计算机科学
COMPUTER SCIENCE

SSFuzz:状态敏感的网络协议服务灰盒模糊测试技术

林家含, 冉猛, 彭建山

引用本文

林家含, 冉猛, 彭建山. SSFuzz:状态敏感的网络协议服务灰盒模糊测试技术[J]. 计算机科学, 2024, 51(12): 71-78.

LIN Jiahan, RAN Meng, PENG Jianshan. SSFuzz:State-sensitive Greybox Fuzzing for Network Protocol Services [J]. Computer Science, 2024, 51(12): 71-78.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231000018

SSFuzz:状态敏感的网络协议服务灰盒模糊测试技术

林家含　冉　猛　彭建山
信息工程大学网络空间安全学院　郑州 450001
(ljh_studypp@outlook.com)

摘　要　网络协议服务作为个人设备与互联网交互的接口,其脆弱性严重威胁用户的隐私和信息安全。最先进的网络协议灰盒模糊测试工具在代码覆盖率的基础上引入了状态反馈,通过分析网络协议服务的状态信息,进一步筛选有效的变异种子。但是,不同的模糊测试工具对网络协议服务状态有着不同的定义,如 AFLNET 通过分析服务器响应数据包的内容提取状态,StateAFL 定义长寿命内存作为程序状态。在状态收集上,SGFuzz 通过分析 Enum 类型数据定义,识别状态变量的赋值语句并插桩。然而,SGFuzz 无法识别状态变量的间接赋值语句,对于状态变量的识别并不全面。同时,在构建状态机时,不同的模糊测试技术对状态机节点有着不同的定义,难以在同一个模糊测试工具上同时使用多种状态收集策略。此外,在实验设计上,现有的方案倾向于比较相同时间内的代码覆盖率情况。但是,代码覆盖率的增长受到多方面因素的影响,如吞吐量、种子筛选策略等。相同时间内的代码覆盖率实验适用于不同模糊测试工具之间的比较,对于其中单个模块的改进实验则不适用。针对以上问题,提出了 SSFuzz。具体地,SSFuzz 研究了基于状态变量的插桩方式,依据代码编译过程中的抽象语法树信息,识别状态变量赋值的间接赋值方法,能够更精准地对状态变量赋值语句进行插桩;其次,SSFuzz 对用于指导状态筛选的状态机进行了定义,该方法有助于不同的状态反馈策略共同构建状态机。实验结果表明,SSFuzz 能够实现对大部分网络协议服务的插桩,并且相较于 SGFuzz,能够实现对间接赋值语句的插桩。此外,讨论了适用于评估状态机有效性的实验方法,并证明了 SSFuzz 能够以更少的测试样例数量达到更高的路径覆盖率。

关键词:网络协议;模糊测试;程序插桩;状态反馈
中图分类号　TP309.1

SSFuzz:State-sensitive Greybox Fuzzing for Network Protocol Services

LIN Jiahan,RAN Meng and PENG Jianshan
School of Cyberspace Security,Information Engineering University,Zhengzhou 450001,China

Abstract　The vulnerability of network protocol services,as the interface for personal devices to interact with the Internet,poses a serious threat to users' privacy and information security. The state-of-the-art network protocol grey-box fuzzy testing tools introduce state feedback on the basis of code coverage,which further filters effective variant seeds by analysing the state information of network protocol services. However,different fuzz testing tools have different definitions of network protocol service state,e.g.,AFLNET extracts state by analysing the contents of server response packets,and StateAFL defines long-lived memory as program state. For state collection,SGFuzz identifies assignment statements of state variables and inserts stakes by analysing Enum type data definitions. However,SGFuzz cannot identify the indirect assignment statements of state variables,and the identification of state variables is not comprehensive. Meanwhile,when constructing state machines,different fuzzy testing techniques have different definitions of state machine nodes,making it difficult to use multiple state collection strategies on the same fuzzy testing tool at the same time. In addition,in terms of experimental design,existing schemes tend to compare the code coverage situation over the same period of time. However,the growth of code coverage is affected by various factors,such as throughput,seed screening strategies,etc. Code coverage experiments within the same time are suitable for comparison between different fuzzy testing tools,not for improvement experiments of individual modules in them. In this paper,we propose SSFuzz. Specifically,SSFuzz first investigates the state-variable based staking approach,which identifies the indirect assignment method of state-variable assignment based on the abstract syntax tree information during the code compilation process,and is able to stake state-variable assignment statements more accurately. Secondly,SSFuzz defines the state machine for guiding state screening,which is able to facilitate the co-construction of state machines by different state feedback strategies. Experiments show that SSFuzz ena-

到稿日期:2023-10-07　返修日期:2024-03-05
基金项目:河南省重大科技专项(221100240100)
This work was supported by the Henan Province Science and Technology Major Project(221100240100).
通信作者:彭建山(jxpjs@163.com)

bles staking of most network protocol services, and compared to SGFuzz, indirect assignment statements. In addition, we discuss experimental methods suitable for evaluating the effectiveness of state machines and demonstrate that SSFuzz is able to achieve higher path coverage with a smaller number of test samples.

Keywords Network protocol, Fuzzing, Program instrument, Statement feedback

1 引言

网络协议服务是为了给应用程序提供网络接口，直接向用户提供服务而设计的网络通信服务程序。网络协议服务器公开暴露在本地局域网或互联网上，允许所有可访问到服务器的用户与服务器通信。然而，其在带来便捷服务的同时，也给恶意用户带来了更多的机会。最具代表性的网络协议漏洞利用事件包括：2001 年红色代码蠕虫(Code-Red worm)利用微软 IIS 服务缓冲区溢出漏洞，通过 HTTP(超文本传输协议，Hyper Text Transfer Protocol)协议发送带有恶意功能的代码，仅在 14 h 内就感染了 30 多万台计算机；2017 年，WannaCry 勒索软件利用 Windows SMB 服务漏洞进行恶意攻击，造成的损失达数亿美元。

根据网络协议的特性，网络协议模糊测试技术主要存在以下挑战。1)网络协议服务依赖套接字通信，相较于 I/O 通信，限制了模糊测试的吞吐量。模糊测试技术的基本思想是通过快速生成大量的测试样例并输入至 PUT(Program Under Test，待测程序)从而触发潜在的安全漏洞。然而相较于通过标准 I/O 或文件读写的交互方式，网络通信耗时长，且快速连续的通信数据包可能被操作系统抛弃。2)网络协议服务是一种有状态的服务程序，有状态程序的执行受多个输入数据共同影响。有状态程序指程序的当前响应受程序当前状态影响的程序。典型的如 FTP 协议，该协议服务首先需要用户提供包含用户名及密码的数据包，验证成功后进入登录状态，此时 FTP 服务才进一步接受并处理文件传输、用户注销等数据包。相对地，无状态程序指程序执行期间各个输入之间互不影响的程序，最典型的如 ffmpeg，jpeg 等格式解析型应用程序。与无状态程序对各个输入的处理相互独立不同，有状态程序对同一组输入的不同组合也可能存在不同的处理模式。非状态灰盒模糊测试器不适用于网络通信协议脆弱性检测[1]，以 AFL[2]为代表的传统灰盒模糊测试器不考虑通信数据前后关系，这将严重影响模糊测试的效率。3)网络通信数据包的格式是受到严格约束的，不符合格式规范的数据包很难深入代码的业务逻辑，在格式解析部分会被丢弃。目前主流的灰盒模糊测试技术主要采用基于遗传算法的变异策略：启发式地打乱测试样例的内部文件结构，以尝试触发 PUT 异常状态。而变异策略的好坏直接决定了代码覆盖率的增长效率，效率较低的变异策略会产生大量的无效测试样例，即对覆盖率增长不具备指导意义或没有成功覆盖目标代码块的数据包。

针对网络通信数据开销大、严重影响模糊测试吞吐量的问题，现有的研究通过网络选择性仿真技术[3]、I/O 同步技术[4]、快照[5-6]等方式，提高了模糊测试器与目标服务程序之间的通信效率。针对状态信息搜集问题，最先进的研究通过分析服务响应数据包[7]、状态变量监控[4,8]，实现了模糊测试过程间的动态状态信息收集；还有部分研究人员通过 NLP 技术实现对 RFC 文档的解析[9]，在模糊测试前完成状态机提取。而针对测试样例生成问题，主要有人为定义测试样例格式[10-12]、人工智能筛选测试样例[13-15]等方式。

本文针对网络协议灰盒模糊测试中的状态信息收集及状态机构建问题，提出了 SSFUZZ——一种基于状态变量监控的网络协议灰盒模糊测试技术。SSFuzz 基于 Macro(宏)即 Enum(枚举)常量的赋值关系，对状态变量进行了定位及插桩；通过插桩提取状态变量属性，实现对不同状态变量的区分，进而实现动态监控。此外，SSFuzz 对状态机节点进行了定义，能够有效提高测试样例生成的有效性。

本文的主要贡献如下：

1)提出了一种基于状态变量监控的网络协议状态感知技术，并实现了原型 SSFuzz。

2)对基于状态变量的状态搜集方法进行了补充。通过分析网络服务协议实现的抽象语法树，对状态变量间接赋值方法进行了补充，能够实现更加全面的状态变量识别。

3)对网络协议模糊测试研究中的状态机进行了定义。将一次信息交互行为所获取的全部状态反馈定义为一个状态节点，并提出通过组合的方式，可以支持多种信息反馈共同构建状态机，以实现更易于兼容的状态机构建模型。

4)基于 AFLNET 实现了 SSFuzz 原型，实验表明 SSFuzz 能够实现更精确的状态变量插桩，并且能够更有效地指导模糊测试过程中的种子筛选及路径发现。

2 相关工作

2.1 网络协议状态描述

1948 年，香农提出有限状态机数学模型。有限状态机被广泛应用于计算机科学相关的各个领域。在网络协议的行为描述上，有限状态机是目前使用最广泛的状态转换描述模型。

有限状态机由有限数量的状态节点和状态转换关系组成。以 TCP 协议为例，TCP 的状态机通常使用两种状态(连接建立期和连接维护期)来描述 TCP 连接的建立和关闭过程。在建立连接阶段，TCP 会通过握手协议，在客户端和服务器之间建立一个 TCP 连接，此时 TCP 处于连接已建立的状态；而在下线时，TCP 会执行关闭连接的过程。在这个过程中，TCP 会将连接断开并进入关闭状态，等待远程端 ACK 确认。

无论是 TCP，UDP，HTTP 还是 FTP 等协议，都可以使用有限状态自动机来描述完整的通信过程。有限状态自动机是一种简单而强大的工具，它能够对协议进行清晰的描述，帮助程序员更好地建模和实现网络协议。

在网络协议设计上，状态转换关系可能包含复杂的信息序列或条件，但对于自动化的脆弱性检测方法而言，则需要对这些复杂条件进行抽象描述，使其能够简单适配到多个目标

上。因此，现有的网络协议模糊测试技术大多采用实际网络通信数据，即测试过程中产生的测试样例，作为状态之间的转换关系，这样的好处是易于复现状态转化过程。而对于状态节点，不同的模糊测试器根据不同的理解有着多样化的定义。不仅如此，受限于模糊测试过程中的吞吐量等难以精确控制的多种因素，如何衡量状态策略的好坏成为一个复杂的问题。

SSFuzz 将一次信息交互所造成的状态变化作为一个状态节点，这样的好处是能够在保留状态变化感知的敏感度下，尽可能降低整个状态机的复杂度，提高模糊测试器的效率。

2.2 网络协议状态感知

过去的网络协议模糊测试技术仅适用于某项协议或某类协议[16-19]，不具有通用性。这是因为为了更好地指导模糊测试器的消息序列生成，开发人员依赖大量的人工工作去根据目标协议的代码实现或公开的协议实现规范（如 RFC 文档）构建用于描述网络协议服务工作期间的状态转换关系。但是，由于各个网络协议在设计上的不同，不存在统一的有限状态机去进行综合描述。因此，网络协议状态机的构建是限制网络协议模糊测试技术通用性的重要因素之一。

目前，自动化构建状态机的技术包括静态构建与动态构建。静态构建自动机依赖目标现有的公开资料，如 RFC 文档等。有研究将 NLP 技术应用于 RFC 文档的解析过程，并提取自动机。静态构建的方法受限于公开资料描述是否详尽，对于私有协议，该方法则不适用。不仅如此，协议的开发人员在实现过程中并不一定完全依照规范文档，而是会根据自身理解或现实业务场景的需要对资源进行调配，导致自动化生成的状态机不一定能够准确描述程序的实际行为。

动态构建技术通过收集协议在工作过程中的报文信息、状态反馈等方式，对程序状态进行高级抽象，并设计一定的算法动态构建程序状态自动机。与静态方法相比，动态构建方法源自程序执行过程的行为反馈，构建的自动机一定能够描述出程序的部分行为，但是其精确性取决于开发人员对程序状态的描述是否合适。

因此，网络协议的状态感知技术是网络协议模糊测试技术的研究重点之一。

2020 年，AFLNET[7] 通过协议在接收到通信数据包后的状态码反馈，实现对协议服务程序当前状态的区分，如 HTTP 协议的状态码 200，401，402 等。这种方法的优点是实现简单，监控机通过简单的正则匹配或字符串解析就能够获得程序的状态信息。但是其缺点也十分明显，该方式只能捕获到程序状态的显示反馈，无法反映出程序的隐式状态。假设协议服务在接收到数据包后，不输出状态码信息，AFLNET 则无法实现协议的状态感知。在此基础上，PS-Fuzz[20] 在模糊测试前，通过静态分析程序内含的错误信息反馈字符串，实现对程序数据包处理相关函数的定位，并进一步进行插桩，实现状态信息获取。虽然 PS-Fuzz 在 AFLNET 的基础上通过静态分析进一步获取到了更多的程序内部信息，但其与 AFLNET 相同，一旦存在隐式反馈，则会大大影响状态感知效果。

2021 年，Roberto 提出 StateAFL[21]，对程序的状态进行高级抽象。StateAFL 通过在内存分配及网络 I/O 操作部分进行插桩，实现对内存空间相关信息的获取，如内存地址及大小。进一步地，StateAFL 将全局变量、堆变量等在程序运行过程中长期存在的变量空间定义为长寿命内存。相对地，因为局部变量保存在栈中，所以在函数推出后就自然销毁的内存空间则不属于长寿命内存。StateAFL 通过快照的方式保存长寿命内存的数值，并计算其 Hash 值，作为程序状态的高级抽象，并进一步以此实现程序状态自动机的构建。该方法的基本思想是：对于程序的完整过程而言，函数的内部逻辑主要受函数调用时的参数、对外交互数据以及长寿命内存的共同影响。因此，当程序进入业务主循环时，长寿命内存作为影响程序流程的主要因素，可以代替协议状态参数对程序行为进行描述。

2022 年，Ba 等[8] 统计了 github 上使用量最高的 50 个协议的开源实现项目的状态表示方法，发现有 44 种均采用 enum 枚举类型对程序状态进行描述。基于这个基础事实，他们提出了 SGFuzz。在协议程序编译前，SGFuzz 使用模式匹配的方法找出协议项目里的所有 enum 类型数据的定义，并进一步在所有使用 enum 类型的代码处进行源码级插桩。实验表明，该方法在代码覆盖速度上能够达到基准 Fuzzer 的 2 倍。

2.3 程序反馈信息

模糊测试技术在代码覆盖率被提出后得到飞速发展。尽管模糊测试技术已经被长期应用于网络协议服务测试，但是在 AFLNET 被正式提出之前，主流的网络协议模糊测试器仍是基于生成的黑盒模糊测试技术[22-27]，不仅依赖测试人员对样本生成格式的规范，且测试效率较差。相比之下，采用覆盖率反馈的方式更具实用性，同时自动化程度更高。

2020 年，Pham 等提出的 AFLNET 首次将代码覆盖率驱动的灰盒模糊测试技术应用于网络协议脆弱性检测中，同时使用了代码覆盖率驱动及状态驱动。

代码覆盖率对于灰盒模糊测试技术而言，是最为直接的度量标准之一。其基本思想是模糊测试器无法触发没有执行的代码漏洞。因此，如何快速提高代码覆盖率始终是灰盒模糊测试技术重点研究的内容之一。

状态驱动可归属于数据流驱动技术。数据流驱动技术[28-29] 是通过对内存变化的监控，实现对控制流不可见的行为的捕获，并直观地表示程序行为。如图 1 所示，构造消息序列 $A \rightarrow B$ 与 $B \rightarrow A$，此时，代码覆盖率驱动的插桩策略无法区分出此类事件，而状态驱动的插桩策略可以通过监控 handle 变量，实现对程序行为的区分。

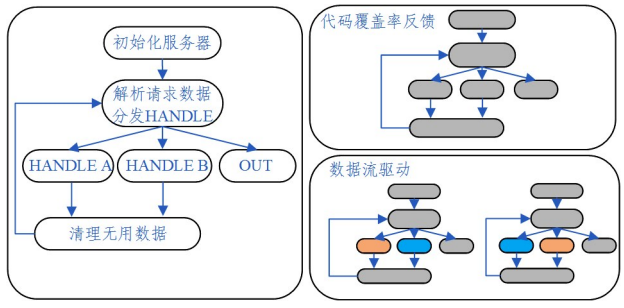

图 1 代码覆盖与数据流驱动

Fig. 1 Code coverage and data flow driven

代码覆盖率是对程序执行指令分支的具体描述，而状态是从程序行为上对程序的概括。状态覆盖率可以帮助测试人员评估系统的鲁棒性和稳定性，因为只有在不同的状态下应用程序才能反映出其真正的行为。

3 方案设计

本章将详细介绍 SSFuzz 的设计和实现。SSFuzz 设计的目标是获取更精准的服务协议状态感知以构造更符合程序行为的有限状态自动机。实现这一目标有两个关键挑战。

第一个挑战是如何正确定位状态变量并实现状态变量监控。2021 年，SGFuzz 证明了通过对枚举变量的插桩能够实现对程序状态的监控，其使用的算法能够覆盖所有类型的枚举变量引用。本文的观点是，不同的枚举变量在使用过程中是可以进一步区分的，例如，当程序接收到某个引起状态改变的数据包时，负责管理该事件的结构体变量内，用于表示程序状态的变量就会发生变化，但是，在内存结构上，变量地址一般是不会发生变化的。基于这一观点，我们在插桩过程中将变量地址作为程序反馈的信息之一，并重新设计了状态树构建算法。通过这种方式，能够实现对程序状态的精确监控，从而生成更加准确的有限状态自动机。

第二个挑战是如何根据第一步获取的状态变量信息构建适用于模糊测试技术的有限状态自动机。SSFuzz 将状态变化分为两种模式。一是仅保留每一轮状态收集时状态变量的最终值，其主要思想是下一轮数据交互仅与程序当前状态相关。二是保留每一轮状态变量变化的完整过程，这有益于对不同响应事件进行区分。

图 2 展示了 SSFuzz 的整体框架。

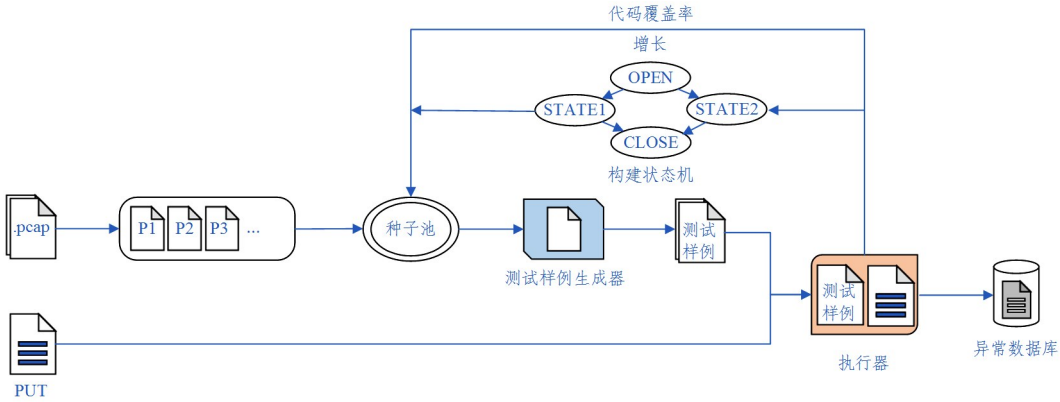

图 2 SSFuzz 系统架构

Fig. 2 System architecture of SSFuzz

3.1 状态赋值代码识别与插桩

SSFuzz 解决的第一个问题是网络协议状态变量识别及状态信息反馈。在协议实现过程中，开发人员通常使用特定的状态变量实现对协议服务当前状态的表示，并将其应用于协议状态转换的指导过程。对状态变量的监控能够有效获取准确的协议服务的运行时状态。但是，不依赖人为工作，自动化地识别状态变量是一个技术难题。

SGFuzz 通过总结 50 个网络协议项目实现，归纳出在协议实现过程中，通常采用枚举变量对程序状态进行标记。在此基础上，SGFuzz 使用格式化匹配的方法实现了对协议项目源代码中所有使用枚举类型数据的代码的定位，并以插入源代码的方式进行插桩。在此基础上，本文通过对源代码的抽象语法树进行解析，能够在保留项目代码的语义信息的基础上，实现更高精度的状态变量识别及插桩。

在网络协议服务的开发过程中，为了更加方便地管理服务程序的状态变化，开发人员通常使用一个专门用于标识程序状态的全局变量或堆变量来对空间内存进行管理。StateAFL 将此类跨越整个会话的内存定义为长生命周期变量，并通过计算长生命周期变量的哈希值来区分不同的状态空间。但是，计算完整内存哈希的开销很大，会严重影响模糊测试效率。对此，本文提出在模糊测试前期可以采用静态插桩的方式对状态变量进行识别与插桩，提取状态变量的地址信息，在程序运行过程中，实现对内存的精准监控，从而避免额外的开销。

在实现上，SSFuzz 通过在编译器前端，在代码解析为抽象语法树阶段完成插桩。传统的插桩方式通过在代码编译为机器码阶段（如 AFL-gcc）或通过编译过程的中间 IR 过程（如 AFL-LLVM）完成插桩。但是，枚举类型数据在编译器前端就会被语义优化为具体常量，在中间 IR 阶段难以被当作 Enum 变量区分。相对地，在编译器前端，我们可以对 Enum 数据进行有效识别。具体来说，SSFuzz 首先将 Macro 类型数据替换为 Enum 类型，通过实验发现，这种改变不会引起代码完整性缺失，也不会造成编译后的程序整体堆栈结构变化；然后 SSFuzz 识别所有的 Enum 变量的赋值代码，并插入桩代码，提取二元组数据（*address*，*value*）。在构建有限状态自动机时，我们根据 address 对数据进行区分，防止不同状态变量错误影响自动机的有效性。

在枚举变量的赋值上，主要考虑两种赋值方式：直接赋值与函数赋值。直接赋值的方式表示为 $A=B$，对于此类赋值代码的插桩，我们只需插入桩代码直接提取 A_{address} 和 B_{value}；函数赋值表示为 $Func(A,B)$ 或 $Func(B)$。该情况下，我们需要在函数调用处记录 B_{value}，并深入函数内部，在具体的赋值语句处提取 A_{address}。逻辑上，在函数调用处，缺少人工分析的前提下，我们无法知道用于保存状态的枚举变量是参数中的哪一项，

甚至当开发人员使用类的成员变量进行传递时，参数的传递不依赖函数调用时的堆栈。为解决上述问题，本文设计并实现了算法 1。所提插桩算法遍历抽象语法树，当遇到直接赋值代码，则直接插入桩代码；当遇到函数调用代码，首先标记形式化参数为枚举变量，再继续深入函数内部寻找赋值代码。

算法 1 Collect process statements

```
Input：AST：程序抽象语法树
function Instrument_function(AST)
1.  for Stmt∈AST do
2.  if isAssign(Stmt) and useEnum(Stmt) then
3.    InsertCode()
4.  else if isCallExpr(Stmt) and ArgIsEnum(Stmt) then
5.    Func ← get_function_decl(Stmt)
6.   if isInstrumented(Func) then
7.      continue
8.    else
9.     Function_AST ← get_ast(Func)
10.     Instrument_function(Func)
11.  end if
12.  end if
13. end for
end function
```

3.2 状态节点定义与状态机构建

对于状态机作用于灰盒模糊测试上，SSFuzz 有着不同的观点。SSFuzz 认为，对于灰盒模糊测试技术而言，评估模糊测试器好坏的关键在于代码覆盖率增长速度及 BUG 检测效果。状态这一概念的提出源于黑盒模糊测试技术发展过程中为指导测试样例生成而引进的关键技术。相较于代码覆盖率明确的边与基本块两种维度，状态可根据开发人员不同的理解进行不同维度的定义。因此，SSFuzz 提出一种状态收集的方式使其能够适配多数网络协议，并对状态节点进行定义，使不同的状态收集策略之间进行统一的适配。

在完成 3.1 节中对状态变量赋值语句的插桩编译后，每当 SUT 运行插桩代码时，都会将状态变量地址及数值写入共享内存。模糊测试器在每一轮发送通信数据前，都会读取并清空共享内存，以保证完整获取上一轮通信数据所产生的状态变化。模糊测试器获取到二元组数据后，首先根据地址对状态变量进行区分，然后根据选择的模式，决定最终保留状态变量的完整变化过程或是将状态变量的最终值作为单个状态节点。

在状态节点的定义上，SSFuzz 认为，由同一通信数据所造成的所有状态变化都应算作同一状态节点。以 AFLNET 为例，AFLNET 根据每轮收到的 Response 数据包提取状态节点。但是，当一个测试数据触发了多个 Response 数据包，且数据包表示的状态不同时，AFLNET 会将其作为多个状态节点保存。如图 3 所示，当 AFLNET 发送测试样例 M_1，SUT 反馈数据 $Response_A$ 与 $Response_B$，并可被解析为 S_A 与 S_B，此时，S_A 与 S_B 就会被加入状态机。在后续的种子调度过程中，对这两种状态分别进行测试，而不考虑是否可归因于同一状态，这导致模糊测试器会浪费双倍的时间在测试相同的状态节点上，严重影响了模糊测试的效率。针对这个问题，SSFuzz 在根据状态变量地址信息完成状态变量的区分后，按照一定规则将各个状态变量的数值组合为一个序列，计算该序列的 Hash 值用于表示状态节点。这样做的好处是，当其中任一状态变量发生变化时，SSFuzz 都能够实现敏锐的感知，并且，SSFuzz 还允许 AFLNET 所定义的状态信息与 SSFuzz 的状态信息相结合，只需要在每一轮处理状态信息时将 AFLNET 所搜集的状态加入该状态序列，并重新计算 Hash 即可。这种方式使得 SSFuzz 的状态机具有极强的精确性及扩展性。

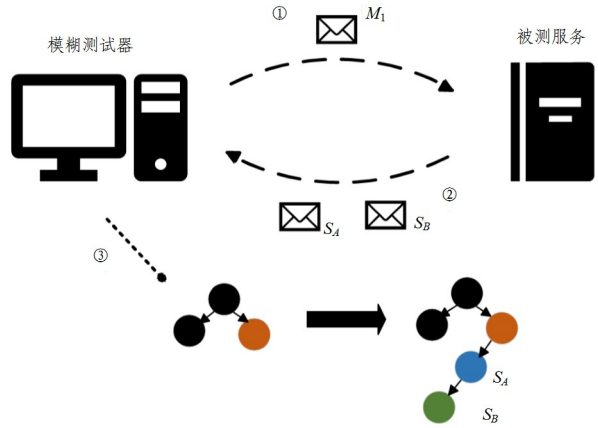

图 3 AFLNET 状态解析模型

Fig. 3 State resolution model of AFLNET

4 实验设计与评估

为了证明 SSFuzz 的有效性，我们从 Profuzzbench 内选取了部分具有代表性的协议。Profuzzbench[30] 是用于网络协议的有状态模糊。该基准测试包括一套具有代表性的用于流行协议的开源网络服务器，以及用于自动化实验的工具。在用于对比的模糊测试器上，选取最先进的模糊测试器 AFLNWE 和 AFLNET 作为比较对象。适用于网络协议的模糊测试工具还包括 SGFuzz，但是 SGFuzz 是基于 LibFuzzer 设计实现的，当触发程序异常后就会直接终止测试，因此，我们仅在回答状态识别效果时与其进行比较。

本文基于 AFLNET 实现了 SSFuzz 原型。为了证明 SSFuzz 的有效性，我们主要针对以下问题进行回答并分析结果产生的原因。

问题 1：SSFuzz 能否作用于多样化的协议实现，并实现状态变量赋值语句的插桩？

问题 2：SSFuzz 构建的状态机能否取得更好的模糊测试效果？

4.1 状态识别效果(问题 1)

程序状态本质上是一个相对宽泛的概念。与代码覆盖率这一明确的程序属性不同，程序状态是一个宏观概念。不同状态可能代表了程序的完整处理分支。也正因为如此，程序状态在缺少先验知识的情况下，难以确认程序状态节点总数，包括状态节点间的转换关系，因此难以直接应用覆盖率相关

概念对状态进行评估。此外，由于状态定义不同，因此也难以从同一维度去评价状态节点数是否会对模糊测试结果产生正反馈。

因此，在状态识别效果的实验评估上，我们通过对比不同状态识别方式所能支持的协议类型数量是否更满足多样性，来证明 SSFuzz 插桩策略的有效性。此外，应用相同理念的 SGFuzz 进行对比，证明 SSFuzz 所能支持的状态识别更加全面。

在这一部分，我们主要评估不同的状态信息收集方法是否适配于多种不同类型的协议代码。协议的选取上，我们在 ProfuzzBench 支持的项目中额外选取了部分 HTTP 协议，以丰富测试集。在对比实验的模糊测试工具上，选取 AFLNET 及 SGFuzz 两项最具代表性的基于状态的网络协议灰盒模糊测试工具作为对照组，结果如表 1 所列。

表 1 状态识别有效性实验

Table 1 Experimental results of status recognition validity

Subject	AFLNET	SGFuzz	SSFuzz
H2O	×	√	√
MbedTLS	×	√	√
Live555	√	√	√
OpenSSH	√	√	√
Exim	√	√	√
LightFTP	√	×	√

由表 1 可知，AFLNET 及 SGFuzz 均存在部分协议无法进行状态获取的情况。AFLNET 的状态捕获是基于服务响应数据包的内容提取实现的，并且，针对不同的协议需要定制化地实现相关的数据包解析方法。因此，对于开发人员尚未实现的协议，AFLNET 无法进行状态收集。SGFuzz 通过解析源代码中 Enum 类型数据的定义方式，解析所有 Enum 类型变量的直接赋值语句，实现插桩。但是，LightFTP 协议并没有基于 Enum 实现状态定义，而是使用宏定义实现。因此，SGFuzz 不能识别出其中的状态变量赋值代码。相对地，通过对此类语句的重定义，SSFuzz 能够实现状态变量赋值代码的插桩。

此外，SSFuzz 能够识别出 SGFuzz 不能捕获到的代码。SGFuzz 仅识别赋值语句右节点为常量的情况，忽视了函数赋值的方式。如图 4 所示，Live555 项目中，通过 setParseState 方法对状态变量进行更新，此时，SGFuzz 无法匹配出可动态变化的 parseState 变量，导致插桩失败；相比之下，SSFuzz 则能够通过 AST 解析实现精准识别。

```
Enum MPEGParseState {
  PARSING_VISUAL_OBJECT_SEQUENCE,
  PARSING_VISUAL_OBJECT_SEQUENCE_SEEN_CODE,
  PARSING_VISUAL_OBJECT,
  PARSING_VIDEO_OBJECT_LAYER,
  PARSING_GROUP_OF_VIDEO_OBJECT_PLANE,
  PARSING_VIDEO_OBJECT_PLANE,
  PARSING_VISUAL_OBJECT_SEQUENCE_END_CODE
};

void MPEG4VideoStreamParser::setParseState(MPEGParseState parseState){
   fCurrentParseState = parseState;
   MPEGVideo StreamParser::setParseState();
}
```

图 4 枚举变量间接赋值

Fig. 4 Indirect assignment of enumerated variables

以上实验数据表明，SSFuzz 在状态识别的效果上，与 AFLNET 相比有着更强的灵活性，减少了开发人员的分析工作；与 SGFuzz 相比，则有着更高的精度，能够结合源代码的语义信息，更准确地定位状态变量并插桩。

4.2 SSFuzz 状态机的有效性(问题 2)

为了验证 SSFuzz 的状态机的有效性，我们在 AFLNET 上实现并部署了相关策略，并在实验部分使用 AFLNET_SSFuzz 进行表示。在对照实验中，以 AFLNET 及 AFLNWE 两款先进的网络协议灰盒模糊测试工作作为对照，以相同测试样例总数的前提下模糊测试工具能够触发的代码路径数量作为指标，进行实验分析。以此作为指标的主要考虑是，模糊测试器的效率受多方面因素的影响，包括种子变异策略、吞吐量等因素。现有的网络协议模糊测试技术研究往往没有对其他变量进行很好的限制，导致实验过程中存在多种变量。最常见的，如比较相同时间内的代码覆盖率，这类实验很大程度上受吞吐量的影响，不足以证明在其之上构建的状态机能否提升模糊测试工具的工作效率。

为了避免实验过程中受到吞吐量的影响，我们比较相同的测试样例数量下所能达到的代码路径数量。这样能够有效避免由时间及吞吐量所带来的额外影响。

如图 5 所示，SSFuzz 能够在执行相同的测试样例数量下达到更多的路径分支，这表明仅改变状态机的构建技术，SSFuzz 所构建的自动机能够更有效地指导模糊测试器进行状态选择及种子选取。SSFuzz 的效果优于 AFLNET 的关键在于其减少了冗余的状态节点。AFLNET 中的状态选择算法是一种基于概率的随机调度算法。它为所有的状态节点均赋予了一定的权重，当模糊测试器选择某状态节点时，将触发该状态的测试样例作为变异种子。这意味着当出现冗余节点时，相同的测试样例被选作变异种子的概率更大。而 SSFuzz 将一次信息交互行为定义为一个状态节点，这使得 SSFuzz 不会出现多个状态共享同一测试样例的状况，使得状态选择更加合理。

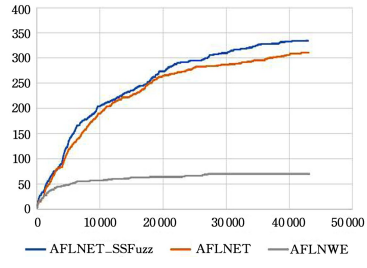

图 5 测试用例有效性(测试样例总数-路径发现数量)

Fig. 5 Test case effectiveness(total number of test samples-number of path findings)

同时，我们也在 ProfuzzBench 上部署了 AFLNET_SSFuzz，结果如图 6 所示。结果表明，AFLNET_SSFuzz 能够比 AFL 更快达到代码覆盖率的阈值。SSFuzz 是基于 AFLNET 实现的，因此，在状态选择算法、种子变异策略等其他组成部分不改变的前提下，存在能够探测到的代码覆盖率阈值。但是，从实验来看，SSFuzz 能够更快地达到该阈值，这同样是因为 SSFuzz 构建的状态机中冗余状态节点较少。并且，我们在

实验中发现，修改状态机构建策略后，SSFuzz 的吞吐量也得到了一定提升，说明精简测试样例复杂度后，SSFuzz 的状态机也能取得较好的效果。

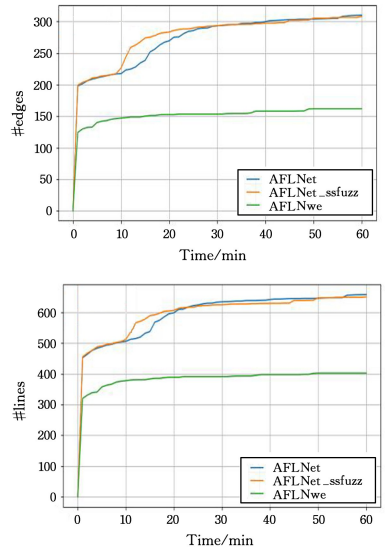

图 6 测试有效性(时间-路径发现数量)

Fig. 6 Test validity(time-number of paths fingdings)

以上数据说明，在忽略吞吐量、种子变异策略等因素的前提下，SSFuzz 设计实现的状态机能够有效提高待测程序的路径发现效率，减少冗余的测试过程。

结束语 本文的主要贡献是弥补了 SGFuzz 在状态收集上的不足，并对网络协议模糊测试状态机进行了定义，使得不同的状态收集策略之间可以共同构建状态机，能够有效推进模糊测试器的状态选择及种子筛选。我们在 AFLNET 上实现了 SSFuzz 原型工具，且通过实验证明了 SSFuzz 能够识别出函数间接赋值类型的状态转换语句并进行插桩。在状态机指导种子选取方面，SSFuzz 能够以更少的测试样例获得更高的路径覆盖率。

随着系统化的测试软件越来越多，对于以网络协议为代表的服务端应用程序的完整性安全检测的重要性也逐渐上升。从网络协议模糊测试特性来看，未来的研究包括以下几个方面：

1)数据交互吞吐量提升：网络协议模糊测试的吞吐量受限于其独特的通信方式，提高模糊测试器与测试程序之间的通信效率能够快速提高模糊测试过程间的路径发现效率。

2)提高测试样例生成的准确性：测试样例的准确性指符合测试程序格式规范的测试样例。网络通信数据相较于文件 I/O 交互，对通信数据有着更严格的格式要求，不符合格式规范的网络通信数据在进入待测程序实际业务之前就会被丢弃。因此，如何提高测试样例生成的准确性是有效提高模糊测试技术路径发现效率的重要技术之一。

网络协议模糊测试技术在近几年受到广泛关注，从过去的基于生成的黑盒模糊测试技术发展为目前基于变异的灰盒模糊测试技术。灰盒模糊测试在过去十几年里已经被证明能够高效挖掘程序漏洞，但是如何将此技术有效应用于以网络协议为代表的系统型服务程序还有着很大的研究空间。

参 考 文 献

[1] CHEN Y R, LAN T, VENKATARAMANI G. Exploring Effective Fuzzing Strategies to Analyze Communication Protocols [C]// Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation. 2019: 17-23.

[2] American fuzzy lop(afl) fuzzer[EB/OL]. http://lcamtuf.coredump.cx/ afl/technical details.txt.

[3] SCHUMILO S, CORNELIUS A, ALI A, et al. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types [C]// USENIX Security Symposium. 2021: 2597-2614.

[4] QIN S S, HU F, MA Z Y, et al. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(6): 1-26.

[5] ANDRONIDIS A, CADAR C. SnapFuzz: high-throughput fuzzing of network applications[C]// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022: 340-351.

[6] LI J Q, LI S Y, SUN G, et al. SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols Using Snapshots[J]. IEEE Transactions on Information Forensics and Security, 2022, 17: 2673-2687.

[7] VAN-THUAN P, BÖHME M, ROYCHOUDHURY A. AFLNET: A Greybox Fuzzer for Network Protocols [C] // 2020 IEEE 13th International Conference on Software Testing, Validation and Verification(ICST). 2020.

[8] BA J S, BÖHME M, MIRZAMOMEN Z, et al. Stateful Greybox Fuzzing. [J]. arXiv: 2204.02545, 2022.

[9] MARIA L P, MAX V H, BEN W, et al. Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents[C]// 2022 IEEE Symposium on Security and Privacy. 2022: 51-68.

[10] Boofuzz: A fork and successor of the sulley fuzzing framework [EB/OL]. https://github.com/jtpereyda/boofuzz.

[11] Peach Fuzzer Platform [EB/OL]. http://www.peachfuzzer.com/products/ peach-platform.

[12] Sulley: A pure-python fully automated and unattended fuzzing framework[EB/OL]. https://github.com/OpenRCE/sulley.

[13] SHE D D, KRISHNA R, YAN L, et al. MTFuzz: Fuzzing with a Multi-Task Neural Network[C]// Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 737-749.

[14] ZONG P Y, LV T, WANG D W, et al. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-Box Fuzzing through Deep Learning[C]// USENIX Security Symposium. 2020: 2255-2269.

[15] LIU S H, MAHAR S, RAY B, et al. PMFuzz: Test Case Generation for Persistent Memory Programs[C]// Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021: 487-502.

[16] LUO Z X, ZUO F L, SHEN Y H, et al. ICS Protocol Fuzzing:

Coverage Guided Packet Crack and Generation[C]// 2020 57th ACM/IEEE Design Automation Conference. 2020:1-6.
[17] ARAUJO R, LUIS G, DANIEL M B. Program-Aware Fuzzing for MQTT Applications[C]// Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020:582-586.
[18] ZUO F L, LUO Z X, YU J Z, et al. PAVFuzz: State-Sensitive Fuzz Testing of Protocols in Autonomous Vehicles[C]// 2021 58th ACM/IEEE Design Automation Conference. 2021: 823-828.
[19] FITERAU-BROSTEAN P, JONSSON B, MERGET R, et al. Analysis of DTLS Implementa-tions Using Protocol State Fuzzing[C]// USENIX Security Symposium. 2020:2523-2540.
[20] LI X Y, PAN X J, SUN Y B. PS-Fuzz: Efficient Graybox Firmware Fuzzing Based on Protocol State[J]. Journal on Artificial Intelligence, 2021(1):21-31.
[21] ROBERTO N. StateAFL: Greybox fuzzing for stateful network servers[J]. Empirical Software Engineering, 2021, 27:1-31.
[22] CANAN A, KARAKAYA U. SP-Fuzzy Soft Ideals in Semigroups[J]. Turkish Journal of Mathematics and Computer Science, 2018, 10:22-32.
[23] KHANDAIT P, HUBBALLI N, MAZUMDAR B. IoT Hunter: IoT network traffic classification using device specific keywords [J]. IET Networks, 2021, 10:59-75.
[24] ZHAO J J, CHEN S L, LIANG S R, et al. RFSM—Fuzzing a Smart Fuzzing Algorithm Based on Regression FSM[C]// 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing. 2013:380-386.
[25] PENG H, SHOSHITAISHVILI Y, PAYER M. T-Fuzz: Fuzzing by Program Transformation[C]// 2018 IEEE Symposium on Security and Privacy. 2018:697-710.
[26] KITAGAWA K, HANAOKA M, KONO K. AspFuzz: A state-aware protocol fuzzer based on application-layer protocols[C]// The IEEE Symposium on Computers and Communications. 2010:202-208.
[27] GORBUNOV S, ROSENBLOOM A. AutoFuzz: Auto-mated Network Protocol Fuzzing Framework[J]. International Journal of Computer Science and Network Secrity, 2010, 10(8): 239-245.
[28] HERRERA A, PAYER M, HOSKING A L. DataFLow: Toward a Data-flow-guided Fuzzer[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32:1-31.
[29] MANTOVANI A, FIORALDI A, BALZAROTTI D. Fuzzing with Data Dependency Information[C]// 2022 IEEE 7th European Symposium on Security and Privacy. 2022:286-302.
[30] NATELLA R, VAN-THUAN P. ProFuzzBench: a benchmark for stateful protocol fuzzing[C]// Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021:662-665.

LIN Jiahan, born in 2000, postgraduate. His main research interests include software automated testing and reverse engineering.

PENG Jianshan, born in 1979, Ph.D, associate professor, master supervisor. His main research interests include cyber security and software automated testing.

(责任编辑:何杨)