计算机科学
COMPUTER SCIENCE

基于静态分析驱动型的IOS-XE Web命令注入漏洞检测方法

鲁波, 吕晓

引用本文

鲁波, 吕晓. 基于静态分析驱动型的IOS-XE Web命令注入漏洞检测方法[J]. 计算机科学, 2025, 52(12): 419-427.

LU Bo, LYU Xiao. Detection of Web Command Injection Vulnerabilities on IOS-XE Based on Static Analysis-drivenApproach [J]. Computer Science, 2025, 52(12): 419-427.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.250100060

基于静态分析驱动型的 IOS-XE Web 命令注入漏洞检测方法

鲁 波 吕 晓
海军工程大学 武汉 430033
(280856780@qq.com)

摘 要 目前针对网络设备 webUI 的漏洞挖掘已经变得非常普遍,漏洞滥用带来严重威胁,网络设备的安全稳定成为安全领域关注的重点。模糊测试是针对网络设备 Web 接口漏洞挖掘的主流方法,但这些方法在 Cisco IOS-XE 系统上效果均不理想。为此,针对 IOS-XE 系统的 Web 框架提出了一种基于静态分析驱动型模糊测试框架 IOXFuzzer,用于检测底层命令注入漏洞。IOXFuzzer 通过对后端 Lua 脚本进行抽象语法树建模,构建危险路径库反向追溯危险路径,再构建参数树筛选高质量种子库,生成高覆盖率测试用例,增加了发现脆弱代码的概率。最后在 Cisco ASR 1000 系列、ISR 4000 系列实体设备和 CSR 1000v 系列虚拟设备上用 2019 年至今的 69 个不同版本固件对 IOXFuzzer 进行了评估,共检测出 8 个底层命令注入漏洞,其中 1 个为未公开漏洞。

关键词:Cisco;IOS-XE;静态分析;模糊测试;命令注入
中图分类号 TP393

Detection of Web Command Injection Vulnerabilities on IOS-XE Based on Static Analysis-driven Approach

LU Bo and LYU Xiao
Naval University of Engineering,Wuhan 430033,China

Abstract Vulnerability mining for the Web interface of network devices has become very common,and the abuse of vulnerabilities poses a serious threat,the security and stability of network devices catch the attention in the security field. Fuzzing is the main method for Web interface vulnerability mining of network devices,but these methods have little effect on the Cisco IOS-XE system. Therefore,a static analysis-driven fuzzing framework based on the IOS-XE webUI,called IOXFuzzer,is proposed to detect the underlying command injection vulnerabilities. IOXFuzzer increases the probability of discovering vulnerable code by modelling back-end Lua scripts with abstract syntax trees,constructing dangerous path libraries to trace dangerous paths backwards,constructing parameter trees to filter high-quality seed libraries,and generating high-coverage test cases. At the end,IOXFuzzer is evaluated on Cisco ASR 1000,ISR 4000 series physical devices,and CSR 1000v series devices with 69 different firmware versions from 2019 to present and detects a total of eight underlying command injection vulnerabilities,one of which is undisclosed.

Keywords Cisco,IOS-XE,Static analysis,Fuzzing,Command injection

1 引言

Cisco 是全球网络设备供应商之一,IOS-XE 系统广泛部署于 Cisco 路由器、交换机等平台,其设备应用于政府机构、电信、企事业单位等各个领域。随着互联网技术的发展和网络攻击威胁的不断演变和增强,Cisco Web 漏洞的滥用导致整个 Cisco 设备被攻击、数据被篡改,产生严重安全隐患[1-3]。Web Fuzzing 技术[4-7]是一种基于输入验证不充分、访问控制缺陷的自动化测试方法,它能够自动构造各种输入来模拟攻击,从而发现 Web 应用程序中存在的各种漏洞。

IOS-XEWeb 命令注入漏洞是指 Cisco IOS-XE 操作系统中的一种安全漏洞,它允许攻击者通过向系统 Web 界面输入恶意构造的命令,在设备上执行未授权的操作,如 CVE-2020-3211 和 CVE-2023-20273。由于系统未对用户输入进行严格的验证和过滤,因此攻击者可以通过发送特制的 HTTP 请求,在目标设备上执行任意命令。

IOS-XE 系统是闭源操作系统,其 Web 接口与传统的 IOT 设备有很大不同,其高复杂度、高可靠性和高定制化的特性使得传统的 Web Fuzzing 技术不再适用,且针对 Cisco IOS-XE 系统 Web 管理服务的模糊测试框架较少。模糊测试需要在大量不同版本的系统上进行,现实网络中使用 IOS-XE 系统的设备主要有 Cisco ASR 和 ISR 系列实体设备以及 CSR

到稿日期:2025-01-09 返修日期:2025-04-28
通信作者:吕晓(0909072004@nue.edu.cn)

1000v 虚拟设备，其中 ASR 系列设备主要用于运营商骨干网络，ISR 系列设备主要用于用户边界网络，CSR 1000v 虚拟设备主要用于 SD-WAN 网络，各自发挥着不可替代的作用。

为解决这些问题，在多年手工挖掘漏洞的基础上，本文提出一种针对 Cisco IOS-XE 设备 Web 漏洞的高效的、准确的、可靠的、自动化的挖掘方法，以发现和识别 Cisco Web 应用程序中潜在的安全漏洞，从而全面提升 Cisco 设备的安全性。本文主要研究以下内容：

1）基于静态分析的高质量种子生成方法

对后端代码进行抽象语法树[8]（Abstract Syntax Tree，AST）建模，构造系统调用路径图（Call Graph，CG），该图为静态分析的基础。采用汇聚点反向追踪策略，构造危险路径库，进而形成 URL 库。对提取的 URL 进行参数构造，结合污点传播思想，根据路径库构建参数树，通过 URL 与参数树生成高质量种子库。

2）针对注入型漏洞测试用例的高效生成方法

某 URL 对应的参数可能是庞大嵌套的树型结构，传统的穷尽式测试会大大增加时间成本，因此需要制定科学的变异策略来提高模糊测试效率。首先，采用控制变量法思想对参数逐一遍历；其次，通过静态分析后端代码提取分支判断语句中的常量（字符串、布尔量、整型等），并将其用于测试用例的生成；最后，基于词义推测构造参数。

本文的主要创新点如下：

1）引入静态分析方法生成高质量种子库，采用 ANTLR4 监听器模式对每个 Lua 脚本[9]构造 CG，提取危险路径，筛选高质量种子，有效解决隐藏 URL 的提取和模糊测试效率低的问题。

2）构建参数树提取多级嵌套参数，并基于参数名称进行词义推测和值填充，有效解决模糊测试覆盖率低的问题。

3）提出了静态分析驱动型模糊测试框架 IOXFuzzer，对基于 Lua 开发的 Web 应用进行安全分析，在 Cisco ASR 1000，ISR 4000 系列实体设备和 CSR 1000v 系列虚拟设备上用 2019 年至今的 69 个不同版本固件对 IOXFuzzer 进行了评估，共检测出 8 个底层命令注入漏洞，其中 1 个为未公开漏洞。

第 2 章分析了 IOS-XE Web 框架，介绍了国内外在 Web 模糊测试领域的研究现状；第 3 章提出了一种基于静态分析驱动型模糊测试方法，从语法树构建、路径库构建和参数树构建进行了详细论述；第 4 章从覆盖率、效率和发现漏洞的能力方面对 IOXFuzzer 进行了评估；最后总结全文，并指出了不足点和下一步的研究方向。

2 背景与相关工作

2.1 IOS-XE Web 框架

在分析 IOS-XE Web 框架时，首先需要了解其构建基础，具体来说，IOS-XE 使用 OpenResty 构建其 Web 服务。OpenResty 是一个基于 Nginx 与 Lua 的高性能 Web 平台，在实际运行中，来自外部的请求通常需要经过 Nginx 反向代理。内部通信则通过 192.168.1.5 和 192.168.1.6 这两个 IP 地址实现。以 GET 请求为例，假设 URL 为 http://*/webui/reset/snmp，其处理流程如下：

首先，该请求会被 Nginx 进程接收，Nginx 根据配置查询对应的处理模块 baseHandler.lua，该模块进一步将功能分发至 snmp.lua 模块。然后，snmp.lua 调用来自 WSMAApiLib.lua 的 API 函数封装成 XML 格式的负载。最终，该负载会被传递给 IOS-XE 系统中的核心进程（IOSD）中的 wsma 模块进行处理。图 1 展示了这一完整的处理流程。

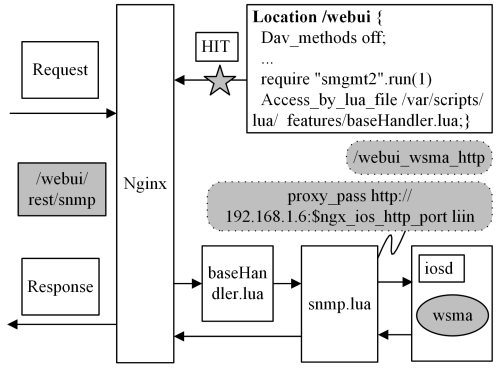

图 1 IOS-XE Web 框架

Fig. 1 Web framework of IOS-XE

2.2 研究现状

自从 Web 应用漏洞开始受到行业安全研究人士的关注以来，大量的针对 Web 漏洞挖掘的方法和工具诞生。目前模糊测试是针对网络设备 Web 接口有效的漏洞挖掘方法，其关键思想就是生成并向目标程序发送大量有可能触发软件错误的测试用例。模糊测试系统可依据对被测软件的信息掌握程度分为 3 类，黑盒、白盒以及灰盒模糊测试技术。

黑盒方法依赖输入输出反馈，代表性方法有 WMIFuzzer[10]等。它们不需要人工编写报文模板就能够生成符合标准要求的种子报文，但是不支持对带有状态性的网络协议的模糊测试。Yu 等[11]通过使用多级消息生成机制来完成对具有状态性网络协议的模糊测试，从而实现了 IoTHunter。Fuzz4All[12]通过大语言模型（LLM）增强语义理解能力，生成跨语言测试用例（C/C++/Java/Python），结合循环生成策略优化覆盖率，在 GCC 和 Clang 中发现 98 个错误（含 64 个未知漏洞）。MesFuzz[13]提出多种群自适应突变策略，通过种子聚类动态优化突变概率，在 LAVA-M 基准测试中路径覆盖率比 AFL++高 12%。

白盒方法通过静态分析或动态插桩获取程序内部信息以指导测试。静态测试不要求在计算机上实际执行所测程序，主要以一些人工的模拟技术对应用进行分析和测试，例如控制流分析、数据流分析、信息流分析等，代表性方法有 SAGE[14]和 KLEE[15]。OSmart[16]提出首个白盒选项感知模糊测试框架，通过静态分析提取 PHP Web 应用的未文档化配置选项依赖关系，结合动态反馈来生成有效的输入组合。DSM-Fuzz[17]针对 Java 反序列化漏洞，提出双向污点追踪与 TrustRank 算法结合的调用链特征提取方法，覆盖率和执行深度优于传统工具 30%以上。Sherin 等[18]提出了一种使用引导搜索的动态搜索方法 Qexplore，该方法系统地探索了动

态 Web 应用程序，只需较少或不需要关于 Web 应用程序的先验知识。

灰盒方法结合反馈机制与轻量级插桩，是主流研究方向，代表工具为 EWVHunter[19]，它在 Web 前端输入源上填充数据，然后重用程序处理逻辑来构建结构化的通信消息。Zhang 等[20]提出了一种面向物联网 Web 接口测试的状态消息生成机制 SMG，并实现了 SIoTFuzzer 框架，通过 Web 前端分析和状态分析来确定 Web 前端与发送报文数据的状态关系，再通过对报文数据的参数变异、报文数据的结构变异对嵌入式设备进行模糊测试。Gao 等[21]实现了一套模糊测试框架 IoT-Parser，从固件中提取页面路径以扩大爬虫范围，提取共享关键字用于加快漏洞发现的速度。Atropos[22]针对 PHP Web 应用提出快照恢复与动态反馈机制，自动推断键值结构并维护会话状态，在真实应用中发现了 7 个未知漏洞，代码覆盖率较 WebFuzz[23]和 WFuzz 分别提升 50% 和 230%。PTree-Fuzz[24]基于解析树建模 Java Web 输入数据，隔离数据块并优化种子权值分配，解决了结构化输入测试低效的问题，异常触发率提升 28%。LLAMAFUZZ[25]首次将 LLM 与灰盒测试结合，通过语义化输入生成和成对变异种子微调模型，在 Magma 基准测试中发现 47 个新漏洞，路径覆盖率提升 41%。

由于 IOS-XE 系统是闭源操作系统，无法获取更多内部信息，因此白盒测试不可行，而使用黑盒测试对 Web 进行遍历式模糊测试的效率不高，并且存在隐藏 URL 问题导致覆盖率得不到保障。He 等[26]提出了第一个针对 IOS-XE 系统 Web 管理服务的模糊测试框架 CRFuzzer，通过分析前端请求和后端程序来优化种子生成，以发现系统中存在的隐藏 API。它虽不能像白盒测试一样获取详细的执行过程和状态信息，但其相较于黑盒测试，通过分析后端脚本获取了更多有用的信息。CRFuzzer 虽然能挖掘一些漏洞，但仅考虑了至多两层嵌套关系的参数形式，而且针对性的 POST 请求形式单一，具有很大的局限性。

当前针对 Cisco IOS-XE 系统 Web 管理服务的公开模糊测试框架较少，现有的方法在 IOS-XE 系统上进行模糊测试时，速度和覆盖率都得不到保障，存在很大的局限性，无法有效检测 IOS-XE 系统 Web 漏洞。

3　方法设计

本文提出了一个针对 Cisco 设备 IOS-XE 系统 Web 管理服务的静态分析驱动型模糊测试框架 IOXFuzzer，如图 2 所示，主要包括静态分析、变异与模糊测试和异常监控 3 个模块。

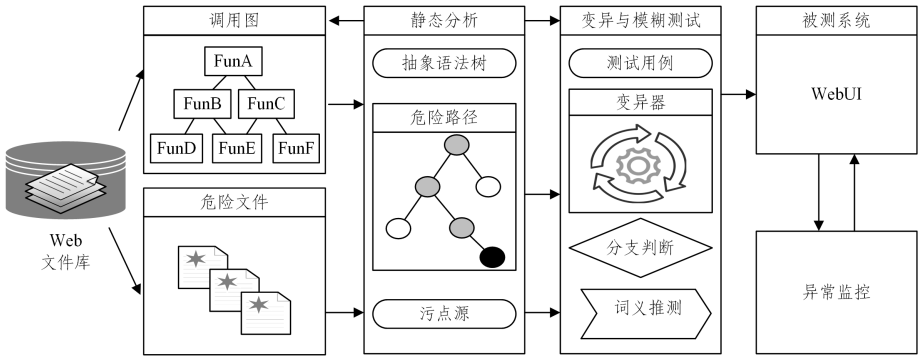

图 2　IOXFuzzer 系统架构

Fig. 2　System architecture of IOXFuzzer

1）静态分析模块。分析后端脚本 Lua 脚本，利用 ANTLR4 构建语法树，在此基础上通过反向追溯提取危险路径库，利用提取出的危险路径进行正向传播危险分析，构建参数树，生成初始种子。

2）变异与模糊测试模块。对初始种子进行变异，结合漏洞类型生成定制化字典，使用推测性填值生成测试用例并进行打标，再向设备发送 POST 请求进行模糊测试。

3）异常监控模块。待所有测试用例发送完毕，将获取的异常结果与打标的测试用例进行对比，进而得到能够触发漏洞的测试用例。

该系统使用静态分析驱动型模糊测试方法，静态分析可弥补模糊测试在有严格结构化协议情况下低效的缺陷，模糊测试可降低静态分析漏洞的复杂度。本文择二者之优点组成了该系统，此系统的静态分析解决的核心问题是 URL 的提取及参数的构造，而非传统的路径分析，模糊测试重点在于对静态提取的参数进行针对某漏洞模型的特定构造和变异。

3.1　静态分析

常用的动态模糊测试会随机或通过特定规律生成大量的种子，效率不高且覆盖率得不到保障。对于特定的漏洞而言，未筛选的种子大多是无效的，而且有些 URL 可能不会直接出现在前端页面中，而是通过后端逻辑动态生成或隐藏。

静态分析是整个 IOXFuzzer 的基础，具体为从后端 Lua 脚本语言中生成高质量测试用例，主要包括 3 个方面的内容：语法树、路径库和参数树构建。

3.1.1　语法树构建——ANTLR4

为有效地分析后端 Lua 脚本文件，这里引入了 ANTLR4。ANTLR4 是一种强大的解析器生成工具，用于处理语言和数据格式的定义，可以根据定义的语法文件自动生成词法分析器和语法分析器，使用生成的解析器来解析后端的 Lua 脚本，从而构建出抽象语法树。

语法树是源代码的一种抽象表示，展示了代码结构的层次关系，它不仅是静态分析的重要内容，还为后续的参数构造

和污点传播分析奠定了坚实的基础。通过对后端 Lua 脚本进行语法树建模，能够深入理解代码的逻辑和结构。如图 3 所示，基于面向对象的思想对 Web 后端 Lua 脚本进行了建模。

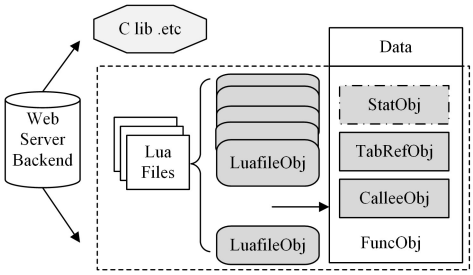

图 3 语法树建模

Fig. 3 AST modelling

建立语法树之单文件语法树构建，即将源代码解析成抽象语法树（AST），并生成对应的 LuafileObj 对象。利用 ANTLR4 监听器模式对每个 Lua 脚本进行分析，收集函数定义、调用关系和变量声明等信息，并将其转换为语法树的节点和边。

建立语法树之解决跨文件引用问题。首先通过解析 require 语句，建立模块别名到实际 Lua 脚本的映射关系，并将其存储为调用映射表。然后基于这些映射关系，遍历文件中的所有调用点以解决跨文件引用问题，并动态更新跨文件的调用关系。

建立语法树之调用路径图，用于直观展示程序中函数之间的调用关系，如算法 1 所示。通过反复迭代，直到调用图的结构在连续两轮中没有发生变化，则终止循环过程。

算法 1 语法树建立

输入：Lua 根目录

输出：语法树 asTree

```
1. init asTree / * 初始化语法树 * /
2. for luafile in luaPool do / * 遍历目录树 * /
3.     get_call_flows(luafile,asTree)
       / * 收集当前文件的调用流，更新语法树 asTree * /
4. end
   / * 建立调用映射表，记录模块导入关系和返回值的使用情况 asTree[filename]['req_mapping_table'] = {"var":['mod_name','return_table']} * /
5. build_mapping_table(asTree)
6. for fname in asTree.keys() / * 遍历语法树 * /
7.     do
8.         last_len = len(asTree[fname].call_flows) / * 记录当前调用流的长度 * /
9.         solving_cross_file_refs(fname,asTree) / * 解决跨文件引用问题，更新调用流 * /
10.        len = len(asTree[fname].call_flows) / * 更新当前调用流的长度 * /
11.    while(last_len != len) / * 当调用流长度不再变化时，结束迭代 * /
12. end
```

通过多轮迭代解决跨文件引用问题，直至调用路径图稳定后，算法 1 完成语法树的层次结构建模。该模型包括代码的嵌套结构和依赖关系，可实现 Lua 脚本向结构化数据的转换。在此基础上，进一步生成调用路径图，用于跟踪输入数据在代码中的流向。图 4 展示了某 Lua 脚本生成的调用路径图结果。

图 4 Lua 调用路径图

Fig. 4 Lua call graph

通过生成调用路径图，能够清晰地看到函数之间的调用链条和依赖关系。这不仅有助于实现代码覆盖分析，还能为输入数据流的追踪与分析提供支持。同时，基于调用路径图的信息，可以进一步优化模糊测试策略，从而提升模糊测试的覆盖率。

3.1.2 路径库构建——反向追溯危险路径提取

通过分析后端框架 OpenResty，可知 webui 处理的 URL 主要来源于 3 个方向：一是通过 urlMap.lua 定义 URL 及对应处理该 URL 请求的 Lua 脚本路径；二是在 Lua files 中也包含了一些 URL，这部分 URL 一般是 urlMap.lua 中 URL 的分支；三是 webui.conf 中也包含一部分 URL 及对应的处理文件。本文从上述文件中提取全部 URL 构建 URL 库。

在代码中，关键的汇聚点（Sink Point）对系统安全和稳定性有重要影响。这些汇聚点包括数据库操作、文件读写、网络通信以及命令执行等。通过识别系统中的关键汇聚点并反向追踪路径，可以高效地构建危险路径库，确保测试的针对性和有效性。再通过反向追踪策略，可以识别并追踪这些关键点的源头和传播路径。具体步骤如下：

1）汇聚点识别。结合 IOS-XE 系统的特性，本文主要挖掘底层 linux 注入漏洞，更关心命令执行相关的操作，即 os.system，subproceess.check_output，io.popen，os.execute，executeOSCommand，utils.executeOSCommand，pexec.safe_shell，utils.runPexecCommand，ngx.Pexec_setsid，ngx.Pexec 等。

2）基于生成的调用路径图，反向追溯危险路径，即从汇聚点开始，反向回溯其调用路径，追踪数据流的源头和中间传递过程，包括：

（1）依赖关系分析：分析汇聚点与其他代码段的依赖关系，识别可能影响汇聚点的变量和参数。

（2）多路径追踪：针对每个汇聚点，追踪多条可能的输入路径，确保全面覆盖所有潜在的影响路径。

从所构建的语法树和汇聚点中提取可能的危险路径，如算法 2 所示。

算法 2　提取危险路径

```
输入：语法树 asTree，汇聚点 sink_fun
输出：危险路径 hazard_urls
1. init hazard_urls
2. for fname in asTree.keys() do /*遍历语法树*/
3.     call_flows_dict=get_callflow(asTree[fname].call_flows) /*
       获取函数调用关系*/
4. G_callflow=callflow_to_digraph(call_flows_dict) /*得到 call
   graph*/
   /*找出所有的叶节点，即出度为 0 的点*/
5. leaves=[v for v,d in G_callflow.out_degree() if d==0]
6. for leaf in leaves do
7.     paths=nx.all_simple_paths(G_callflow,'main',leaf) /*得到
       main 到叶节点的路径*/
8.     if(path_check(sink_func,paths)) then /*检查路径中是否包含
       汇聚点*/
9.         url=path_to_url(paths,asTree) //将调用路径转换为 URL
10.        hazard_urls.append(url)
11.    end
12. end
```

3.1.3　参数树构建——正向传播污点分析

构建参数树的目的是为了提高生成测试用例的覆盖率。参数树是一个层次结构，展示了参数在路径中经过的每一个操作（如赋值、函数调用、传递等）。根据 3.1.2 节中所构建的危险路径库，对其构造参数树，基于 URL 中参数在后端传递和使用的特征，采用了污点分析的方法，以 POST 或 GET 的赋值变量作为污染源，跟踪其传播路径来进行污点分析。在此基础上，本文提出了参数的基本树和扩展树。其中，基本树仅包含严格的表引用和赋值，适用于简单的表结构分析，而扩展树则在基本树的基础上进一步包含了更松散的表引用。

1）基本树的构建

基本树的构建，通过分析 Lua 代码中的表引用和分支常量构建一个反映变量层次结构的树型结构，展示变量之间的引用关系。

在 Lua 中，表的引用通常采取点操作符“.”或者方括号“[]”的方式，分别表示访问某个字段或索引。例如，表引用语句（z=x.y）或（a=z.b）对应一个访问路径，揭示了表变量之间的层级关系，对这些引用进行递归和分析，构建出父子关系，为参数树的建立提供原始的数据结构。

然而，表变量的取值不仅仅受到语句本身的影响，还可能会受到程序中分支条件的制约，条件语句（如“if”）常用于控制表的取值，这些分支条件不仅为树的节点提供了额外信息，即描述了当前节点的取值范围，还可能影响后续节点的解析过程。

2）扩展树的构建

扩展树的构建是在基本树的基础上进一步扩展树型结构的过程，尤其是在处理表作为函数参数传递和带有 wrapper 的引用等更为松散的情况，例如（f(x.y)）或字符串键（x.y["key"]）访问表的情况。这些引用方式在基本树中可能无法完全捕捉，但在扩展树中可以通过动态分析将其纳入树型结构中。通过遍历函数中的所有表引用语句，扩展树将这些较为松散的表引用动态地关联到基本树的对应节点上，并将其整合纳入树形结构中，从而构建出一个覆盖更全面、能够表征如 f(x.y)和 x.y["key"]等松散引用情况的扩展参数树。

3）参数树的形成

表的引用路径可能会跨越多个函数调用，结合 3.1.1 节中生成的调用路径图，将表的引用路径与目标函数的参数树进行关联，促进树的生成。例如，如果表（x.y）作为参数传递给函数 Func_A，而 Func_A 又将其传递给 Func_B，则表的引用路径（x.y）被扩展到 Func_B 的参数树中，最后遍历生成整个参数树。

对 configSerialInterface.lua 文件构造的参数树部分结果如图 5 所示，其中包括了多层嵌套关系。

```
1. Parsing ['/webui/rest/configSerialInterface']
2. Node('/params')
3.     Node('/params/interfaceConfigArray')
4.         Node('/params/interfaceConfigArray/LLLL')
5.             Node('/params/interfaceConfigArray/LLLL/admin')
6.             Node('/params/interfaceConfigArray/LLLL/ipv4')
7.                 Node ('/params/interfaceConfigArray/LLLL/ipv4/
                   ipv4Address')
8.             Node('/params/interfaceConfigArray/LLLL/ipv4/Nat')
9.             Node('/params/interfaceConfigArray/LLLL/ipv6')
10.                Node ('/params/interfaceConfigArray/LLLL/ipv6/
                   listIpb6Address')
11.                Node ('/params/interfaceConfigArray/LLLL/ipv6/
                   listIpb6Address/LLLL')
12.                Node ("/params/interfaceConfigArray/LLLL/ipv6/
                   listIpb6Address/LLLL/'ipv6Type'")
13.            Node('/params/interfaceConfigArray/LLLL/wanPrimaryOr-
               Backup')
14.        Node('/params/interfaceConfigArray/i')
15.            Node('/params/interfaceConfigArray/i/ipv4')
16.        Node('/params/platform')
```

图 5　参数树

Fig. 5　ParaTree

3.2　变异策略

优秀的变异策略能有效提高代码覆盖率，从而增加发现漏洞的可能性。常见的变异策略有字节翻转、边界值分析、格式变异、随机变异等，针对 Cisco IOS-XE 这样复杂的系统，变异策略应考虑到系统的特殊性以及 OpenResty 架构的特点。

某些 URL 的参数可能呈现为庞大嵌套的树型结构，穷尽式测试会大幅增加时间成本，因此需要制定科学的变异策略，提高模糊测试效率。

1）采用控制变量法思想对参数进行逐一遍历。通过隔离目标变量并对其进行变异处理，可以清晰地观察其对系统的影响。

2）对后端代码进行静态分析，提取分支判断语句中的常量（如字符串、布尔值、整型等），并将这些常量用于测试用例的生成。由于表引用常量判断分支会影响代码执行覆盖率，传统模糊测试中常量难以被覆盖。例如，在执行 copy 操作时，源路径可能为固定常量（如 bootflash/tftp/scp 等），或布尔值（如 isEnable=true），提取这些参数常量进行针对性填

充，能够有效提高代码执行覆盖率。

3)引入基于词义推测的参数值构造方法。通过分析参数名称和上下文信息，可以推导出合理的测试数据。例如，参数名字含有“IP”字样可推测其值为 x. x. x. x 形式；而含有“password”“username”“config”等词根的参数，则可以对应推测其具体值。这种基于词义推测的构造方法能够生成语义上合理的输入数据，特别适用于业务逻辑验证，并有助于发现特定场景下的漏洞。

IOXFuzzer 针对 Cisco IOS-XE 底层注入漏洞进行测试时，通过将“;”“&”等与用户输入参数连接来构造 shell 命令，触发命令注入。为了便于后续监控，在变异规则上采用了创建文件和重启系统两种方式。其中，创建文件是指在 IOS-XE 系统 bootflash 目录下创建文件，其文件存储设备 bootflash 对应底层 linux 目录/bootflash。

在生成测试用例时，为验证注入是否成功，可以在变异关键字中加入在 bootflash 目录下创建文件的命令，同时需避免使用空格并正解处理引号和转义等特殊字符，以保证命令执行成功。部分变异示例如下：

1) ‘‘id>/bootflash/CCC’

2) ‘&id>/bootflash/CCC’

3) ‘$(id>/bootflash/CCC)’

4) ‘ftp://$(id>/bootflash/CCC)’

虽然 IOS-XE 会对参数做安全处理，但某些情况下仍然会完全绕过完全机制，因此添加了重启系统的规则以应对无法写文件的场景。

此外，为了提升模糊测试速度，在对种子进行变异处理时，可对最终生成的测试用例进行打标处理，以便后续进行异常监控。

图 6 即为采用了控制变量的思想，通过隔离变量并对其中一个变量进行变异处理后打标生成的测试用例(测试用例被触发后生成不一样的文件)。在正常进行模糊测试中，通常需要与被测系统进行交互以确认当前测试用例是否通过，而采用打标方式只需待所有测试用例发送完毕后再获取异常结果，即可判断出通过的测试用例。这种方法减少了交互次数，提高了整体测试效率。

```
/webui/rest/saveFile{“feature”:“id>/bootflash/CCC15”,…}
/webui/rest/saveFile{“feature”:“‘id>/bootflash/CCC16’”,…}
/webui/rest/saveFile{“feature”:“$(id>/bootflash/CCC17)”,…}
/webui/rest/saveFile{“feature”:ftp://$(id>/bootflash/CCC18)”,…}
/webui/rest/saveFile{“feature”:“\\$(id>/bootflash/CCC19)”,…}
```

图 6 测试用例

Fig. 6 Test cases

3.3 异常监控

由于 IOXFuzzer 在生成测试用例时，加入了在 bootflash 中创建文件和重启系统两种规则，因此，为了验证测试用例是否触发脆弱代码，命令是否注入成功，采取以下方式进行监控。

1)监控文件生成

针对创建文件的规则，本文利用后端 Lua 脚本执行命令的功能获取 bootflash 存储内容。具体方法是向‘/webui/rest/execCliCommand’构造查看 bootflash 内容的负载并获取返回结果，检查返回内容中是否含有所创建的文件名即可。

CRFuzzer 也是通过类似的机制来检测漏洞，其优点在于实时性较高，缺点是需要频繁与设备交互，效率不高。然而，IOXFuzzer 在对种子进行变异生成测试用例时进行了标记。待所有测试用例发送完毕后，仅需获取一次 bootflash 内容即可判断出命中的测试用例，这样将速度提高了 1 倍。

2)监控重启

针对重启系统的规则，通过以下方式实现：首先发送 POST 请求并观察响应包状态码，以判断测试用例对系统的影响；当发生异常时，记录对应的测试用例；同时，检测系统重启状态，当 Web 服务恢复后继续发送后续的测试用例。

4 实验评估

IOXFuzzer 全部使用 Python 实现，并采用了开源模块 ANTLR4 进行开发。

实验所用设备包括 Cisco ASR 1000 系列、ISR 4000 系列实体设备和 CSR 1000v 虚拟设备。软件版本覆盖 16. 3. x—17. 9. x，共计 69 个固件版本，对 IOXFuzzer 进行了全面测试以确保其稳定运行。

在提高 IOXFuzzer 适应性方面主要考虑了以下两点：

1)支持不同类型的 POST 请求，包括 JSON 格式和表单格式的数据类型；

2)兼顾纯文本负载和经过 base64 编码数据的情况，以应对不同版本之间的差异。

为了全面评估 IOXFuzzer，本文从以下几个角度进行了分析：

1)从提取的 URL 数量和 URL 对应的 POST 请求参数数量来评估覆盖率；

2)从漏洞检测效率以及脆弱代码发现能力来评估其漏洞发现能力。

实验中将 IOXFuzzer 与现有模糊测试工具 W13scan 和 CRFuzzer 进行对比。W13scan 是一款开源的 Web 漏洞发现工具，支持主被动扫描和自定义插件，它本身不支持页面爬取功能，本文结合 crawlergo 使其具备爬虫能力，通过自定义插件为其添加对 Lua 脚本的支持，开发出 W13scan+。而 CRFuzzer 是首个针对 IOS-XE 系统 Web 管理服务的模糊测试框架，采用静态分析的方法提取前后端信息来优化种子生成并加速漏洞发现过程。

4.1 算法性能分析

1)时间复杂度

对整个算法的时间复杂度进行了详细分析，具体分解为以下 3 个主要阶段。

(1)语法树建立阶段：该阶段的主要任务是将 Lua 脚本转换为抽象语法树，采用 Antrl4 来解析实现，其计算复杂度与脚本规模呈线性关系，若输入包含 N 个 Lua 脚本，则该阶段的时间复杂度为 $O(N)$。

(2)危险路径库与参数树的构建阶段：IOXFuzzer 采用反向污点传播的思想，假设通过污点函数(汇聚点)过滤出 URL 的数量 U，参数树平均深度为 D，平均节点数为 B，则该阶段

的时间复杂度约为 $O(U*D*B)$。

(3)变异与模糊测试阶段：对 URL 参数进行组合测试，每个 URL 包含 K 个可变参数，每个参数具有 M 种变异方式，由于采用笛卡尔积生成所有可能的参数组合，且各参数选择相互独立，总组合数是各参数选择的乘积，呈指数关系，因此该阶段的时间复杂度为 $O(U*M^K)$。

2)空间复杂度

(1)语法树存储：假如有 F 个函数，每个函数的调用关系最多有 F 个调用，则处理 N 个 Lua 脚本的空间复杂度为 $O(N*F^2)$。

(2)参数树存储：假设参数树平均深度为 D，每个节点最多含有 C 个子节点，则存储这个参数结构所需空间为 $1+C+C^2+\cdots+C^D$。由于算法仅处理了包含污点函数的 URL 路径，则该阶段的空间复杂度为 $O(U*C^D)$。

(3)测试用例存储：在生成测试用例时，IOXFuzzer 以空间换速度，即将测试用例打标并存储至文件(见图 6)，而不是采用发送请求－等待结果这种方式，最后用 bootflash 目录下生成的文件与存储的测试用例进行对比，找出能触发漏洞的测试用例。空间复杂度随测试用例 T 增加而增加，则处理所有过滤出的 URL 的空间复杂度为 $O(U*T)$。

经分析，算法的性能是高度依赖输入规模的，并且在处理大型或复杂的 Lua 脚本时可能会变得非常缓慢。但通过限制污点函数可减少 URL 数量，通过限制变异方式可以减少组合爆炸的问题，因此最后能显著降低时间和空间复杂度，提升算法整体性能。

4.2　URL 数量

为了测试 IOXFuzzer 生成的初始种子覆盖率，本实验选取了 10 个具有代表性的固件版本(包括 16.3.8，16.6.4，16.8.1，16.9.5，16.12.3，17.1.1，17.2.1r，17.3.2，17.3.5 和 17.9.3a)，对提取的 URL 数量进行了对比分析，结果如图 7 所示。

对于提取的 URL 数量，CRFuzzer 采用了从前端请求和后端 Lua 程序提取 API 信息的方式，能够获取大部分的 URL，但未考虑 webui.conf 中部分 URL 对应的处理文件。而 W13scan＋采用从前端页面爬取的方式获取 URL，爬取的数量少于前两者，反映了隐藏 URL 的存在。同时从数据中可以看出，随着版本的升级，URL 的数量呈明显上升趋势。

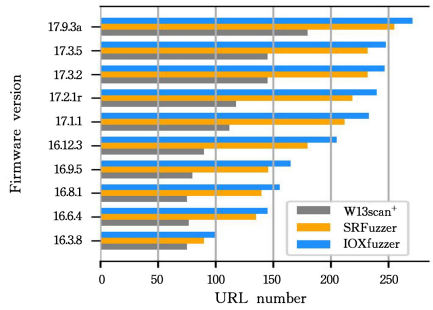

图 7　URL 数量

Fig. 7　URL number

4.3　POST 请求参数

为了测试 IOXFuzzer 生成的测试用例的覆盖率，本文选取了四个固件版本(16.3.8，16.9.5，17.1.1 和 17.9.3a)中的 8 个 URL(如 configController，policy，logs 等)，并对其 POST 请求参数提取结果进行了对比实验。在提取请求参数时，IOXFuzzer 通过构建语法树、参数树的方式从 Lua 脚本中提取参数，提取的参数量显著高于另外两种，结果如表 1 所列。

从实验结果可以看出，由于 W13scan＋是通过前端页面爬取的方式识别页面元素的，无法获取包括隐藏 URL 在内的所有页面，无法解决后端业务处理上的一些复杂情况。对于某一 URL，由于 CRFuzzer 是从前端请求识别具体参数的，仅考虑了两层以内嵌套的参数，对于仅有两层嵌套参数的页面 configStaticRoute，configSystemGeneral 和 updateInterfaceDetails，W13scan＋和 CRFuzzer 都能提取出所有参数；而对于 configSerialInterface 和 license 这些包含复杂处理逻辑的页面，它们提取的数量就会少很多。IOXFuzzer 从后端页面构建的参数树中提取的参数相比 W13scan＋和 CRFuzzer 会多很多，从而可以生成更多数量的种子，大大提高了发现脆弱代码的概率。

表 1　POST 请求参数提取

Table 1　Extract POST parameters

URL	W13scan＋				CRFuzzer				IOXFuzzer			
	16.3.8	16.9.5	17.1.1	17.9.3a	16.3.8	16.9.5	17.1.1	17.9.3a	16.3.8	16.9.5	17.1.1	17.9.3a
configController	10	8	8	8	11	15	15	15	15	21	21	21
configSerialInterface	2	2	2	2	7	1	1	1	16	52	52	97
configStaticRoute	15	16	19	14	15	16	19	14	15	16	19	14
configSystemGeneral	13	43	53	56	13	43	53	56	13	43	53	56
http	5	5	6	10	1	1	1	10	11	12	13	24
license	2	11	15	16	3	37	42	40	13	68	87	85
logs	2	4	5	5	6	4	5	5	6	8	14	11
updateInterfaceDetails	18	18	18	18	18	18	18	18	18	18	18	18
Total	67	107	126	129	74	135	154	159	107	238	277	326

4.4　漏洞检测效率

为了测试 IOXFuzzer 的漏洞检测效率，本实验在性能相对较好的实体设备 ASR1000 上对比了 W13scan＋，CRFuzzer 和 IOXFuzzer 在 16.3.8，16.6.4，16.8.1，16.9.5，16.12.3，17.1.1，17.2.1r，17.3.2，17.3.5，17.9.3a 这 10 个版本固件上测试时所使用的时间，结果如图 8 所示。

从图 8 中可以看出：W13scan＋使用从前端爬取的方式进行模糊测试，耗时远大于后两者；IOXFuzzer 对生成的测试用例进行了打标处理，在模糊测试阶段大幅减少了与设备交互的次数，耗时短于 CRFuzzer；同时 IOXFuzzer 通过汇聚点反向追踪策略，减少了初始种子的数量，提高了漏洞检测的效率。

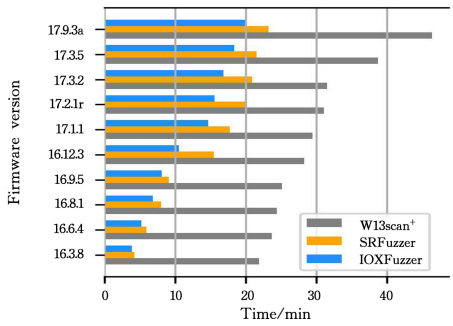

图 8 测试时间对比

Fig. 8 Comparison of testing time

4.5 脆弱代码发现

IOXFuzzer 使用静态分析的方法从后端 Lua 脚本进行语法树建模，通过分析参数树，增加条件判断，从而能覆盖到更多的分支，且能包含更多的嵌套和参数数量。但实际进行模糊测试时发现，有些功能如 VPN、容器配置等，需要进行复杂的配置，需要发送多个 POST 请求进行漏洞触发，或者需要将变异代码植入插件，再上传插件进行漏洞触发等，这些都无法通过单一的 POST 注入来发现脆弱代码。

实验选取了 69 个版本固件进行基于后端 Lua 脚本文件的语法树建模，生成高质量的种子来发现脆弱代码。实验中提取了包含危险路径的 Lua 脚本生成打标的测试用例，选取测试方法 W13scan+，CRFuzzer 进行对比，其中 W13scan+ 加入了对 Lua 脚本命令注入的支持，对 IOS-XE 系统进行 POST 注入漏洞测试，实验结果如表 2 所列。IOXFuzzer 共检测出 8 个底层命令注入漏洞，其中 1 个为未公开漏洞。

表 2 漏洞检测效果对比

Table 2 Comparison of vulnerability detection

ID	W13scan+			CRFuzzer			IOXFuzzer		
	ISR 4000	CSR 1000v	ASR 1000	ISR 4000	CSR 1000v	ASR 1000	ISR 4000	CSR 1000v	ASR 1000
CVE-2019-1862	1	1	1	1	1	1	1	1	1
CVE-2019-12650	1	1	1	1	1	1	1	1	1
CVE-2020-3211	1	1	1	1	1	1	1	1	1
CVE-2021-1435	0	0	0	1	1	1	1	1	1
CVE-2022-20693	0	0	0	0	0	0	1	1	1
CVE-2022-20851	0	0	0	1	1	1	1	1	1
CVE-2023-20273	0	0	0	0	0	0	1	1	1
unknown	0	0	0	0	0	0	1	1	1
Total	3	3	3	5	5	5	8	8	8

IOXFuzzer 基于后端 Lua 脚本提取所有参数进行测试，而不依赖上层配置或功能，而 W13scan+仅从前端请求中分析提取 API 信息，CRFuzzer 只提取了两层嵌套关系的参数，都无法生成更完整的测试用例，覆盖更多的程序执行路径。从实验过程分析，同一版本 IOS-XE 系统 ISR4000、CSR1000v 和 ASR1000 底层 Lua 脚本基本一致，IOXFuzzer 分析出的漏洞一样。实验结果表明，IOXFuzzer 可以有效地检测底层命令注入漏洞。

5 讨论和局限性

1）应用场景

相较于物联网设备，Cisco 设备主要用于企业、院校等大型网络环境，而传统检测工具难以满足其需求，IOXFuzzer 能够为网络安全管理员提供有效的漏洞检测辅助功能。

2）局限性

IOXFuzzer 的设计目标是针对 Cisco 设备 IOS-XE 系统进行模糊测试，在实际应用中存在一定限制：(1) IOXFuzzer 目前仅关注可能引发严重威胁的命令注入漏洞，对其他类型的安全问题尚未涉及；(2) 该方法采用语法树建模的方式进行静态分析，目前只针对 Lua 后端脚本设计，难以进行简单的扩展以适用于基他编程语言的程序分析；(3) 与常用模糊测试工具相比，IOXFuzzer 的优势在于它直接对后端 Lua 脚本进行处理，而不是通过爬取页面输出来发现漏洞，因此能够获得更高的测试覆盖率。然而，在应对涉及复杂逻辑场景下的漏洞挖掘时，其效果有限。

3）使用环境

由于 IOXFuzzer 采用静态分析框架，目前仅支持本地部署，适用于实验室环境或离线分析任务。对于需要实时检测的场景，可能无法满足需求，未来可结合动态模糊测试或其他技术手段以扩展其应用范围。

结束语 本文提出了一种覆盖率高的静态分析驱动型模糊测试框架 IOXFuzzer，用于检测命令注入漏洞。通过对后端脚本进行语法树建模能发现隐藏 API，从静态分析的角度提取了更多的 URL 和更完整的参数，大幅提升了模糊测试的覆盖率，同时采用汇聚点反向追踪和种子打标的方法提高了模糊测试效率。实验结果表明，IOXFuzzer 在 IOS-XE 系统有更强的漏洞检测能力。

在下一步工作中，将从以下两个方面开展研究：

1）复杂配置场景下的自动处理：开发能够自动识别复杂配置的算法，并基于此生成相应的 POST 请求链。这不仅能够提升系统的智能化水平，还为实现更高效的人工智能辅助模糊测试奠定了基础。

2）动态插桩技术的应用：通过引入动态插桩方法，进一步增强 IOXFuzzer 的覆盖率和漏洞检测能力，使其在复杂场景下的表现更加出色。

此外，考虑到部分防火墙设备（如 CheckPoint，F5 和 Fortiweb 等）的后端同样使用 Lua 脚本，未来将针对这些设备的特点进行深入分析，探索其与 Cisco IOS-XE 系统之间的差异性，并在此基础上扩展 IOXFuzzer 的通用性和可移植性，为后续开发更普适化的模糊测试框架提供重要参考。

参 考 文 献

[1] MUNIZ S. Killing the myth of Cisco IOS rootkits [EB/OL]. (2008-05-01) [2025-01-05]. https://drwho. virtadpt. net/images/killing_the_myth_of_cisco_ios_rootkits. pdf.

[2] LI F, ZHANG L, CHEN D. Vulnerability mining of Cisco router based on fuzzing [C]// The 2014 2nd International Conference on Systems and Informatics. 2014: 649-653.

[3] ZHOU J X, FENG D, LI B. A fuzzing method based on dual variation strategy for Cisco IOS [C]// 2017 3rd IEEE International Conference on Computer and Communications (ICCC). 2017: 205-209.

[4] LI J, ZHAO B D, ZHANG C. Fuzzing: a survey [J]. Cybersecurity, 2018, 1(1): 6.

[5] MANES V J M, HAN H S, HAN C, et al. The Art, Science, and Engineering of Fuzzing: A Survey [J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.

[6] COSTIN A, ZARRAS A, FRANCILLON A. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces [C]// Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. 2015: 437-448.

[7] XU W, WU Z H, WANG Z M, et al. Protocol Fuzzing Based on Testcases Automated Generation [J]. Computer Science, 2023, 50(12): 58-65.

[8] GU S K, CHEN W. Function Level Code Vulnerability Detection Method of Graph Neural Network Based on Extended AST [J]. Computer Science, 2023, 50(6): 283-290.

[9] COSTIN A. lua code: security overview and practical approaches to static analysis [C]// 2017 IEEE Security and Privacy Workshops (SPW). IEEE, 2017: 132-142.

[10] WANG D, ZHANG X, CHEN T, et al. Discovering Vulnerabilities in COTS IoT Devices through Blackbox FuzzingWeb Management Interface [J/OL]. https://doi. org/10. 1155/2019/5076324.

[11] YU B, WANG P F, YUE T, et al. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation [C]// Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 2525-2527.

[12] XIA C S, PALTENGHI M, TIAN J L, et al. Fuzz4All: Universal Fuzzing with Large Language Models [C]// 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE). 2024: 1547-1559.

[13] JIAO W H, LI X L, LI Q B, et al. Adaptive mutation based on multi-population evolution strategy for greybox fuzzing [J]. Information Sciences, 2025, 705: 121959.

[14] GODEFROID P, LEVIN M Y, MOLNAR D. SAGE: Whitebox Fuzzing for Security Testing [J]. Queue, 2012, 10(3): 20-27.

[15] CADAR C, DUNBAR D, ENGLER D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs [C]// Usenix Conference on Operating Systems Design & Implementation. USENIX Association, 2008: 209-224.

[16] WANG K L, CHEN M D, HE L, et al. OSmart: Whitebox Program Option Fuzzing [C]// Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. 2024: 705-719.

[17] WANG J, ZHANG B, ZHANG Z J, et al. Java Deserialization Vulnerability Mining Based on Fuzzing [J]. Netinfo Security, 2025, 25(1): 1-12.

[18] SHERIN S, MUQEET A, KHAN M U, et al. Qexplore: An exploration strategy for dynamicWeb applications using guided search [J]. Journal of Systems and Software, 2023, 195: 111512.

[19] WANG E Z, WANG B, XIE W, et al. EWVHunter: Grey-Box Fuzzing with Knowledge Guide on EmbeddedWeb Front-Ends [J]. Applied Sciences, 2020, 10(11): 4015.

[20] ZHANG H, LU K, ZHOU X, et al. SIoTFuzzer: Fuzzing Web Interface in IoT Firmware via Stateful Message Generation [J]. Applied Sciences, 2021, 11(7): 3120.

[21] GAO Y F, ZHOU X, XIE W, et al. Optimizing IoTWeb Fuzzing by Firmware Infomation Mining. Applied Sciences [J]. Applied Sciences, 2022, 12(13): 6429.

[22] GULER E, SCHUMILO S, SCHLOEGEL M, et al. Atropos: Effective Fuzzing ofWeb Applications for Server-Side Vulnerabilities [C]// Proceedings of the 33rd USENIX Security Symposium. Boston: USENIX Association, 2024: 4765-4782.

[23] ROOIJ O V, CHARALAMBOUS M A, KAIZER D, et al. WebFuzz: Grey-Box Fuzzing for Web Applications [C]// European Symposium on Research in Computer Security. 2021.

[24] WANG J, ZHANG Z J, YANG H Y, et al. Gray-box Fuzzing for JavaWeb with Parse Tree [J]. Computer Systems & Applications, 2023, 32(9): 67-76.

[25] ZHANG H X, RONG Y Y, HE Y F, et al. LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing [J]. arXiv: 2406.07714, 2024.

[26] HE J, CAI R J, YIN X K, et al. Detection ofWeb Command Injection Vulnerability for Cisco IOS-XE [J]. Computer Science, 2023, 50(4): 343-350.

LU Bo, born in 1985, engineer. His main research interests include computer confidentiality and network security.

LYU Xiao, born in 1983, Ph. D, professor, is a member of CCF (No. 61813M). Her main research interests include collaborative computing and computer network security.

（责任编辑：柯颖）