协议; 有限状态机

中图法分类号: TP311

中文引用格式: 袁斌, 任家俊, 陈群锦明, 张驰, 邹德清, 金海. 基于有限状态机引导的网络协议模糊测试方法. 软件学报, 2025, 36(8): 3726–3743. http://www.jos.org.cn/1000-9825/7260.htm

英文引用格式: Yuan B, Ren JJ, Chen QJM, Zhang C, Zou DQ, Jin H. Fuzz Testing Method for Network Protocols Guided by Finite State Machine. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3726–3743 (in Chinese). http://www.jos.org.cn/1000-9825/7260.htm

Fuzz Testing Method for Network Protocols Guided by Finite State Machine

YUAN Bin[1,2,3,4,5,8,9,10], REN Jia-Jun[1,2,3,4,5], CHEN Qun-Jin-Ming[1,2,3,4,5], ZHANG Chi[1,2,3,4,5], ZOU De-Qing[1,2,3,4,5,8], JIN Hai[1,2,6,7]

[1](National Engineering Research Center for Big Data Technology and System, Wuhan 430074, China)

[2](Services Computing Technology and System Lab (Huazhong University of Sicence and Technology), Wuhan 430074, China)

[3](Hubei Engineering Research Center on Big Data Security (Huazhong University of Science and Technology), Wuhan 430074, China)

[4](Hubei Key Laboratory of Distributed System Security (Huazhong University of Science and Technology), Wuhan 430074, China)

[5](School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China)

* 基金项目: 国家自然科学基金面上项目 (62372191); 国家重点研发计划 (2022YFB3103400)
收稿时间: 2024-04-06; 修改时间: 2024-06-11; 采用时间: 2024-07-17; jos 在线出版时间: 2024-12-25
CNKI 网络首发时间: 2024-12-26

[6](Cluster and Grid Computing Lab (Huazhong University of Science and Technology), Wuhan 430074, China)
[7](School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China)
[8](Jinyinhu Laboratory, Wuhan 430040, China)
[9](Songshan Laboratory, Zhengzhou 452470, China)
[10](Research Institute of Huazhong University of Science and Technology in Shenzhen, Shenzhen 518057, China)

Abstract: Fuzz testing automatically uncovers vulnerabilities in software. However, existing fuzz testing tools for network protocols are not able to fully explore their internal state space, resulting in limited coverage. Finite state machines comprehensively model the implementation of network protocols to provide an in-depth understanding of their system behavior and internal state space. This study proposes a fuzz testing method for network protocols based on finite state machines. It focuses on the commonly used TLS protocol, using finite state machine learning to model the implementation of the TLS protocol, reflecting the protocol's internal state space and system behavior. Subsequently, guided by finite state machines, the fuzz testing of the TLS protocol achieves deeper depth and broader code coverage. This study also implements a prototype system, SNETFuzzer, which outperforms existing methods in important metrics such as coverage in a series of comparative experiments. SNETFuzzer successfully discovers multiple vulnerabilities, including two new ones, demonstrating its practicality and effectiveness.

Key words: software testing; fuzz testing; network protocol; finite state machine

互联网的广泛应用对网络空间的安全性提出了极高的需求. 其中, 网络协议是维护网络空间安全的关键基石, 其安全性直接决定着网络空间的整体安全水平. 网络协议当中普遍存在各种漏洞, 而任何一个网络协议漏洞都可能带来严重的损失和危害.

网络协议的漏洞具有广泛性、隐蔽性、可用性、持久性及危害性等特点[1]. 这是由于网络协议本身使用范围广, 使得其漏洞具有广泛性, 并且这些漏洞既可能存在于协议设计当中, 也可能存在于协议具体实现当中, 隐蔽性较强. 其次, 网络协议实现在运行的时候一定会暴露其通信端口, 这使得网络协议漏洞具有高可用性. 协议实现目前缺乏足够的安全审查和测试, 这导致协议漏洞可能存在较长时间, 具有持久性. 而网络协议漏洞可能会导致严重的安全问题和数据泄露, 给个人、组织甚至整个社会都有可能带来严重经济损失, 这使得其具有严重危害性. 因此, 及时发现并修补网络协议漏洞对于网络空间安全来说至关重要.

模糊测试技术是软件测试领域中的代表性技术[2]. 它通过输入大量随机、无效或异常的数据到应用程序中, 以侦测系统潜在的漏洞, 这种方法简单直接、高效有效. 目前, 许多开源软件已经通过这种方法发现了大量未知的新漏洞. 模糊测试工具具有高度自动化、能够模拟程序实际运行情况以及强适应性的特点.

目前已经有一些工作将模糊测试技术应用在网络协议当中. 它们以特定的方式生成用于大量用于测试网络协议实现的数据包输入, 并且通过变异来使数据包输入尽可能覆盖更多的代码. 然而大部分网络协议模糊测试工作当中有一个严重的问题, 即在探索协议实现内部状态空间方面存在不足. 虽然这些工具在生成大量的随机、无效或异常数据方面表现出色, 确实可以检测一部分潜在的安全漏洞, 但它们对于深入理解和模拟协议实现内部状态的能力有限. 这意味着这些工具可能会错过一些仅在特定状态下才会触发的漏洞, 因为它们未能完全模拟协议实现的各种状态变化, 这使得其代码覆盖率有限.

针对以上问题, 本文提出了一种基于有限状态机的网络协议模糊测试方法, 将有限状态机技术与模糊测试技术相结合并应用在网络协议测试当中. 有限状态机是一种抽象的数学模型, 用于描述系统的行为. 有限状态机所描述的系统具有有限个状态, 并且能够在这些状态之间进行有规则的转移. 有限状态机的状态用于表示系统所处的不同运行状态, 而状态之间的转移描述了系统在不同状态之间的过渡规则和条件. 这些特性使得有限状态机非常适合描述网络协议实现的系统行为和内部状态空间. 有限状态机可用于引导网络协议模糊测试, 解决当前网络协议模糊测试工具对协议实现内部状态空间探索不充分的问题, 以扩大模糊测试的代码覆盖率, 从而可能发现新的未知安全漏洞.

本文的主要贡献如下.

(1) 提出了一种基于有限状态机引导的网络协议模糊测试方法. 通过对于网络协议实现进行有限状态机建模来尽可能探索其内部状态机空间, 并以此引导模糊测试的进行, 提高模糊测试的代码覆盖率.

(2) 针对 TLS 协议设计了一套较为全面的有限状态机学习方法. 本文根据 TLS 协议的各个细节设计了一个详尽的有限状态机模型, 该模型能够更好地反映 TLS 协议的系统行为特征.

(3) 实现了原型系统 SNETFuzzer. SNETFuzzer 在 5 个开源 TLS 协议实现中的模糊测试表现均优于 AFLNET, 并且成功发现了两个新漏洞.

本文第 1 节介绍网络协议模糊测试的相关工作. 第 2 节介绍本文相关基础知识. 第 3 节介绍原型系统 SNETFuzzer 的设计及实现方法. 第 4 节介绍用于测试 SNETFuzzer 的实验及分析. 第 5 节总结全文.

1 相关工作

1.1 模糊测试技术

软件测试可以分为两种主要类型: 静态测试和动态测试. 静态测试是在不执行程序的情况下对软件进行检查和分析的方法[3], 主要关注软件的静态属性, 如代码、设计、文档等. 而动态测试[4]则是在执行程序的情况下对软件进行检查和分析, 重点关注软件的动态属性, 如运行结果、记录反馈、行为和性能表现等. 模糊测试技术属于动态测试的范畴, 其在工业界和学术界都已经有许多相关工作. 模糊测试能够模拟真实世界中的攻击场景, 通过不断变异输入数据, 模糊测试可以触发系统中潜在的边界条件错误、内存泄漏、缓冲区溢出等问题, 从而揭示系统中的未知漏洞. 通过及时发现并修复这些漏洞, 可以有效提高软件的安全性和可靠性, 防止黑客利用这些漏洞对系统进行攻击. 模糊测试技术已经被广泛应用在各个细分领域的安全测试当中, 包括但不限于因特网、工业互联网[5]、IoT[6]、车联网[7]、深度学习[8]等领域.

模糊测试可以大致分为 3 类, 白盒模糊测试、黑盒模糊测试和灰盒模糊测试. 白盒模糊测试的优势在于其能够开源精确测试代码路径和逻辑, 但前提是要求测试人员掌握被测系统源代码和相关文档. Godefroid 等人开发的白盒测试工具 SAGE[9]发现了存在于 Windows 应用系统中的 20 多个未知漏洞. SAGE 是首个在 x86 系统上运行的动态符号执行模糊测试软件, 它运用符号执行技术深入理解程序内部结构, 并生成测试用例以发现潜在的漏洞. Stephens 等人提出了 Driller[10], 这是一种混合漏洞挖掘工具, 它结合了白盒模糊测试技术. Driller 采用了模糊测试和选择性混合执行技术, 以避免混合执行的路径爆炸和模糊测试的不充分性问题. 黑盒模糊测试的优势在于其无需对被测系统有深入了解就可以进行测试, 然而正是因为其对于被测系统内部结构一无所知, 导致难以对系统进行深入测试. Snipuzz[11]是一个用于 IoT 设备的黑盒模糊测试工具, 它通过消息段推断来实现自动化测试, 并且成功发现了 5 个零日漏洞.

灰盒模糊测试技术则是融合了黑盒和白盒的特点, 会在对被测程序有限了解的情况下, 生成输入测试数据进行测试. 相对于黑盒测试, 灰盒模糊测试具有明显的优势, 其测试深度可达一定水平, 且无需白盒测试所需的专业知识. AFL 一直以来都是模糊测试领域最著名的工具之一[12], 被认为是“模糊测试事实上的标准”. 用户需提供至少一个初始种子输出给 AFL, 随后 AFL 会运用变异算法对输入进行变异, 生成新的种子. 该工具利用覆盖率来指导生成新的变异输入, 因此需要获得程序的覆盖率. AFL 提出了基于代码覆盖率的模糊测试算法, 这一算法影响了许多其他灰盒模糊测试工具. AFLGo[13]则是基于 AFL 的改进工作, 旨在通过一种基于模拟退火的资源分配规则, 使输入有效地到达给定的目标程序位置. Gan 等人提出了 CollAFL[14], 旨在减少 AFL 的路径冲突并优化性能表现. 与 AFL 相比, CollAFL 的路径冲突率大幅降低, 测试性能显著增强, 并且在代码覆盖率和漏洞发现方面表现优异. 特定的变异策略和变异引导策略也能够显著改善 AFL 的模糊测试效果, 目前也有相关工作在此方面展开[15,16]. Fioraldi 等人总结了现有的最先进的模糊测试研究, 并将其整合成 AFL++[17]. AFL++有望成为当前研究的新基准工具, 帮助研究人员了解自己的工作与最新工作之间的差异. 作者发现, 尽管每种新颖的模糊测试方法都具有其特点, 能够提高在某些目标或环境下的性能, 但也会降低其他目标的性能. 有些工作将其他技术与模糊测试技术结合, 如在静态检测中常用的符号执行技术[18], 也获得了相当不错的成果. 为了更好地综合比较这些模糊测试工具的效果, Metzman 等人提出了开源的模糊测试评估平台 Fuzzbench[19], 该平台集成了多个模糊测试工具的 API, 为模糊测试工具开发者提供多样化的对比平台.

目前已经有一部分工作将灰盒模糊测试技术应用在网络协议模糊测试当中. 为了将 AFL 用于网络协议模糊测试, Pham 等人提出了 AFLNET[20]. 该工具采用突变方式, 并利用状态反馈机制指导模糊测试过程. AFLNET 利用这些服务器返回的响应代码来标记被测网络协议实现的内部状态, 这些状态共同构成 AFLNET 所理解的被测系统状态空间. Song 等人提出的 SPFuzz[21]也是专门针对网络协议的灰盒模糊测试工具. 与 AFLNET 不同, SPFuzz 定义了一种语言来描述协议规范、协议状态转换和依赖关系, 并采用三级变异策略. 它结合随机分配消息和策略权重的方法来推动模糊测试过程, 以覆盖更多的路径. SnapFuzz[22]则在效率上有了较大的提升, 它将原本慢速的异步网络通信转换为快速的同步通信.

1.2 有限状态机技术

有限状态机 (finite state machine, FSM) 是一种抽象化的数学模型, 被用来描述系统的行为. 它在计算机科学、控制科学、电子工程、机械自动化等领域都有广泛应用. 有限状态机所描述的系统具有有限个状态, 并能按照规则在这些状态之间转移. 状态机的状态用于表示系统的不同运行状态, 而状态之间的转移描述了系统在不同状态之间的过渡规则和条件. De Ruiter 等人提出了一种 TLS 协议状态机学习框架[23], 并利用该框架检查 TLS 协议实现中潜藏的逻辑性漏洞. 逻辑性漏洞与内存型漏洞不同, 指的是系统在错误情况下发生了不应该出现的状态转移, 这种转移不符合协议 RFC 文档的规定, 可能会造成严重的安全威胁. 该框架采用一种有限状态机学习算法对 TLS 协议进行状态机建模, 其黑盒自动化的方式颇具前瞻性. De Ruiter 等人对生成的协议状态机进行人工检查, 在 3 个 TLS 实现中发现了新的安全漏洞. 之后, Fiterau-Brostean 等人在此基础上扩展了工作, 增加了对 DTLS 的支持[24], 并发现了多个安全漏洞. 同时他们证明了这种方法的通用性, 最近还将其扩展到 SSH 协议上[25]. Zou 等人基于 TCP 状态模型的前提知识提出了 TCP-Fuzz[26]. 在此之前的 TCP 模糊测试更多地关注覆盖更多的状态, 而忽略了状态转移, 导致程序覆盖范围有限. TCP-Fuzz 专注于覆盖更多的状态转移, 并考虑了数据包和系统调用之间的依赖关系, 在 5 个不同的 TCP 堆栈程序中发现了多个内存错误和语义错误. StateAFL[27]则提供了一种适用于大量流行协议的模糊测试工具, 与 AFLNET 类似, 它利用逐步探索和构建协议有限状态机来帮助模糊测试, 并加入了内存分析判断.

2 基础知识

本节将主要展示与本方法相关的基础知识, 其中包括 TLS 协议、有限状态机学习框架.

2.1 TLS 协议

本文选择 TLS 协议作为主要研究对象, 主要原因在于其在当今互联网通信中的广泛应用. TLS 协议不仅提供了数据加密、身份认证、数据传输等多种功能, 而且其安全性对于保障网络通信的机密性和完整性至关重要.

TLS 协议由两层组成: 记录协议和握手协议. 记录协议规定传输数据的封装格式, 而握手协议通过交换应用数据来传递双方所需的所有信息, 包括身份认证和加密方式等安全属性. 需要注意的是, TLS 并不完全符合 OSI 模型或 TCP/IP 的任何一层. 它运行在可靠的传输协议 (如 TCP) 之上, 理论上应该位于传输层之上, 提供的加密服务通常属于表示层的功能. 然而, 应用程序通常将 TLS 视为传输层协议, 因为使用 TLS 需要发起握手和处理身份验证证书.

如图 1 所示, TLS 握手过程可以分为 4 个主要阶段[28]: (1) 协商阶段. 首先由客户端发送一条 ClientHello 消息, 指定其支持的 TLS 协议版本、随机数、建议的密码套件和数据压缩方法. 之后服务端将用 ServerHello 消息予以相应, 消息中包含指定的 TLS 协议版本、随机数、密码套件和压缩方法. 服务器还会发送证书消息 (Certificate), 并且可能发送 ServerKeyExchange 消息 (根据所选密码套件, 可能会不发送这个消息), 最后发送 ServerHelloDone 消息表明握手协商完成. 客户端在确认服务端传来的上述消息后, 会使用 ClientKeyExchange 消息进行响应. 随后, 服务端和客户端将使用随机数来计算一个主密钥 (master secret), 该连接的所有其他密钥数据都会源于这个主密钥. (2) 客户端发送一条 ChangeCipherSpec 消息. 实际上是告诉服务器, 接下来之后的所有消息将经过身份验证和

加密. 并发送通过身份验证和加密的 Finished 消息, 这条消息还包含了先前握手消息的哈希值. 服务端将解析 Finished 消息的内容并验证哈希值. 如果验证失败则认为握手失败; 反之进入下一阶段. (3) 服务器发送一条 ChangeCipherSpec 消息. 实际上是告诉客户端, 接下来之后的所有消息将经过身份验证和加密. 同样会发送 Finished 消息. (4) 应用程序阶段: 应用程序在这条连接上传输经过 TLS 协议加密过的数据. 本文旨在为 TLS 协议的握手协议设计一个有限状态机模型. 由于握手阶段涉及多个输入输出, 并且 TLS 协议服务端会根据客户端消息进入多个不同状态, 因此适合将其系统行为建模成有限状态机.

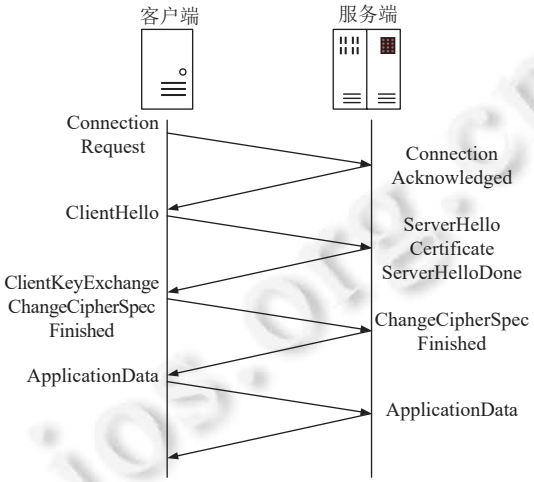

图 1 典型的 TLS 握手过程

2.2 有限状态机学习框架

目前已经有一些开源的有限状态机学习框架. 有限状态机学习框架是用于帮助实现和应用有限状态机学习算法的软件工具. LearnLib[29]是一个用于学习有限状态机的 Java 框架, 它提供了一系列的算法和工具以用于从观测信息序列中学习系统的模型, 常被应用于软件测试、模型检查、自动化验证等各个方面. LearnLib 支持多种算法, 包括 Angluin's L*算法[30](如图 2 所示). Angluin's L*算法是一种经典的用于有限状态机学习的算法, 该算法建立在对一个黑盒被测系统的逐步推测基础上. 在这个算法的构造中, 它假定了一个"老师 (Teacher)"的存在, 认为这位"老师"了解被测系统的一切. 而学习程序则作为"学生 (Learner)". 在 Angluin's L*算法的运行过程中, 有两类主要的问题, 由"学生"向"老师"提出, 分别是成员查询 (membership query) 和等价查询 (equivalence query). 这两类查询对于学习被测系统的有限状态机起着关键作用. 成员查询类似于学生询问老师在给定某个输入时, 被测系统的输出是什么. 通过向老师提出这样的问题, 学生可以逐步收集有关系统行为的信息. 这有助于学生构建对系统状态和转移的假设. 等价查询是学生询问老师目前构造的有限状态机是否与实际被测系统完全吻合. 这种查询的目的在于验证学生对系统行为的假设是否准确, 如果不准确, 则学生需要调整其有限状态机的模型. Angluin's L* 算法的优势在于它的学习过程是通过与系统进行有限次交互来完成的, 且可以保证最终学习到的模型与实际系统行为等价. 然而, 该算法的主要限制在于它对系统的可观察性要求较高, 即需要能够通过输入输出序列来观察系统的行为.

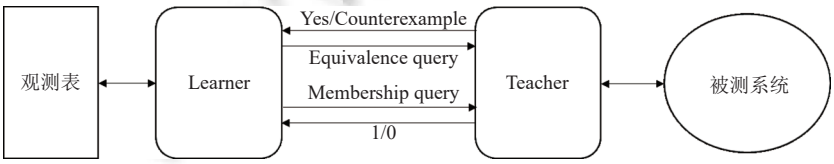

图 2 Angluin's L*算法原理图

3 系统设计

3.1 问题提出

有限状态机在软件行为建模方面具有独特的优势, 尤其是在检查网络协议实现时. 通过对网络协议的有限状态机模型进行检查, 可以发现潜在的逻辑漏洞, 例如状态转换不当或未处理的异常情况. 这样的做法为全面洞察协议行为及其潜在的安全隐患提供了可能. 此外, 有限状态机还能够作为引导模糊测试的有效工具. 在对网络协议实现进行有限状态机学习的过程中, 将其实现视为服务器端, 有限状态机学习程序则扮演客户端角色. 这表明, 通过学习程序构建的状态机模型能够准确映射实际通信中可能出现的各种状态及其转换. 将这些学习到的状态机模型与灰盒模糊测试技术结合起来, 可以更好地模拟实际网络通信场景, 并实现更高效的动态测试. 通过将有限状态机学习和模糊测试相结合, 能够有效地解决传统模糊测试中状态空间探索不充分的问题. 这种组合方法不仅能够提高代码覆盖率, 还能够更快地发现协议实现中的潜在漏洞. 与传统的灰盒模糊测试工具相比, 这种方法还可能发现那些在传统模糊测试中难以察觉的安全漏洞. 具体而言, 在传统的模糊测试过程中, 新生成的测试用例往往仅依赖于上一轮的测试用例, 这样的测试方法难以保证测试用例能够驱使系统到达特定的状态, 并测试该状态下的输入对系统的影响, 因此, 传统模糊测试方法难以测试深层次的具有多级状态转移的漏洞. 而本文提出的基于状态机引导的模糊测试, 可以基于状态机的状态转移信息, 事先输入特定的测试用例, 高效的驱使系统转移到任意状态, 并在该状态下进行测试, 从而有效提升测试的覆盖率.

可将主要研究问题归纳为两个子问题.

(1) 如何生成 TLS 协议的有限状态机并使有限状态机尽可能丰富, 最大可能还原网络协议实现的真实行为. 尽管有限状态机可有效反映软件的行为模式, 但其生成的丰富性与准确度会受到所采用的学习算法、参数配置以及环境设定等因素的影响. 据此, 设计一种能够全面呈现 TLS 协议的行为逻辑的有限状态机构成了本项工作需解决的核心问题. 文章设计并实现一种针对 TLS 协议的有限状态机学习系统, 其中主要采用了 Angluin's L*算法作为核心学习算法. 通过依据 TLS 协议的标准文件 (RFC) 定制字母表, 并对协议的各个握手阶段进行详细的行为分析, 确保了学习结果的精确与完整. 此外, 实现了一个专用的 TLS 协议消息处理与映射程序, 确保学习系统能深入理解待测 TLS 实现的特点与逻辑结构.

(2) 如何利用生成的 TLS 协议实现有限状态机来引导模糊测试的进行. 由于灰盒模糊测试程序本身无法直接访问有限状态机的内部信息, 必须将有限状态机提供的数据转换成模糊测试程序能识别和利用的格式. 执行的策略是从有限状态机中导出程序状态的转移路线, 并依据这些路线信息构建测试案例作为初始种子输入模糊测试. 这一做法使模糊测试在理解了系统状态空间的基础上进行. 例如, 参照后文图 3 所呈现的 OpenSSL 某版本的状态机, 传统灰盒模糊测试工具仅限于对红色路线上的路径进行变异操作. 但这种做法可能遗漏那些在状态机图中以黑色标示、未被测试程序知晓的状态转移. 与之相反, 如果利用 TLS 协议的有限状态机指引模糊测试, 就会像向测试程序透露了所有可能的状态转移路线一样. 这将大大提高程序检测的广度, 因为模糊测试得以更深入地探究系统的状态空间. 并且, 随着初始种子种类的增多, 模糊测试的效能也相应提升, 这促进了更迅速地揭露可能的未知安全漏洞.

针对这两个子问题, 本文提出了一个基于有限状态机引导的网络模糊测试原型系统 SNETFuzzer. 该系统能够学习并建立一个完备的有限状态机模型, 这一模型精确地映射了 TLS 协议的实现行为逻辑. 利用这一有限状态机, SNETFuzzer 引导模糊测试的执行, 旨在扩大代码的检测覆盖面并提升模糊测试的效率. 后文图 4 展示了 SNETFuzzer 的系统架构图.

3.2 TLS 协议有限状态机模型设计

为使 SNETFuzzer 生成的有限状态机尽可能贴近 TLS 协议真实的系统行为, 需要提前设计一个适合描述 TLS 协议的有限状态机模型. 有限状态机模型可以被分为两类: Moore 机和 Mealy 机. Moore 机的状态仅和输出有关, Mealy 机的状态和输出和输入都有关. 然而在网络协议当中, 仅以输出来判定程序进入一个状态是不充分的,

需要再引入输入信息, 由输入和输出共同决定协议进入了一个状态. 这里的输入指的是客户端发送给服务端的数据包, 换言之是有限状态机学习程序发送给被测系统的数据包. 在这种情况下, 使用 Mealy 机更符合要求. 每个有限状态机都可以用一个六元组来表示, 可写为公式 (1):

$$M = (S, s0, \delta, \lambda, X, Y) \tag{1}$$

其中, M 表示有限状态机 (FSM); S 表示所有状态的集合; X 是输入字典; Y 是输出字典; δ 函数是状态转移函数, $sj=\delta(si, x)$ 表示从状态 si 出发, 输入 x 后转移到状态 sj, $x \in X$; λ 函数是输出函数, $y=\lambda(si, x)$ 表示从状态 si 出发, 输入 x 后获得的响应 y, $y \in Y$. 所以, 一次完整的状态转移可以表示为一个三元组: $(si, sj, x/y)$. 在确定了使用的有限状态机类型之后, 需要确认有限状态机的字母表. 字母表指的是在有限状态机中会出现的输入信息和输出信息, 有限状态机学习程序需要通过字母表才能了解它能够发送和可能接收到的信息.

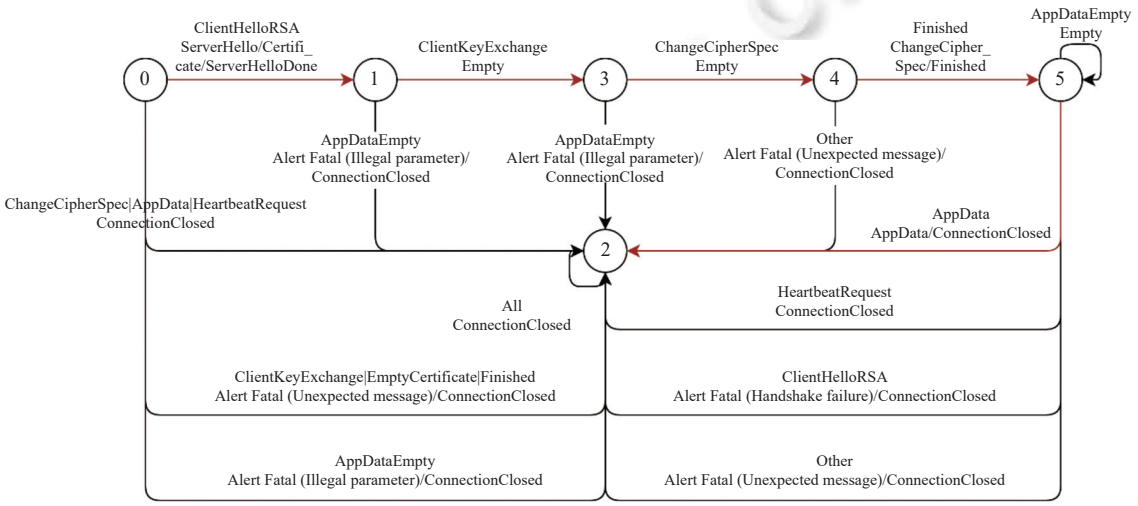

图 3　OpenSSL 1.1.1k 的简化有限状态机

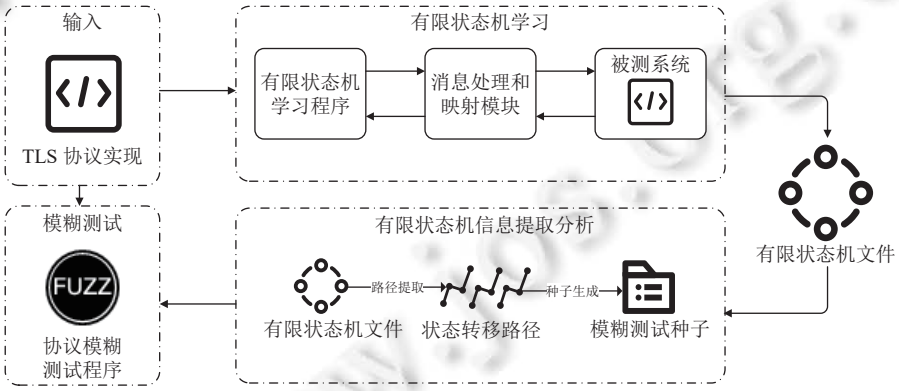

图 4　SNETFuzzer 的系统框架图

以 TLS 版本 1.2 为例, 其官方文档 RFC 详尽地描述了一次完整的 TLS 握手过程中的消息流动. 整个过程启动于客户端向服务端发送 ClientHello 握手请求消息, 携带着客户端支持的 TLS 版本信息、随机数、偏好的密码套件列表及可选的 TLS 扩展信息. 接着, 服务端响应这一请求, 发送 ServerHello 消息, 选择并通报其确定使用的密码套件、随机数、数字证书及进一步的信息 (包括可选的 TLS 扩展). 之后, 服务端继续发送包含其身份验证所用

数字证书的 Certificate 消息, 其中也包括了服务端的公开密钥. 紧接着进行 KeyExchange 密钥交换阶段, 客户端用服务器公钥对一个预共享主密钥进行加密发送, 服务端用私钥进行解密, 使双方共享相同的主密钥. 在随后的通信中, 客户端和服务端基于之前协商的参数计算出一个共同的主密钥, 用以加密通信, 并通过 ChangeCipherSpec 消息相互通告将启用新的加密参数. 完成这些后, 双方互发 Finished 消息, 含有握手阶段所有消息的哈希值用以确认握手是否成功完成. 在握手成功后, 客户端和服务端将利用之前约定的密钥与密码套件来加密和解密数据, 进而开始安全传输应用层的数据 ApplicationData. 为了确保不同密码套件带来的代码路径覆盖差异, 构建有限状态机模型时应涵盖所有 3 种主要握手情景: RSA 加密、DH 加密以及 ECDHE 加密. 这将导致在 ClientKeyExchange 消息中会出现 3 种不同的情况, 具体取决于所采用的密钥交换算法, 从而影响该消息的内容和格式.

然而上述只展示了一种理想情况下的握手, 在很多场景下可能会出现错误情况, 这个时候传输的任意一方都可能发送错误信息并终止消息传输. 通信中的错误分很多种, 比如服务端接收到了非法的字符 (illegal_parameter)、服务端的记录层溢出 (record_overflow)、服务端接收到了未预料到的消息 (unexpected_message)、客户端遇到错误需要终止连接 (close_notify) 等. 同样, 论文会将这些错误消息考虑进去. 如果服务端并未回复消息, 会将这种情况标记为 Empty. 最终得出的用于 TLS 协议有限状态机的字母表如表 1 所示.

表 1　TLS 协议的字母表

输入/输出	字母表
输入 (客户端)	ClientHello, Certificate, ClientKeyExchangeRSA, ClientKeyExchangeDH, ClientKeyExchangeECDH, ChangeCipherSpec, AlertWarningCloseNotify, Finished, ApplicationData
输出 (服务端)	ServerHello,Certificate,ServerKeyExchange,ServerHelloDone,ChangeCipherSpec,Finished,ALERT_FATAL_ILLEGAL_PARAMETER,ALERT_FATAL_UNEXPECTED_MESSAGE,ALERT_FATAL_DECRYPT_ERROR,ALERT_FATAL_DECODE_ERROR

依照 Mealy 机模型的原理, 有限状态机的每一次状态转移都是由输入和输出共同决定的, 这里的输入和输出分别指客户端发送至服务端的数据包以及服务端反馈给客户端的数据包, 这些信息在状态转移图的边上会被标出. 图 5 展示了一次可能会出现的状态转移, 由状态 1 转移到状态 3, 造成状态转移的原因是客户端发送了 ClientKeyExchange 信息而服务端没有回复数据包.

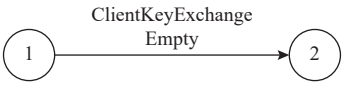

图 5　一次状态转移的示例

值得注意的是, 使用状态机来描述系统往往存在状态定义的粒度选择问题. 如果对状态粒度过细, 那么很有可能由于状态空间过大, 导致在进行系统分析时候出现效率过低甚至是状态爆炸的问题; 而如果以较粗粒度来进行状态的定义, 则可能由于模型未能描述系统的某些关键信息, 而导致系统出现漏报的情况. 为了解决这个问题, 本文以 TLS 协议的 RFC 文档为参考标准, 从中提取了 TLS 握手过程的关键消息类型, 以构建输入输出字母表, 并以此来构建 Melay 机. 如表 1 所示, 字母表中一共包含 9 个输入和 10 个输出, 由此, 既能有效避免状态爆炸的问题, 又保证了状态机能够有效捕获和描述 TLS 握手过程的关键信息.

3.3　有限状态机学习

SNETFuzzer 将基于 LearnLib 和 Angluin's L*算法实现有限状态机学习程序. SNETFuzzer 当中的 Angluin's L*伪代码如算法 1 所示. 在算法 1 中, 对于集合 R 中的每个消息 r, 它被用于测试有限状态机中的每个状态. 也就是说, 对于每对 w=(s, r), 其中 s 是有限状态机中的一个状态, r∈R, 如果从目标协议观察到的输出 $o(s, r)$ 与目标有限状态机的输出 $o'(s, r)$ 不同, 就将 w 添加到 S 中. 在 Angluin's L*算法的观察表中, 状态表示为输入字母表 S 中的前缀字符串. 前缀字符串中的每个输入符号对应于一个实际的 TLS 请求消息. 因此, 如果要导出 $o(s, r)$, 可以将消息 r 附加到前缀字符串, 按顺序发送到目标协议以获得其输出. 然后将此输出与有限状态机的目标有限状态机 $o'(s, r)$

进行比较, 以查看它们是否相同. 在有限状态机学习当中, SNETFuzzer 实际上是作为 TLS 通信中的客户端, 而被测系统作为服务端.

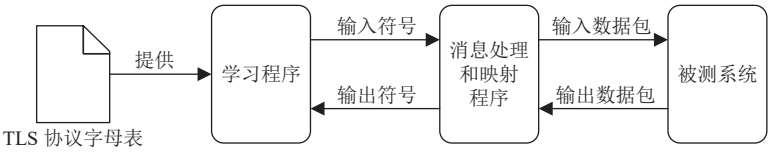

图 6 消息处理和映射程序

算法 1. Angluin's L*算法的伪代码.

输入: *S*: 字母表的集合 (初始为空), 包含输入; *L*: 当前的假设有限状态机 (初始为一个初始状态), 包含输入输出; *R*: 表示等价关系的表 (初始为空);
输出: *L*: 学习之后的有限状态机.

function Angluin's_L*(*S*, *L*, *R*):
1. **while** *L*!=目标有限状态机 **do**:
2. (a) 使用成员查询学习状态机:
3. **for each** *w*∈ *S* **do**:
4. 执行 *w* 在 *L* 上的模拟, 获取模拟的状态序列
5. **if** 模拟状态序列与目标有限状态机不一致 **then**:
6. 将 *w* 添加到 *S*
7. **end if**
8. **end for**
9. (b) 使用等价查询完善假设:
10. **for each** *s* ∈ *S* **do**:
11. 向被测系统查询 *s* 在目标有限状态机上的行为
12. **if** 被测系统返回的行为与 *L* 上的模拟不一致 **then**:
13. 将该字符串 *s* 和目标有限状态机的真实行为添加到 *R*
14. **end if**
15. **end for**
16. (c) 更新假设 *L*:
17. 通过等价关系 *R* 更新 *L* 的状态转移和输入输出
18. **end while**
19. **return** *L*

SNETFuzzer 的学习程序是无法直接传输数据包, 也无法直接处理被测系统返回的数据包. 所以需要如图 6 所示的消息处理和映射程序作为二者的通信中介, 将学习程序生成的输入符号转化成被测系统能够接收和处理的具体数据包, 并将其发送给被测系统; 其次, 将被测系统返回的数据包转换为学习程序能够理解的输出符号. 这里所提到的输入和输出符号 (symbol) 指的是 TLS 协议字母表中列出的字符串, 每个字符串对应着一种 TLS 协议数据包. 模块的任务就是处理这些符号之间的对应关系, 并将其转化为学习程序和被测系统之间可互相理解的形式.

消息处理和映射程序首先将根据有限状态机学习程序发送的输入符号, 生成合法的数据包. 比如有限状态机学习程序发送了输入符号 Certificate, 消息处理和映射程序将依据表 2 的对应关系找到对应的十六进制值并且按

照协议要求、被测系统 IP 和端口、数据包结构生成一个发送 Certificate 请求的数据包, 这个数据包是合法的并且能够被被测系统接收的. 消息处理和映射程序本质上还是一个 TLS 客户端, 需要具有向指定 IP 和端口的对象发送数据包、发起握手、发送 ClientHello 消息、接收 ServerHello 消息、验证服务器证书、发送 ClientKeyExchange 消息、发送 ChangeCipherSpec 消息、完成握手、发送 ApplicationData 等基本功能.

表 2 TLS 协议关键字段十六进制值及其含义

字段名	十六进制	含义
ContentType	0x14	ChangeCipherSpec
	0x15	Alert
	0x16	Handshake
	0x17	Application
	0x18	Heartbeat
Version	0x0300	SSL3.0
	0x0301	TLS1.0
	0x0302	TLS1.1
	0x0303	TLS1.2
	0x0304	TLS1.3
MessageType	0x00	HelloRequest
	0x01	ClientHello
	0x02	ServerHello
	0x0b	Certificate
	0x0c	ServerKeyExchange
	0x0e	ServerHelloDone
	0x10	ClientKeyExchange
	0x14	Finished

学习程序最后会输出描述被测系统的有限状态机文件, 格式为 dot. 有限状态机文件描述了一个图结构, 该图结构包含多个节点和多条单向边. 文件中每一行数据都代表了图中的一次状态转移.

在图 7 中, 第 1 行代码表示从状态 1 到状态 4 的转移, 客户端发送了一个空证书 (CertificateEmpty) 用于身份认证, 之后服务端没有回复; 第 2 行代码表示从状态 1 到状态 2 的转移, 代表客户端发送了一个 ClientKeyExchange 请求数据包并且选择的加密方式是 ECDH, 之后服务端回复了一个表示发生错误的数据包, 具体错误原因是“UNEXPECTED_MESSAGE”, 也就是服务端并没有预料到客户端会在状态 1 下直接发送这个请求, 并且中断了握手, 所以返回了“ALERT_FATAL_UNEXPECTED_MESSAGE”这条消息.

```
…
s1->s4 [label="CertificateEmpty/-"];
s1->s2 [label="ClientKeyExchangeECDH/ALERT_FATAL_UNEXPECTED_MESSAGE|x"];
…
```

图 7 有限状态机文件实例

在不经过处理的情况下, TLS 有限状态机的图复杂且难阅读, 其经过简化之后的示意图如图 3 和后文图 8.

3.4 有限状态机信息提取

在获得了有限状态机之后, 需要对有限状态机中的信息进行提取并且分析, 然后生成模糊测试能够使用的模糊测试种子以指导模糊测试的进行. 如图 9 所示, 在已有工作当中常以一轮正常握手的数据包作为种子, 这些种子所能触发的状态和状态转移有限.

然而通过学习程序生成的有限状态机所描述的状态和状态转移明显更加复杂 (如图 8), 更能够反映系统行为, 并可能覆盖更多的代码. 为最大限度利用有限状态机引导模糊测试的进行, 需要提取有限状态机的每一条从握手开始到握手结束的状态路径.

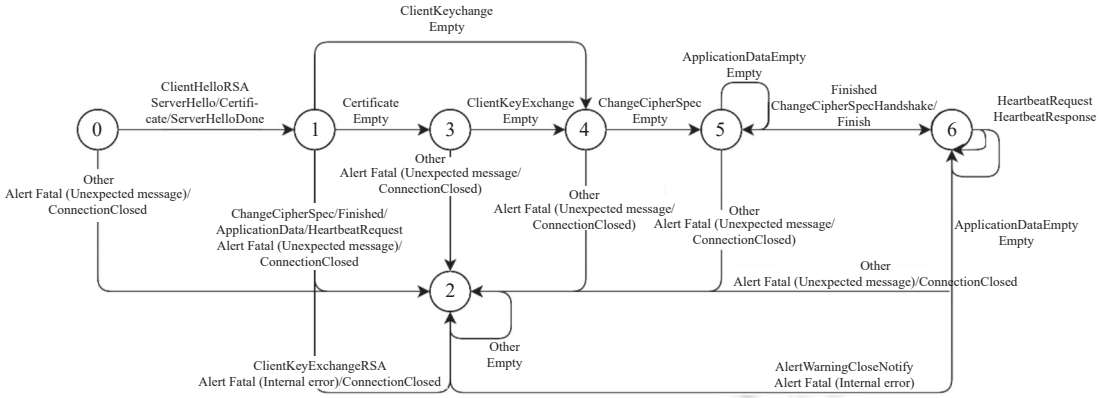

图 8　GnuTLS 的简化有限状态机

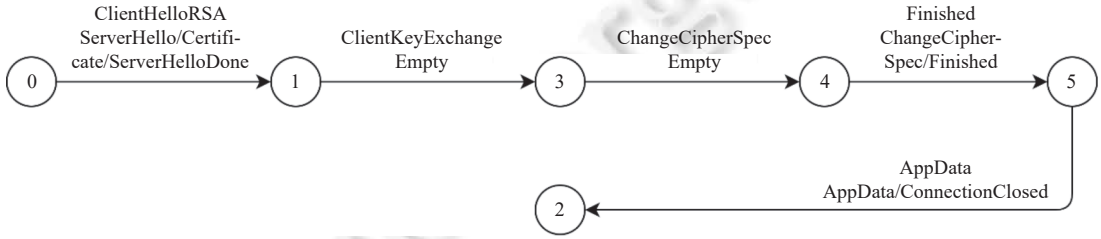

图 9　典型的 TLS 握手流程的状态转移过程

为了获取所有的状态转移路径并获得它们边上的信息, SNETFuzzer 使用一种深度优先搜索算法来获得这些状态转移路径, 与传统针对图的深度有限算法不同的是, TLS 有限状态机的图结构存在以下特点: (1) 图中两个节点之间可能有多条不同的边; (2) 图中存在环路; (3) 图中每个路径必须包括起始节点和终止节点. 算法将都由固定的起始节点开始深度优先搜索, 一直搜索到终止节点为止. 如果遇到两个节点之间都存在多条边, 将每个节点对应的邻接节点列表变成一个字典, 其中键是邻接节点, 而值是边的列表. 如果遇到环路, 则考虑 1 次、2 次和 3 次循环的情况, 将在运行算法之前对于图结构直接进行预处理以避免在运行状态转移路径获取算法的时候遇到死循环.

有限状态机信息提取分析的第 1 个任务就是提取有限状态机中所有的状态转移路径. 然而仅有状态转移路径是无法作为模糊测试种子的, 因为模糊测试种子都是在针对程序的实际输入, 在网络协议模糊测试领域就是实际通信过程中由客户端发送给服务端的数据包. 只有使用真实有效的数据包才能确保协议模糊测试的有效执行, 而不是在一开始就被被测系统判定为非法数据包导致模糊测试无法进行. 所以信息提取分析的第 2 个任务将是根据提取出来的多条状态转移路径生成多个数据包序列, 每个数据包序列都对应一条状态转移路径, 并且要求每个数据包序列中的每个数据包都是有效可用的, 不会被被测系统判断为非法数据包. 之后这些数据包序列会以模糊测试种子的形式输入到 SNETFuzzer 的模糊测试部分当中进行使用.

为实现这个目标, SNETFuzzer 实现了一个简单发包和抓包程序, 能够根据状态转移路径发送构造好的 TLS 数据包到被测系统并且截获双方通信中的数据包序列.

3.5　基于有限状态机引导的网络协议模糊测试

SNETFuzzer 将作为客户端对被测系统进行灰盒模糊测试. 在进行模糊测试的过程中, 将持续监控二进制程序覆盖率, 二进制程序覆盖率和代码覆盖率是成正比的. 在收集覆盖率的同时, 该模块还需要评估每个输入的质量, 具体而言如果一个输入导致一段新的代码路径被执行, 那么它就是一个有趣的输入, 这个输入将会被保留并且分配更多的能量, 也就是分配更多的运行时间和资源.

为使得模糊测试尽可能随机和激进以便更好地探索被测系统的状态空间, SNETFuzzer 会使用一种融合性变

异方式 havoc. havoc 拥有更大程度的随机性, 会在输入的随机位置插入和删除字节, 也对进行大范围字节替换. 它会融合位反转、随即字节插入和删除、模糊字节、数值替换、重组数据块等多种变异方式, 进行更加激进的模糊测试.

在模糊测试过程中, 通过变异操作常常能够进入新的状态. 这些新状态可能是冗余状态、导致程序挂起或崩溃等异常情况 (如图 10), 并且通常伴随对未曾探索过的代码的覆盖. SNETFuzzer 将这些能够探索到新状态的输入视为具有较大潜在价值的, 因为它们有可能揭示系统的潜在漏洞或异常行为. 因此, 在模糊测试的过程中, 针对这些能够触发新状态的输入, 则会为其分配更多的能量, 以促进 SNETFuzzer 更深入地探索系统的状态空间, 提高漏洞发现的可能性. 通过识别并重点关注导致状态变化的输入, 模糊测试可以更有针对性地探索被测系统的状态空间, 提高对潜在漏洞的发现概率.

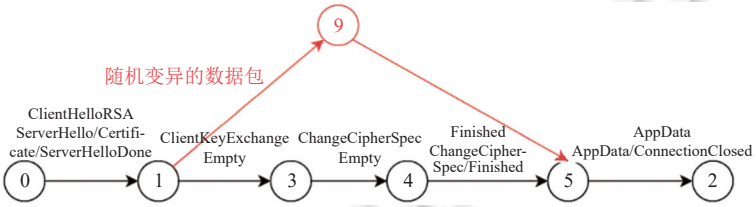

图 10 模糊测试中触发了一个新状态

在网络协议模糊测试领域, 通常采用实际通信中的数据包作为种子, 以确保模糊测试程序能够正确理解协议数据包的结构. 这一做法的重要性在于, 大多数协议实现会对数据包的关键字段进行严格检查. 如果数据包中存在非法内容, 协议实现通常会直接丢弃该数据包, 导致模糊测试失效. 此外, 数据包中还包含了目标系统的 IP 地址和端口号等关键信息. 若这些信息在变异过程中被修改, 就会导致变异后的数据包无法到达目标系统. 因此, 确保模糊测试程序的变异范围十分重要. SNETFuzzer 将避开一些 TLS 重要字段, 以确保每个变异后的数据包都能被被测系统所接受.

4 实验分析

为了评估 SNETFuzzer 的实用性和有效性, 我们研究了以下两个问题.

- RQ1: SNETFuzzer 与其他主流网络协议模糊测试工具相比性能如何?
- RQ2: SNETFuzzer 能否挖掘出网络协议实现当中的漏洞? 能否发现新的漏洞?

4.1 实验环境

表 3 列出了实验当中的具体实验环境配置参数. 为了确保 TLS 协议实现的正确安装和测试, 本实验为每个协议实现构建了独立的虚拟化环境. 这是因为每个 TLS 协议实现通常依赖于不同的软件库和工具, 如果将不同的实现安装在同一环境下, 很容易引发依赖冲突, 影响测试的可靠性和准确性. 为了解决这一问题, 系统使用了 Docker Engine 作为虚拟化环境的实现工具. Docker Engine 允许将每个 TLS 协议实现封装在独立的容器中, 确保了它们之间的隔离性.

4.2 实验评价指标

为了回答 RQ1, 本实验设置了多个指标来综合评估 SNETFuzzer 的性能, 分别为以下 5 个.

(1) 程序覆盖率 (map density): 程序覆盖率指的是从二进制程序插桩中获取的二进制程序覆盖率. 该指标的值越大表示模糊测试的输入覆盖了更多的状态路径. 该指标的格式为 A/B. A 描述了当前输入的覆盖率, 而 B 则表示整个语料库的覆盖率. 较高的程序覆盖率与更全面的代码覆盖相关, 是模糊测试中最重要的评估指标之一.

(2) 有趣的路径 (favored paths): 有趣的路径指的是模糊测试程序认为有趣的路径数量. 该指标的增加代表着模糊测试程序能够发现更多有趣的路径, 直接反映了模糊测试的效果. 它是模糊测试中的一个关键指标, 用于评估测试用例的多样性和发现能力.

(3) 边覆盖数 (new edges on): 边覆盖数指的是模糊测试程序中能够覆盖的边的数量. 在将程序视为流程图的情境下, 每个图代表一个基本代码块, 而边则表示基本代码块之间的跳转. 该指标反映了代码覆盖率信息, 是模糊测试中另一个重要的评估指标.

(4) 路径深度 (geometry[levels]): 路径深度表示通过引导模糊测试能够达到的路径深度. 初始种子被视为“级别 1”, 而经过基于引导的模糊测试变异会使路径的级别逐渐上升. 该参数反映了模糊测试种子从引导方式中获得的收益, 是模糊测试中的一个辅助指标.

(5) 路径稳定性 (geometry[stability]): 路径稳定性指模糊测试用例的稳定性. 随着模糊测试用例的有效性下降而下降. 该参数反映了模糊测试用例的可用率, 是模糊测试当中的辅助指标.

表 3　实验环境的配置参数

配置	名称	参数
硬件配置	CPU型号	AMD EPYC 7742 64核处理器 2.25 GHz
	GPU型号	NVIDIA GeForce RTX 3090
	内存	256 GB
	硬盘	16 TB SSD
软件配置	操作系统	Ubuntu 18.04
	Java JDK	JDK 11
	LearnLib	0.16.0
	Python	3.9
	TLS协议实现	OpenSSL、GnuTLS、MatrixSSL、WolfSSL、MbedTLS
	虚拟化环境	Docker Engine

利用模糊测试挖掘的 TLS 协议实现潜在安全漏洞主要分为以下两种.

(1) 独特的崩溃 (uniq crashes): 独特的崩溃指的是在模糊测试中导致程序出现独特崩溃的次数. 这一指标通过去除重复的崩溃, 以获取唯一的崩溃事件数. 在模糊测试过程中, 独特的崩溃次数反映了程序在不同输入条件下产生的各种崩溃类型. 这些独特的崩溃可能是潜在的安全漏洞迹象, 需要进一步深入分析和验证.

(2) 独特的挂起 (uniq hangs): 独特的挂起指的是在模糊测试中导致程序出现独特挂起的次数. 挂起表示程序未能正常退出, 而是进入了无响应状态. 这种情况可能暗示着潜在的死锁、资源争用或其他与并发性质相关的问题.

4.3　实验数据和对比

AFLNET 是目前主流的协议模糊测试框架之一, 其提供的工具经常被用于有状态的协议模糊测试. 在同样进行 24 h 的模糊测试的情况下, SNETFuzzer 和 AFLNET 在 5 个不同 TLS 协议实现 (OpenSSL、GnuTLS、MatrixSSL、WolfSSL、MbedTLS) 的模糊测试中得到的指标数据如表 4 所示.

表 4　SNETFuzzer 与 AFLNET 的对比实验

TLS协议实现	测试方案	程序覆盖率 (%)	有趣的路径	边覆盖数	路径深度	路径稳定性 (%)
OpenSSL	SNETFuzzer	15.71/19.70	199	289	26	81.7
	AFLNET	4.46/10.67	108	169	17	30.6
GnuTLS	SNETFuzzer	4.89/8.71	158	241	8	73.68
	AFLNET	3.40/7.78	148	234	7	61.46
MatrixSSL	SNETFuzzer	4.85/9.71	230	384	36	60.42
	AFLNET	6.32/7.89	124	190	10	74.91
WolfSSL	SNETFuzzer	7.88/12.50	216	354	7	38.53
	AFLNET	4.23/10.66	143	230	12	30.02
MbedTLS	SNETFuzzer	6.50/10.43	177	287	13	15.90
	AFLNET	5.23/9.84	129	228	16	17.75

实验数据表明, SNETFuzzer 在大部分性能指标上优于 AFLNET. 特别是在程序覆盖率方面, 如图 11 所示, SNETFuzzer 在对 5 种 TLS 协议实现的测试中均展现出高于 AFLNET 的总体语料库覆盖率, 其针对当前输入的覆

盖率也呈现出优势, 除了在 MatrixSSL 实验对比中以 23% 的差距稍落后外, 在其他实现中均占领先地位. 整体而言, SNETFuzzer 在当前输入覆盖率方面比 AFLNET 提高了 76%, 而在总体语料库覆盖率方面则高出了 29%. 在探索有趣路径的能力指标上, SNETFuzzer 同样表现出色, 其平均表现比 AFLNET 高出 53%. 在边覆盖数方面, SNETFuzzer 平均超过 AFLNET 51%. 特别是在 OpenSSL 上, SNETFuzzer 不仅在程序覆盖率上大幅领先, 同时在其他各类性能指标上也呈现出显著的领先优势.

在辅助指标路径深度, SNETFuzzer 在 OpenSSL、GnuTLS、MatrixSSL 的路径深度好过 AFLNET, AFLNET 则在另外两个 TLS 实现当中领先; 类似的, SNETFuzzer 在 OpenSSL、GnuTLS、WolfSSL 的路径稳定度好过 AFLNET, AFLNET 则在另外两个 TLS 实现当中领先. 这证明 SNETFuzzer 虽然在程序覆盖率上领先, 但是不能证明 SNETFuzzer 的模糊测试种子从引导方式中获得了比 AFLNET 更多的收益.

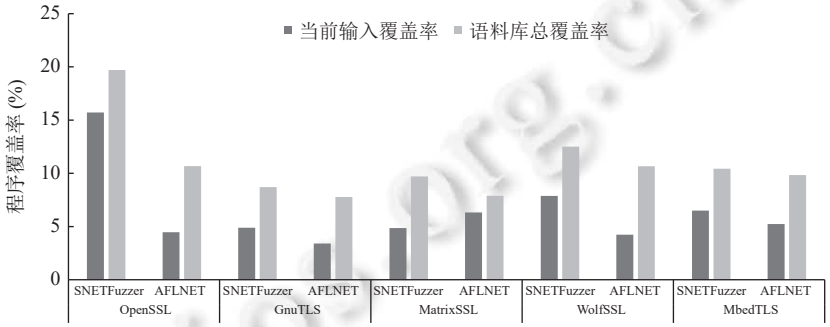

图 11 不同方法的程序覆盖率对比

4.4 实验数据分析

实验结果表明, 在大部分实验环境中, SNETFuzzer 比现有的主流协议模糊测试工具 AFLNET 更为出色, 在多个重要指标上都取得了不错的成绩. 尤其是在能反映代码覆盖率的指标中, 如语料库总覆盖率、有趣路径和边覆盖数等, 均高于目前主流的工作.

表 5 展示了 SNETFuzzer 与 AFLNET 在 OpenSSL 实验当中的部分数据对比. 在 OpenSSL 1.0.1b 版中, SNETFuzzer 在完成有限状态机学习后, 发现了其中存在的 12 个不同状态和 108 个状态转移. 基于这些发现, SNETFuzzer 生成了初始种子集合, 用以映射这些状态转换, 在模糊测试阶段, 它进一步识别了 3 个新增状态和 7 个状态转移. 相比之下, AFLNET 默认配置下的初始种子反映了正确握手过程中的 6 个不同状态和 5 个状态转移, 在随后的模糊测试阶段, 它额外发现了 6 个状态和 10 个状态转移. 尽管 AFLNET 在模糊测试过程中识别出了比 SNETFuzzer 更多的状态和状态转移, 但鉴于 SNETFuzzer 已经通过预先的有限状态机学习步骤发掘了众多状态和转移, 因此 SNETFuzzer 在模糊测试中相较于 AFLNET 在反映代码覆盖率的多个关键指标上领先. 综上所述, SNETFuzzer 一共识别了 15 个状态和 115 次状态转移, 而 AFLNET 在默认设定下则仅发现 12 个状态和 15 次状态转移, 这些显著的差异同样体现在代码覆盖率的相关指标上.

表 5 SNETFuzzer 与 AFLNET 在 OpenSSL 实验当中的部分数据对比

方法	初始的状态/状态转移数	新发现的状态/状态转移数	共发现的状态/状态转移数
SNETFuzzer	12/108	3/7	15/115
AFLNET	6/5	6/10	12/15

SNETFuzzer 比 AFLNET 多发现了 3 个状态和 100 个状态转移. 从有限状态机的角度来看, 这意味着在模糊测试中, SNETFuzzer 实际上呈现了比 AFLNET 多出了 3 个状态节点和 100 条状态转移的边. 这些额外的状态和状态转移可能代表了更多的函数调用、更多的条件判断或者更多的变量赋值等. AFLNET 虽然同样能够识别和追踪网络协议的状态变化, 但是其在不依赖协议字母表的前提下, 所发送的数据包随机性很强, 探索新状态更为随机

和被动. 另一方面, AFLNET 也缺少状态机中已有状态转移信息的引导, 因此导致其最终效果不如 SNETFuzzer.

4.5 漏洞分析

针对问题 RQ2, SNETFuzzer 成功地发现了两个新漏洞和一个已发现漏洞, 本节将详细介绍这些漏洞.

4.5.1 MatrixSSL 漏洞分析

SNETFuzzer 在 MatrixSSL 上一共触发了 68 次异常崩溃, 其中独特的崩溃为 21 次. 经过程序调试, 发现这些漏洞虽然触发位置不同, 但漏洞类型可以分为两类.

图 12 为第 1 种漏洞的调试信息. 在 21 次独特的崩溃中, 有 15 次为此类崩溃. 这是一种非常经典的内存泄露漏洞, 是在使用 C 语言的标准库函数 free() 释放内存时出现的错误消息之一. 这通常是由于程序中某处发生了内存越界、内存重复释放、内存损坏等情况. 与此错误相关的 CWE 是 CWE-762, 即不匹配的内存管理例程.

```
Session Ticket resumption enabled
Listening on port 4433
free(): invalid next size (normal)
Aborted
```

图 12 在 MatrixSSL 上发现的漏洞 1

图 13 为第 2 种漏洞的调试信息. 在 21 次独特的崩溃中, 有 6 次为此类崩溃. 这同样是一种内存泄露漏洞, 是在使用 C 语言的标准库函数 free() 释放内存时出现的错误消息之一. 这种错误通常表示发生了重复释放或者内存损坏的情况. 与这种错误相关的 CWE 是 CWE-415, 即重复释放.

```
Session Ticket resumption enabled
Listening on port 4433
double free or corruption (!prev)
Aborted
```

图 13 在 MatrixSSL 上发现的漏洞 2

经过分析, 这两个漏洞虽然错误类型不同, 但实际上指向同一行代码, 位于 matrixssl.c: 1566. 漏洞触发流程如下, 首先客户端发送了用于触发漏洞的数据包序列到服务端, 共包含了 6 个数据包. MatrixSSL 接收了前面几个数据包并且写入到了内存当中, 在接收到某一个数据包的时候, 服务端想要结束连接, 并且进入了 closeConn 函数, 在 closeConn 函数中, 存在一个 matrixSslDeleteSession 函数负责删除 socket 的连接以释放内存, 而这个函数存在 matrixssl.c 当中. 在 matrixSslDeleteSession 尝试删除并且释放数据缓冲区, 清除任何剩余的用户数据的时候出现了错误. 程序使用了一个 psFree 函数进行已写入 buffer 的数据释放, 这个函数在 psmalloc.h 中被定义. 而 psFree(A, B) 就是 free(A) 函数, 也就说调用了一个危险的 free 函数并且在释放内存之前并没有对释放的对象做检查. 以上两个漏洞并未在 CVE 或 CNVD 网站上登记, 属于新漏洞.

4.5.2 OpenSSL 漏洞分析

SNETFuzzer 于 OpenSSL 上一共触发了 12 次异常崩溃, 其中独特的崩溃也为 12 次. 图 14 展示了对应的调试信息, 经过程序调试后发现, 这 12 个漏洞都指向相同的位置, 即 s23_srvr.c 文件的第 628 行.

进一步分析表明漏洞出现在 ssl23_get_client_hello 函数, 该函数无法正确处理未知的协议而导致崩溃. 相关 CWE 编号为 CWE-476, 即空指针取消引用错误. 该类错误指在程序中使用了空指针进行内存访问操作, 导致程序崩溃或产生未定义的行为.

该漏洞实际上已经登录至 CVE 网站, 其 CVE 号为 CVE-2014-3569. 该漏洞是由于 OpenSSL 构建配置实施不当造成的, 在某些特定版本中, s23_srvr.c 中的 ssl23_get_client_hello 函数无法正确处理使用不受支持的协议的尝试, 这使得远程攻击者可以通过构造特殊数据包来造成拒绝服务攻击. 当 OpenSSL 打开 no-ssl3 选项的时候仍然接收到一个未知协议的握手请求时, SSL 方法会被置为 NULL 并将导致在之后 SSLerr 中的空指针的间接引用.

该漏洞触发需满足更复杂的前提条件, 而 SNETFuzzer 仍旧能构造成功触发该漏洞的测试样例. 在充分体现模糊测试随机性的同时, 证明了 SNETFuzzer 拥有发现复杂内存型漏洞的能力.

```
Using default temp DH parameters
Using default temp ECDH parameters
ACCEPT
140737354049216:error:140760FC:SSL routines:SSL23_GET_CLIENT_HELLO:unknown
protocol:s23_srvr.c:628
```

图 14　在 OpenSSL 上发现的漏洞

4.5.3　漏洞总结

SNETFuzzer 成功挖掘了 2 个新漏洞和 1 个已发现漏洞. 这 3 个漏洞虽属于不同的 CWE 标准, 但均属于内存型漏洞, 这表明了 SNETFuzzer 在发现内存型漏洞方面的表现优秀. SNETFuzzer 成功挖掘两个新漏洞, 并且很有可能是零日漏洞, 这也说明了其具有挖掘出其他工具难以挖掘出的漏洞的能力.

5　未来工作展望

SNETFuzzer 拥有扩展至其他网络协议的可能性, 因为有状态的网络协议具备被建模为有限状态机的可能性. 如果需要将 SNETFuzzer 扩展至其他有状态的网络协议, 则需要向 SNETFuzzer 提供以下先验知识.

(1) 协议的字母表. 用于设计协议的有限状态机模型, 并告知学习程序可发送和接收的数据包类型.

(2) 协议的数据包结构. 用于解析和发送协议数据包, 以及后续的模糊测试.

上述信息都可以从待测协议的标准文档中获得, 因此, 在网络协议测试方面, SNETFuzzer 具有较好的普适性和可扩展性. 下一步工作将尝试将 SNETFuzzer 扩展至其他有状态的网络协议 (如 SMTP、Telnet 等). 这些协议的握手阶段和 TLS 协议类似, 同样具有被建模为有限状态机的潜力.

6　总　结

本文主要提出了一个基于有限状态机引导的网络协议模糊测试方法, 融合了有限状态机学习和基于覆盖引导的模糊测试技术, 并且实现了一个原型系统 SNETFuzzer. 该系统通过有限状态机学习技术对被测网络协议实现进行了深入建模, 从而使得模糊测试工具能够更准确地理解其系统行为. 利用 Angluin's L*算法, 成功实现了针对 TLS 协议的有限状态机学习功能, 使得 SNETFuzzer 能够自动学习协议的系统行为特征及状态转移规律. 在此基础上, SNETFuzzer 基于有限状态机生成了可用于引导模糊测试的种子文件, 并且实现了针对网络协议特征设计的模糊测试工具, 使得经过变异之后的测试用例可靠有效. 在一系列实验中, SNETFuzzer 表现出色, 不仅在覆盖率、路径发现、边覆盖数等主要指标上领先于相关工作, 而且成功地发现了多个安全漏洞, 其中包含两个新漏洞, 证明了其实用性和有效性.

References:

[1] Xiong G, Tong JY, Xu Y, Yu HL, Zhao Y. A survey of network attacks based on protocol vulnerabilities. In: Proc. of the 2014 Asia-Pacific Web Conf. Changsha: Springer, 2014. 246–257. [doi: 10.1007/978-3-319-11119-3_23]

[2] Zhang X, Li ZJ. Survey of fuzz testing technology. Computer Science, 2016, 43(5): 1–8, 26 (in Chinese with English abstract).

[3] Pistoia M, Chandra S, Fink SJ, Yahav E. A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM Systems Journal, 2007, 46(2): 265–288. [doi: 10.1147/sj.462.0265]

[4] Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. on Software Engineering, 2009, 35(5): 684–702. [doi: 10.1109/TSE.2009.28]

[5] Fang DL, Song ZW, Guan L, Liu PZ, Peng AN, Cheng K, Zheng YW, Liu P, Zhu HS, Sun LM. ICS3Fuzzer: A framework for discovering protocol implementation bugs in ICS supervisory software by fuzzing. In: Proc. of the 37th Annual Computer Security Applications Conf. New York: Association for Computing Machinery, 2021. 849–860. [doi: 10.1145/3485832.3488028]

[6] Aichernig BK, Muškardin E, Pferscher A. Learning-based fuzzing of IoT message brokers. In: Proc. of the 14th IEEE Conf. on Software Testing, Verification and Validation. Porto de Galinhas: IEEE, 2021. 47–58. [doi: 10.1109/ICST49551.2021.00017]

[7] Zuo FL, Luo ZX, Yu JZ, Liu Z, Jiang Y. PAVFuzz: State-sensitive fuzz testing of protocols in autonomous vehicles. In: Proc. of the 58th ACM/IEEE Design Automation Conf. San Francisco: IEEE, 2021. 823–828. [doi: 10.1109/DAC18074.2021.9586321]

[8] Gao X, Saha RK, Prasad MR, Roychoudhury A. Fuzz testing based data augmentation to improve robustness of deep neural networks. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: Association for Computing Machinery, 2020. 1147–1158. [doi: 10.1145/3377811.3380415]

[9] Godefroid P, Levin MY, Molnar D. SAGE: Whitebox fuzzing for security testing. Communications of the ACM, 2012, 55(3): 40–44. [doi: 10.1145/2093548.2093564]

[10] Stephens N, Grosen J, Salls C, Dutcher A, Wang RY, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the 23rd Annual Network and Distributed System Security Symp. San Diego: NDSS, 2016. [doi: 10.14722/ndss.2016.23368]

[11] Feng XT, Sun RX, Zhu XG, Xue MH, Wen S, Liu DX, Nepal S, Xiang Y. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. In: Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security. New York: Association for Computing Machinery, 2021. 337–350. [doi: 10.1145/3460120.3484543]

[12] Fioraldi A, Mantovani A, Maier D, Balzarotti D. Dissecting American fuzzy lop: A Fuzzbench evaluation. ACM Trans. on Software Engineering and Methodology, 2023, 32(2): 52. [doi: 10.1145/3580596]

[13] Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: Association for Computing Machinery, 2017. 2329–2344. [doi: 10.1145/3133956.3134020]

[14] Gan ST, Zhang C, Qin XJ, Tu XW, Li K, Pei ZY, Chen ZN. CollAFL: Path sensitive fuzzing. In: Proc. of the 2018 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2018. 679–696. [doi: 10.1109/SP.2018.00040]

[15] Xie XF, Ma L, Juefei-Xu F, Xue MH, Chen HX, Liu Y, Zhao JJ, Li B, Yin JX, See S. DeepHunter: A coverage-guided fuzz testing framework for deep neural networks. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: Association for Computing Machinery, 2019. 146–157. [doi: 10.1145/3293882.3330579]

[16] Lemieux C, Sen K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proc. of the 33rd IEEE/ACM Int'l Conf. on Automated Software Engineering. Montpellier: IEEE, 2018. 475–485. [doi: 10.1145/3238147.3238176]

[17] Fioraldi A, Maier D, Eißfeldt H, Heuse M. AFL++: Combining incremental steps of fuzzing research. In: Proc. of the 14th USENIX Conf. on Offensive Technologies. Berkeley: USENIX Association, 2020. 10.

[18] He JX, Balunović M, Ambroladze N, Tsankov P, Vechev M. Learning to fuzz from symbolic execution with application to smart contracts. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: Association for Computing Machinery, 2019. 531–548. [doi: 10.1145/3319535.3363230]

[19] Metzman J, Szekeres L, Simon L, Sprabery R, Arya A. Fuzzbench: An open fuzzer benchmarking platform and service. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: Association for Computing Machinery, 2021. 1393–1403. [doi: 10.1145/3468264.3473932]

[20] Pham VT, Böhme M, Roychoudhury A. AFLNET: A greybox fuzzer for network protocols. In: Proc. of the 13th IEEE Int'l Conf. on Software Testing, Validation and Verification. Porto: IEEE, 2020. 460–465. [doi: 10.1109/ICST46399.2020.00062]

[21] Song CX, Yu B, Zhou X, Yang Q. SPFuzz: A hierarchical scheduling framework for stateful network protocol fuzzing. IEEE Access, 2019, 7: 18490–18499. [doi: 10.1109/ACCESS.2019.2895025]

[22] Andronidis A, Cadar C. SnapFuzz: High-throughput fuzzing of network applications. In: Proc. of the 31st ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: Association for Computing Machinery, 2022. 340–351. [doi: 10.1145/3533767.3534376]

[23] De Ruiter J, Poll E. Protocol state fuzzing of TLS implementations. In: Proc. of the 24th USENIX Conf. on Security Symp. Washington: USENIX Association, 2015. 193–206.

[24] Fiterau-Brostean P, Jonsson B, Merget R, de Ruiter J, Sagonas K, Somorovsky J. Analysis of DTLS implementations using protocol state fuzzing. In: Proc. of the 29th USENIX Security Symp. Berkeley: USENIX Association, 2020. 2523–2540.

[25] Fiterau-Brostean P, Jonsson B, Sagonas K, Tåquist F. Automata-based automated detection of state machine bugs in protocol implementations. In: Proc. of the 30th Annual Network and Distributed System Security Symp. San Diego: NDSS, 2023.

[26] Zou YH, Bai JJ, Zhou JL, Tan JF, Qin CG, Hu SM. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In: Proc. of the 2021 USENIX Annual Technical Conf. Berkeley: USENIX Association, 2021. 489–502.

[27] Natella R. STATEAFL: Greybox fuzzing for stateful network servers. Empirical Software Engineering, 2022, 27(7): 191. [doi: 10.1007/s10664-022-10233-3]

[28] Dierks T, Rescorla E. RFC 5246: The transport layer security (TLS) protocol version 1.2. 2008. https://www.rfc-editor.org/rfc/rfc5246

[29] Raffelt H, Steffen B, Berg T. LearnLib: A library for automata learning and experimentation. In: Proc. of the 10th Int'l Workshop on

Formal Methods for Industrial Critical Systems. Lisbon: Association for Computing Machinery, 2005. 62–71. [doi: 10.1145/1081180.1081189]

[30] Angluin D. Learning regular sets from queries and counterexamples. Information and Computation, 1987, 75(2): 87–106. [doi: 10.1016/0890-5401(87)90052-6]

附中文参考文献:

[2] 张雄, 李舟军. 模糊测试技术研究综述. 计算机科学, 2016, 43(5): 1–8, 26.

袁斌(1990－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为物联网安全, 逻辑漏洞检测, 安全检测.

张驰(1999－), 男, 硕士生, 主要研究领域为软件定义网络, 软件定义网络安全.

任家俊(1998－), 男, 硕士生, 主要研究领域为软件安全, 网络协议安全.

邹德清(1975－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为云计算安全, 网络攻防与漏洞检测, 软件定义安全与主动防御, 大数据安全与人工智能安全, 容错计算.

陈群锦明(2000－), 男, 硕士生, 主要研究领域为软件安全, 网络协议安全.

金海(1966－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为计算机系统结构, 虚拟化技术, 集群计算, 网格计算, 并行与分布式计算, 对等计算普适计算, 语义网, 存储与安全.