一种针对网络设备的已知漏洞定位方法

王 琛[1,2,3,4], 邹燕燕[1,2,3,4], 刘龙权[1,2,3,4], 彭 跃[1,2,3,4], 张 禹[1,2,3,4],
卢昊良[1,2,3,4], 王鹏举[1,2,3,4], 郭 涛[1], 霍 玮[1,2,3,4]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 中国科学院网络测评技术重点实验室 北京 中国 100195
[3] 网络安全防护技术北京市重点实验室 北京 中国 100195
[4] 中国科学院大学网络空间安全学院 北京 中国 100049

摘要 骨干级网络设备作为关键基础设施, 一直是网络攻防中的焦点, 与此同时, 其作为一个封闭、复杂的信息系统, 漏洞的公开研究资料相对较少、漏洞细节缺失较多。补丁对比是一种有效的漏洞分析手段, 而骨干级网络设备固件解包后通常具有单体式可执行文件, 这类文件具有函数数量多、文件规模大、调试符号信息缺失等特点, 直接进行补丁比对会产生大量待确认的误报差异, 同时启发式算法可能将两个不相关的函数错误匹配, 导致正确的安全修补缺失及漏报。传统的补丁比对方法无法有效地解决这类文件的补丁分析问题, 漏洞细节的分析遇到挑战。本文提出了一种针对单体式可执行文件中已知漏洞的定位方法 MDiff, 通过漏洞公告描述中的子系统概念与目标二进制文件的内部模块结构对目标进行了拆分, 在基于局部性的二进制比对技术之上, 利用语义相似度衡量方法对比对结果进行筛选排序。具体来讲, MDiff 首先利用入口函数及局部性原理识别存在漏洞的网络协议服务代码, 即粗粒度定位阶段。其次针对已识别出的、存在漏洞的网络协议服务代码模块中存在差异的函数进行动静态结合的语义信息分析, 包括基于扩展局部轨迹的安全修补识别, 基于代码度量的安全修补排序等步骤, 即细粒度定位阶段。基于该两阶段漏洞定位方法, 我们实现了一个原型系统, 对 4 个厂商设备中已经披露的 15 个漏洞进行实验。实验结果表明, 本文提出的漏洞定位方法可以提高网络设备的补丁分析效率, 支持研究人员发现已知漏洞细节。

关键词 网络设备; 模块划分; 补丁比对
中图法分类号 TP309.1 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2023.11.05

Locating 1-Day Vulnerabilities in Network Equipment

WANG Chen[1,2,3,4], ZOU Yanyan[1,2,3,4], LIU Longquan[1,2,3,4], PENG Yue[1,2,3,4], ZHANG Yu[1,2,3,4],
LU Haoliang[1,2,3,4], WANG Pengju[1,2,3,4], GUO Tao[1], HUO Wei[1,2,3,4]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing 100195, China
[3] Beijing Key Laboratory of Network Security and Protection Technology, Beijing 100195, China
[4] University of Chinese Academy of Sciences, Beijing 100049, China

Abstract Backbone network equipment, a key infrastructure, has always been the focus of network attack and defense. At the same time, as a closed and complex information system, there are relatively few public research materials on vulnerabilities and many details of vulnerabilities are missing. Patch comparison is an effective method for vulnerability analysis, but the firmware of backbone network equipment is usually unpacked into monolithic executable files, which have characteristics such as a large number of functions, a large file size, and missing debugging symbol information. Direct patch comparison will produce a large number of unconfirmed false positive differences, and heuristic algorithms may mistakenly match two unrelated functions, resulting in the lack of correct security patches and false negatives. Traditional patch comparison methods cannot effectively solve the patch analysis problem of these files, and the analysis of vulnerability details faces challenges. This paper proposes a method called MDiff for locating known vulnerabilities in monolithic executable files. MDiff decomposes the target binary file into internal modules based on the subsystem concept in the description of the vulnerability bulletin and the internal module structure of the target binary file, and uses semantic similarity measurement to filter and sort the comparison results based on binary comparison technology based on locality. Specifically, MDiff first uses entry functions and the principle of locality to identify vulnerable network protocol service codes, that is, the coarse-grained location phase. For the identified network protocol service code modules with vulnerabilities, MDiff performs semantic information analysis combining static and dynamic analysis, including the identification of security patches based on extended local traces and the ranking of security patches based on code metrics,

通讯作者: 邹燕燕, 硕士, 助理研究员, Email: zouyanyan@iie.ac.cn。
本课题得到自然科学基金项目(No. U1836209, No. 61802394)、中科院先导项目(No. XDC02040100)、国家重点研发计划(No. 2016QY071405)资助。
收稿日期: 2020-04-07; 修改日期: 2020-05-13; 定稿日期: 2022-12-20

tion of security patches based on extended local traces and the ranking of security patches based on code metrics, that is, the fine-grained location phase. Based on this two-phase vulnerability location method, we have implemented a prototype system and experimented with 15 vulnerabilities disclosed in devices from four vendors. The experimental results show that the proposed vulnerability location method can improve the efficiency of patch analysis for network devices and support researchers in discovering known vulnerability details.

Key words network equipment; module decomposition; patch comparison

1 引言

作为核心网、局域网中关键基础设施的大型网络设备, 一直是网络攻防中的焦点。2016 年 8 月, 被认为隶属于美国国家安全局(NSA)的方程式组织(Equation Group)的漏洞利用代码被曝光, 其中就包含大量网络设备漏洞利用代码。2017 年 3 月, 维基解密公布了美国中央情报局(CIA)的机密文件 Vault 7, 其中包括一个存在于 Cisco IOS 及 IOS-XE 系统中 CMP 集群管理协议的高价值漏洞, 通过该漏洞可以取得设备的完全控制权。2018 年 4 月, 诸多 Cisco 设备因使用带有已知漏洞的 Smart Install 组件导致设备被入侵, 其中俄罗斯和伊朗的大量设备瘫痪并被遗留美国国旗字条。除了市场领导者思科的网络设备外, 飞塔(Fortigate)设备、瞻博(Juniper)设备、沃奇卫士(WatchGuard) 设备、华为设备、天融信设备都被证明可入侵[1]。因此提高大型网络设备的安全性具有重要意义。

对网络设备进行漏洞挖掘是提高其安全性的重要手段。然而网络设备作为封闭、复杂的信息系统, 其漏洞的公开研究资料相对较少、漏洞细节缺失较多, 大大提高了网络设备漏洞挖掘的难度。以厂商 1 的网络设备为例, 虽然 2018 年官方披露大量的安全漏洞, 但有 90%都是通过内部安全审计和客户技术援助中心(TAC)上报完成, 这部分漏洞一般缺少漏洞细节, 无法进一步分析和发掘潜在问题。

补丁比对[2]是用于已知漏洞定位的最为广泛使用的方法之一。然而通过补丁比对方法定位漏洞存在 2 个主要的问题: 1) 基于补丁比对方法进行漏洞定位存在误报, 即对比结果中包含大量非安全修补, 由于固有的待确认差异多、筛选自动化程度低, 实际工作中需要花费大量人力对结果一一确认; 2) 基于补丁比对方法进行漏洞定位存在漏报, 如在没有调试符号的情况下, 比对启发式算法可能将两个不相关的函数错误的匹配, 导致正确的安全修补缺失。这 2 个问题在网络设备中更加严重。网络设备的封闭性和定制化程度高, 使得单个可执行文件更大, 因此待确认的差异更多, 误报的可能性更高, 导致漏洞定位效率和准确率问题更加严重。网络设备通常把网络协议服务代码与公共运行支撑代码等主要功能一起封装到一个可执行文件中, 本文将其称为单体式可执行文件。其中网络协议服务通常包括 SNMP、HTTP、IKE、OSPF、BGP 等, 涵盖所有管理平面、控制平面、数据平面内容。表 1 为单体式设备固件实例, 列出了不同设备的单体式可执行文件的基本属性。如表中的数据所示, 单体式可执行文件最大可达到 305 兆, 若通过 Bindiff[3]等工具进行补丁比对, 会产生多达 8429 个差异位置。同时, 待比对函数越多, 非线性的启发式算法对比效率越低, 甚至无法完成比对。总之, 单体式可执行文件在漏洞定位方面面临更大挑战。

表 1 单体式设备固件实例

Table 1 Monolithic equipment firmware example

厂商设备	单体式可执行文件规模(MB)	单体式可执行文件函数数量(k)
厂商 1 网络设备 1	83	98
厂商 1 网络设备 2	305	380
厂商 1 网络设备 3	227	300
厂商 2 网络设备	34	45
厂商 3 网络设备	144	240
厂商 4 网络设备	49.7	90
厂商 5 网络设备	24.7	40

(注: 单体式可执行文件需设备固件解包得到)

1.1 本文方法

通过对表 1 中各设备对应的单体式可执行文件的分析, 我们发现: 1)单体式可执行文件中的网络协议服务是以相对固定的模式启动并执行的; 2)在单体式可执行文件上应用传统乃至最先进的补丁比对工具, 海量的待确认函数给工具结果造成了质变的影响。因此本文提出了一种两阶段的漏洞定位方法 MDiff(MonolithDiff)。第一阶段是模块级的粗粒度定位, 通过识别网络协议服务的启动代码进而识别不同的网络协议服务代码, 将单体式可执行文件拆分成若干模块。通常情况下, 通过分析单体式可执行文件中的某一个模块即可定位已知漏洞, 因此我们根据已知漏洞的 CVE 描述信息, 选择需进一步分析的模块。第二阶段是函数级的细粒度识别, 在第一阶段识别出的模块中, 针对具有差异的函数, 利用动静

态结合的语义信息对补丁比对后的结果进行细粒度的分析, 即基于函数复杂性度量和脆弱性度量的静态语义分析以及基于函数模拟执行的动态语义分析, 最终完成对差异函数的筛选, 并按照确定程度自高而低排序, 识别安全修补代码。

基于该两阶段漏洞定位方法 MDiff, 我们实现了一个原型系统, 对 4 个厂商设备中已经披露的 15 个漏洞进行实验。结果表明, 本文的两阶段漏洞定位方法在分析时间花费相当的基础上, 平均仅分析 31 个差异函数即可定位漏洞函数。定位漏洞所需分析的差异函数个数分别是当前先进水平的补丁比对工具 Bindiff 和 Diaphora[4]的 0.42%和 0.38%, 说明本文方法的有效性。同时, 由于减少了需要比对差异的函数, 降低了比对误差, 使得 MDiff 比 Bindiff 多准确定位 1 个漏洞, 比 Diaphora 多准确定位 3 个漏洞。我们利用本系统, 从 4 个补丁中确认了实际设备中的未公开漏洞细节信息。

1.2 本文主要贡献与结构

我们的贡献主要在于:

1) 提出一种针对网络设备单体式可执行文件的两阶段漏洞定位方法, 通过模块级和函数级逐级定位的方式, 有效提高定位的精度;

2) 提出了基于服务启动模式的模块识别技术和动静态语义结合的修补函数识别及排序技术, 服务于漏洞定位;

3) 实现了一个原型系统 MDiff, 通过对 15 个真实已知漏洞的定位实验表明, 本文方法在定位效率和准确率方面都优于当前先进对比工具 Bindiff 和 Diaphora。

内容组织如下。本文第 2 节介绍相关工作; 第 3 节具体阐述系统的关键技术和算法, 包括: 基于模块识别的粗粒度定位方法、基于动静态语义信息的细粒度定位方法; 第 4 节给出实验设计和结果分析; 最后第 5 节对本文进行总结, 并讨论本文提出的系统存在的局限性, 以及下一步工作。

2 相关工作

2.1 二进制逆向分析

二进制逆向分析是一个复杂的过程, 不同于传统开发人员的程序理解, 逆向分析通常不能访问源代码、内部文档, 甚至需要克服符号剥离等反制措施。

目标二进制函数识别是进行二进制逆向分析的前提。函数识别[5]的主要问题即反汇编, 常见的反汇编方法包括线性扫描和递归遍历, 后者很大程度上克服了线性扫描无法有效区分代码和数据的缺点, 当前流行的反汇编工具 IDA Pro[6]使用的就是递归遍历算法。但由于静态分析无法准确识别间接跳转指令的跳转目标, 递归遍历面临巨大的挑战。虽然有研究人员尝试诸多方法解决反汇编的问题, 如符号执行方法[7-8]、动态分析方法[9-10]、机器学习方法[11]等, 但是依旧存在很多局限。除此之外, 函数识别面临的另一个问题是函数的高级语义恢复。Zeng J[12]通过跟踪内核对象执行过程的上下文信息, 自动推断内核对象的语义。其他工作[13-15]利用函数结构信息、调用 API 等特征, 将函数和变量名恢复问题形式化为一个机器学习问题。

目标子模块识别是二进制逆向分析的另一个主要内容, 通过寻找高层次的程序概述, 可以指导程序的哪些部分优先进行更复杂的调查[16]。在一些安全分析中, 尤其是在函数数量很大的情况下, 仅在函数级别分析未知二进制文件是费时或不充分的。模块识别技术所识别的模块实体[17]可以是文件、函数、类、进程等。自动化逆向识别子模块主要依赖图论方法[18]、信息检索方法[19-20]、模式匹配方法[21]等, 但目前绝大多数任务都工作在源码级。在二进制的探索方面, Karande V[22]提出了一种面向二进制文件的图聚类模块划分方法 BCD, 作者在 lzip 等小型二进制文件上做了验证, 但是需要标签训练权重, 并且划分的模块数量交由广义社区检测算法[23]自动生成, 容易出现过度分裂、欠分裂的问题。Votipka D 提出人工逆向识别子模块主要依赖 API 以及字符串信息。不同于 Votipka D 在整个固件文件集合中执行 grep 命令以查找 httpd 等文件, 如何在单体式可执行文件中自动识别和剥离出特定功能是本文关注的一个焦点。

2.2 二进制相似性分析

本文关注的网络设备固件代码主要以二进制形式存在。二进制代码相似度检测[24]是二进制补丁分析的主要技术手段, 该技术通过比较修补前、修补后程序的差异, 实现漏洞定位[2]。

常见的二进制相似度检测方法有基于静态方法的相似度检测、基于动态方法的相似度检测、基于机器学习方法的相似度检测等。其中, 基于静态方法的相似度检测技术, 通常将二进制代码转化为图, 然后进行比较。如以图形同构理论为基础的 Bindiff[3], 但图同构算法耗时较长, 缺乏多项式时间解, 同时很容易受到细微 CFG 变化的影响, 因此具有较低的准确性。以符号执行等程序分析方法辅助的工具如 BinHunt[25]、Spain[26], 通过符号执行扩展图同构来解决这些问题, 但是仍然具有较低的精度和较高的开

销。基于动态方法的相似度检测, 通常使用相同的输入执行两个二进制文件的函数, 通过比较执行结果进而衡量两个函数的相似程度, 比较有代表性的如Blanket[27]。动态方法擅长于提取代码的语义, 并且对编译器优化和代码混淆具有良好的包容性, 但是由于动态分析的性质, 这些技术通常具有较差的可扩展性和不完整的代码覆盖。除了传统的相似度检测方法, 其他有基于机器学习方法的相似度检测, 比较有代表性的如 αdiff[28]。

对大型网络设备固件的补丁比对研究相对较少, 肖等人[29]提出了一种节点层次化的二进制文件比对技术, 并在 Cisco 的固件上进行了比对, 达到了与 Bindiff 相当的对比能力, 并且将对比时间缩减为原来的二分之一, 节点匹配数量减少在15%左右, 降低了误匹配的概率。

3 两阶段漏洞定位方法

3.1 总体方法阐述

单体式可执行文件通常由若干个相对独立的网络协议服务代码和公共运行支撑代码组成。同时, 我们发现一般漏洞仅存在于特定的一个网络协议服务中, 因此对于修补前的单体式可执行文件以及补丁文件, 可以首先识别存在漏洞的网络协议服务代码, 即基于模块识别的粗粒度定位。然后针对已识别出的、存在漏洞的网络协议服务代码模块, 对于其中存在差异的函数进行动静态结合的语义信息分析, 即基于动静态语义的细粒度定位。由于安全补丁通常不会较大地改变目标的语义[26], 所以可以剔除语义变化大的非安全补丁对。由于结果缺乏面向专家确认的排序机制, 我们结合代码度量的静态语义信息对结果进行了进一步的排序, 从而完成函数级的漏洞定位。通过上述两阶段方法, 基于补丁完成网络设备已知漏洞的定位。

方法的整体流程如图 1 所示。

第一阶段进行基于模块识别的粗粒度分析, 定位漏洞所在模块, 即网络协议服务的处理代码。首先通过两个步骤将单体式可执行文件分解为模块。具体而言, 第一步在基于进程启动模式识别出所有的模块入口函数后, 通过程序的静态调用图, 获取每一个入口函数可达的函数, 这些函数与入口函数属于一个模块。特别的, 如果一个函数可由多个入口函数可达, 那么这个函数同时属于多个模块。程序中存在的间接调用可能导致静态分析无法获取全部的调用函数。为弥补该问题, 第二步利用同一模块的函数在地址空间上具有局部性的特点, 将入口函数在地址空间邻近的函数也归纳到入口函数所在的模块中。通过上述两个步骤即可实现单体式可执行文件的模块分解。之后, 我们基于已披露的漏洞信息, 即 CVE 描述, 在已经分解得到的模块集合中筛选出目标模块, 从而完成模块级粗粒度的漏洞定位。

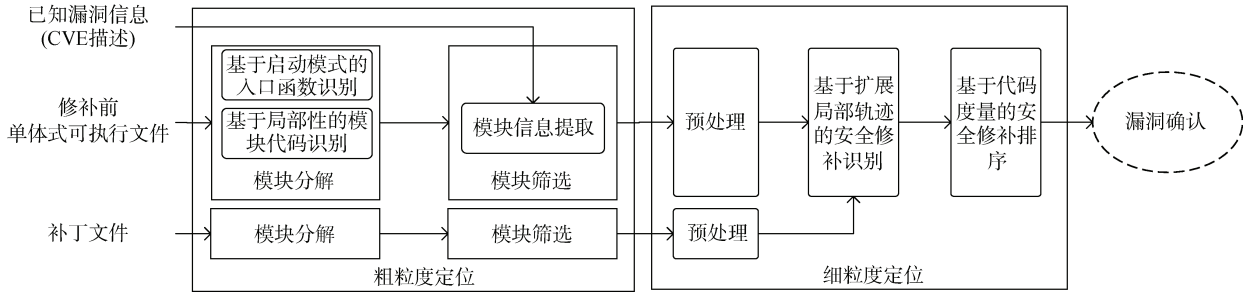


图 1 MDiff 整体流程图

Figure 1 MDiff overall flow chart

第二阶段进行基于动静态语义结合的细粒度定位。为了消除程序修补前后由于编译优化、安全选项等造成对比不准确的问题, 首先进行预处理, 将二进制代码转化为统一反编译后的中间表示, 并通过归一化手段消除偏移等影响。其次, 对于预处理后的修补前、修补后代码, 我们选择有差异的函数对, 使用模拟执行的方法获得差异函数的执行语义并比较。在函数模拟执行过程中, 我们使用扩展局部轨迹的概念扩大模拟执行的基本单元, 从而获得更准确的执行语义。最后, 由于安全修补对函数的执行语义影响较小[26], 我们剔除了执行语义相似度小于阈值的函数对, 并进一步度量函数代码复杂度, 将更可能是安全修补的函数排序在前, 从而完成细粒度的漏洞定位。

3.2 基于模块识别的粗粒度漏洞定位

通常情况下, 网络设备在系统加电引导后, 有两种启动单体式可执行文件的方式。一种由操作系统执行, 另一种由网络设备直接启动。网络协议服务

在单体式可执行文件被加载执行后, 以守护进程的方式启动并执行。单体式可执行文件自身实现了线程/进程调度代码, 从而支持类似内核线程的机制, 以支持多个协议服务的并行或并发处理, 各协议服务处理过程通常运行在同一个地址空间中。以厂商 1 设备为例, 通过设备内置的内部进程打印命令, 显示该设备存在 139 个正在运行的网络协议服务, 表 2 为截取部分网络协议服务守护进程的示例。

表 2 典型正在运行的网络协议服务

Table 2 Typical running network protocol services

webvpn_task
WebVPN KCD Process
snmpfo_timer_thread
SNMP Notify Thread
SNMP Host Timer Thread
snmp
IKE Timekeeper
IKE Receiver
IKE Daemon
IKE Common thread
aaa_shim_thread
aaa-url-redirect-task
……

CVE 信息中通常包含目标漏洞的一些简要信息, 包括漏洞受影响版本、漏洞类型和漏洞存在的模块, 如图 2 所示, 我们发现漏洞模块与设备网络协议服务名称是对应的。以协议功能为模块粒度, 一方面可以拟合单体式网络设备的运行特性, 另一方面可以有效与漏洞描述信息对应。在确定协议功能为模块粒度后, 由于单体式可执行文件中通常没有符号信息, 因此如何找到网络协议服务进程的入口函数及该服务的相关功能函数是模块分解所解决的主要问题。

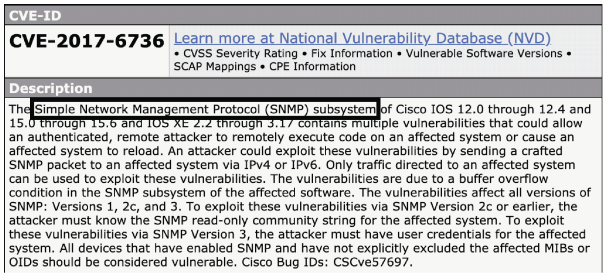


图 2 漏洞中的子系统描述

Figure 2 Subsystem description of vulnerability

3.2.1 入口函数识别

进程入口函数通常进行数据包队列中的数据获取, 其后续调用的函数进行具体的协议解析、协议状态转换、状态功能处理等。进程入口函数识别有两个关键的观察。第一, 部分网络协议服务以守护进程模式运行, 在后台持续监听数据包消息队列。因此, 我们可以通过守护进程的特征寻找到符合特征的守护进程入口函数。第二, 以进程形式存在的网络协议服务具有固定的启动模式。我们可以利用进程创建等启动模式, 对其余进程入口函数进行识别。

入口函数是网络协议服务代码的入口点。通过对所有服务入口函数进行定位和分析, 可以较好地拆解目标文件结构。因此, 本文将协议功能子系统、子进程作为目标拆分粒度, 对应的子系统、子进程入口函数作为识别和拟合目标模块结构的锚点。

动态运行中表现为后台时刻运行的网络协议服务, 在静态代码中对应为永不返回、存在多个循环结构的入口函数。这些守护进程入口函数 daemon()的代码结构如图如图 3 所示。

```
def daemon():
    ...
    while (true):
        req = dequeue_request()
        process(req)
    ...
```

图 3 守护进程入口函数 daemon()

Figure 3 Daemon entry function daemon()

大多数单体式设备中, 服务进程都是通过进程创建函数创建, 如图 4 所示, 该函数的参数会指明所创建进程的函数名称, 进而可将服务进程与模块名称进行映射。同时根据其引用, 可以补全非守护进程模式的进程入口函数。具体过程如算法 1 所示。本文在 4.3 节对该方法的准确度等问题进行了进一步的讨论。

```
def subsystem():
    ...
    ret=process_create(daemon_name, daemon, 0, 0)
    if (ret!=0) :
        exit(1)
    ...
```

图 4 进程创建函数 process_create()

Figure 4 Process creation function process_create()

3.2.2 基于局部性的模块代码识别

为提取网络协议服务模块的代码, 即模块函数, 本文首先进行了预识别, 利用函数调用关系, 将模块入口函数后续调用的所有函数视为该模块的函数, 以得到**初始的模块代码**。

由于静态分析中存在间接调用的问题, 往往会造成调用关系丢失, 影响粗粒度定位的效果。而间接引用的分析[10, 30]通常较为复杂, 基于指针分析的保守性会引入大量无关函数。本文采用一种轻量级局部性识别方法。经过统计分析, 除去公共调用函数后, 初始目标模块函数呈现多点聚集状态且大部分聚集在入口函数附近。因此, 我们利用局部性的思想, 将入口函数附近的函数作为补充, 在较低的开销下实现模块函数补充。具体来讲, 由于 3.2.1 节中识别的模块入口函数均匀分布在整个 text 段空间, 我们将地址空间上, 与目标入口函数相邻最近的两入口函数间的函数视为**局部的模块函数**, 并纳入目标模块函数集合。

此外, 不同模块入口函数得到的调用结果中存在大量相同函数, 即存在公共函数调用的情况, 因此, 提取所有的模块函数之后, 需要去除各模块中公共调用函数。

整体的模块代码识别算法描述如算法 1 所示。第 1～16 行, 即基于模块识别的粗粒度漏洞定位, 将待分析函数限定在单个特定模块代码中。第 1～5 行对可执行文件中的所有函数迭代分析, 寻找具有守护进程模式的函数, 即永不退出 is_noretn, 具有一个以上循环结构 has_loop。第 6 行根据守护进程以及网络协议服务的启动模式寻找其余的进程入口函数, 这里通过进程创建的启动模式, 找到进程创建函数 processcreate 的地址 *addr_procreate*。第 7～10 行, 根据 processcreate 的引用, 找到所有的进程函数及其进程名, 至此我们得到了识别和拟合目标模块结构的锚点。第 11～16 行对包含特定目标模块名的入口函数进行分析, 这里将所有与特定目标模块名相关的入口函数都作为待分析的潜在模块入口函数。首先根据 gen_complex_call_chart 函数得到初始的模块函数, 即利用函数引用和数据引用关系, 将模块进程入口函数后续调用的所有函数视为该进程的模块函数。之后利用 get_funcs_between 函数将局部的模块函数纳入目标模块代码中, 得到最终的目标模块函数 fs_procreate。

算法 1.模块代码识别算法

输入: *fs* : 单体式可执行文件中的所有函数
module_name : 目标 CVE 描述中的模块名
输出: *fs_procreate* : 目标模块函数

```
1  FOR each f ∈ fs DO
2      IF is_noretn(f) && has_loop(f) THEN
3          fs_daemon = fs_daemon U {f}
4      ENDIF
5  ENDFOR
6  addr_procreate = get_characteristic(fs_daemon)
7  addrs_proc = xrefsto(addr_procreate)
8  FOR each addr_proc ∈ addrs_proc DO
9      fs_proc = fs_proc U
   {get_function_name(addr_proc):addr_proc}
10 ENDFOR
11 FOR each fcur ∈ fs_proc DO
12     IF fcur->name.contain(module_name) THEN
           fs_procreate = fs_procreate U
13 {gen_complex_call_chart(fcur, CHART_REFE-
   RENCED+CHART_RECURSIVE)}
           fs_procreate = fs_procreate U
14 {get_funcs_between(fcur->prev, fcur->next)}
15 ENDIF
16 ENDFOR
```

基于局部性的模块代码识别示例如图 5 所示, 首先根据模块入口函数的函数调用关系, 迭代地分析其调用路径上的所有函数, 将调用图上的所有函数视为目标模块函数集合, 图 5 中粗实线的 WEBVPN1 为一个 webvpn 模块入口函数, 其调用图如实线曲线所示, 包含直接调用以及迭代的调用链上的调用。然后根据模块入口函数的位置关系, 将入口函数周围的局部函数也纳入目标模块函数集合, 如图 5, 将包含在 DHCP、CMP 入口函数间, 即灰色区域内的函数纳入 webvpn 目标模块集合。同理, 继续提取 webvpn 的其余相关入口函数的模块函数集合, 如 WEBVPN2。最后, 通过提取所有模块入口函数相关的模块函数, 进一步计算并删除公共调用函数。

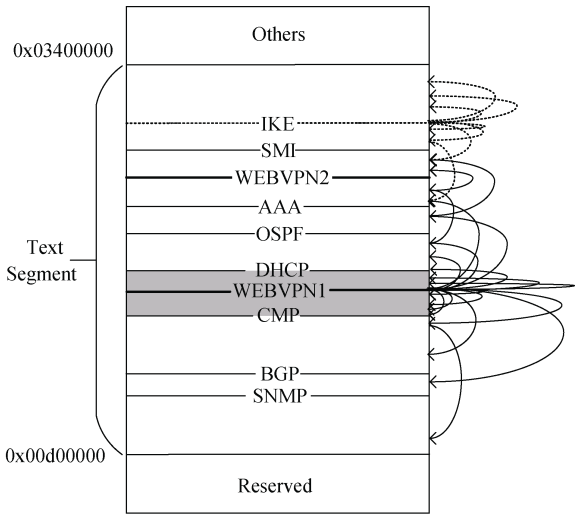


图 5 模块代码识别过程示意图

Figure 5 Schematic diagram of module code identification process

至此得到了目标模块函数集合, 结合 CVE 描述中的子系统信息, 我们将待分析的补丁对限定在单个特定模块代码中。

3.3 基于动静态语义的细粒度漏洞定位

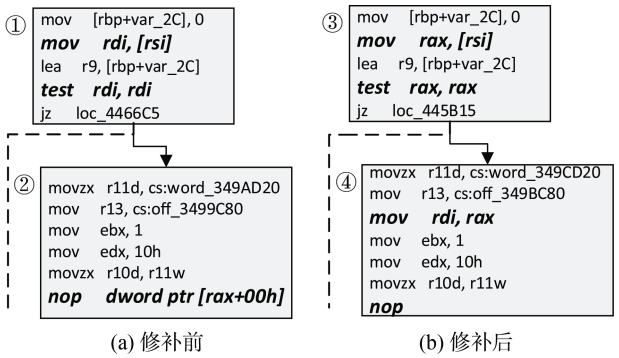
对单体式可执行文件进行模块分解与筛选后, 排除了其他模块的干扰, 仍需细粒度方法对补丁函数进行进一步的定位。针对现有模拟执行、比较执行 语义方法存在的不足, 本文提出一种动静态结合的 函数级细粒度漏洞定位方法, 可以有效识别差异函数中的安全修补, 完成更精细的漏洞定位。

3.3.1 现有方法不足

当前典型的函数级安全修补识别方法, 是 Xu Z 等人[26]提出的 Spain。该方法基于安全修补对程序语义影响较小的假设来识别安全修补。具体而言, 通过模拟执行函数修补前和修补后的差异基本块, 并比较执行状态的差异程度, 得到修补前和修补后函数的相似度。如果两段指令执行终止时, 相同的执行状态越多, 表明两段指令的语义相似度越高。最后, 相似度高于阈值之上的补丁对即为安全修补。

虽然 Spain 对自动化补丁比对进行了良好的探索, 但是该方法应用到实际的安全修补识别中, 在分析准确性和效率方面会存在如下不足:

(1) 由于修补前和修补后编译优化选项及安全选项的不同, 可能造成安全修补误报。如图 6 所示, 修补后的指令中, 寄存器分配算法使用 rax 进行计算, 之后将计算结果存入 rdi 中, 基本块④中新增的语句“mov rdi, rax”使得 Spain 认为修补前后的指令具有明显差异, 实际上, 修补前和修补后的两段代码片段具有相近的执行语义, 最终由于执行语义相似度高而被误判为安全修补。

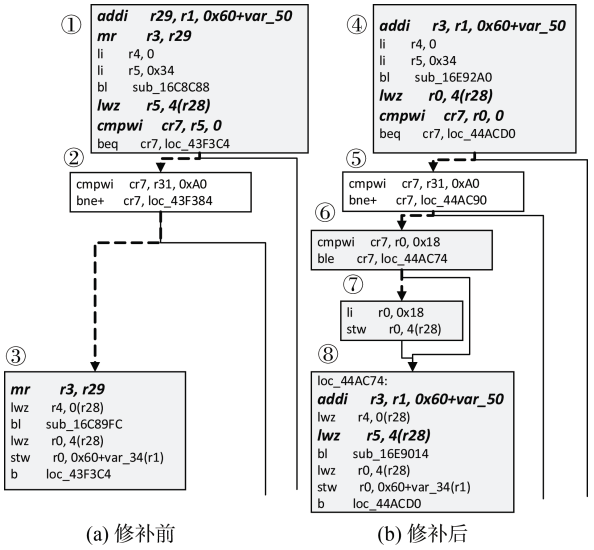


(a) 修补前 (b) 修补后

图 6 编译环境不同造成的指令增加

Figure 6 Increased instruction due to different compilation environments

(2) 由于仅识别连续差异基本块, 对于不连续的语义相近的差异基本块可能误报为非安全修补。如图 7 所示, 虽然修补前和修补后的代码有变化, 但是执行语义是几乎相同的。由于 Spain 只会针对连续的差异基本块进行执行语义判别, 对如图 7 所示的代码片段, 由于①和③、④和⑥中间分别有相同基本块②⑤, 分割成两段分别计算, 增大了执行语义差异, 从而可能造成误报。



(a) 修补前 (b) 修补后

图 7 局部轨迹下的分割问题

Figure 7 Segmentation problem caused by partial traces

(3) 未考虑基本块删减类的修补方式, 从而存在安全修补漏报。如果一个修补是从修补前代码中删减基本块, Spain 的修补基本块提取算法认为前后不具有差异, 从而造成安全修补漏报。

(4) Spain 所采用的基于符号公式的模拟执行方法, 效率存在不足。Spain 根据 Chandramohan M 的 Bingo 以及 Pewny J 的方法[31-32], 基于符号公式进行函数级的模拟执行, 通过比较结束时执行状态的差异程度, 判断是否为安全修补。虽然通过输入具体值来简化计算, 但由于需要提取符号公式等至少两步操作, 符号公式存在路径爆炸等固有问题, 依旧存在效率方面的不足。

(5) 在总结补丁和漏洞模式的过程中, 除了添加校验类的修补模式, 漏洞定位的过程仍然需要手工确认, 没有进一步的优先级排序机制。测试中最差情况下手工确认工作量达到 342, 依旧存在提升空间。

为了解决(1), 在 3.3.2 节的预处理中, 我们将第一步补丁比对结果中的有变化函数对进行统一反编译、标准化和安全机制识别, 降低编译环境不同造成的误报。为了解决(2)(3)(4), 在 3.3.2 节的安全修补识别方法中, 我们引入扩展局部轨迹的概念, 通过删除修补前后无变化基本块, 并联结剩余的连续差异

基本块集合, 扩大分析范围。之后, 基于仿真模拟执行计算扩展局部轨迹的执行语义, 将相似度最高的扩展局部轨迹相似度作为修补前后函数的语义相似度, 识别安全修补。为了解决(5), 在 3.3.2 节的基于代码度量的安全修补排序方法中, 我们引入了基于函数复杂性度量和脆弱性度量的静态语义筛选, 对安全修补函数进行进一步筛选排序, 精细化漏洞定位结果。

3.3.2 基于动静态语义的细粒度定位方法

(1) 预处理

在编译环境(如编译工具、编译优化等级等)不同的情况下, 即使相同的源程序也会编译出不同的二进制代码, 从而出现基本块分裂、基本块合并、指令重排、指令替换等现象。这种语法级的变化给后续的安全修补识别带来了误报。

本文使用反编译方法将原始代码翻译到统一的中间表示, 并基于中间表示进行比较, 识别部分仅由编译优化造成的代码差异。经过反编译优化的函数, 会存在内存偏移不同、特殊情况下使用的寄存器名不同等问题, 因此这里引入了标准化过程, 将中间表示中的操作数规范化为 reg、imm 或 mem 类型。例如, r1=INT_ADD R1, 16:4 规范化后为 reg=INT_ADD reg, imm。

对于修补后版本引入新的安全机制(如 canary、CFI、PAC 等)的问题, 本文对常见的 canary 模式进行了定义和识别, 将仅由 canary 校验带来变化的补丁对筛除。具体而言, canary 通常会在函数末尾增加一个调用检查的基本块, 且函数的序言和尾声也会改变, 本文通过检查是否存在调用检查基本块, 以及基本块的变化数量, 将仅由引入新安全机制带来变化的补丁对识别并剔除。

(2) 基于扩展局部轨迹的安全修补识别

本文提出了基于扩展局部轨迹的安全修补识别方法, 将多组局部轨迹根据前驱后继关系合并, 一方面能够减少误报, 同时能够提高计算的效率。

这里首先需要引入一个概念:

定义 1.局部轨迹(partial traces):

给定一个函数, 函数的控制流图(CFG)记为元组 $G=(N, E, Ns, Nt, \iota)$。其中 N 是一组有限的节点, 每个节点代表函数的一个基本块; E: $N\times N$ 是一组连接两个节点的边, 表示两个基本块之间的控制流; Ns, $Nt\subset N$ 分别为函数的开始节点和结束节点, ι 是一个将节点映射到基本块的函数, 对于每个节点 $n\in N$, 节点对应的基本块为 $\iota(n)$。若给定一个函数的控制流图 G, 那么 G 中的局部轨迹 t 是一个有限的节点序列 $<n_1, n_2,\ldots,n_k>$, $k\geqslant 1$, 其中 $(n_i, n_{i+1})\in E$, $1\leqslant i<k$。

给定修补前后的两个函数控制流图 *Go*、*Gp*, *Gp* 中的修补局部轨迹集合 *Tp* 为 $\{tp_1, tp_2, \ldots, tp_n\}$。对于 *Gp* 中任意两个局部轨迹 tp_{k1}、tp_{k2}, 如果 tp_{k1} 与 tp_{k2} 中任意节点都不相同; 同时在 tp_{k1} 中存在一个叶子节点 n_l, 与 tp_{k2} 中一个根节点 n_r 路径可达, 则根据路径的前驱后继关系将叶子节点 n_l 与根节点 n_r 联结, 我们将联结后的局部轨迹称为 *Gp* 的扩展局部轨迹 tp_k'。*Gp* 的扩展局部轨迹集合 *Tp*′为 $\{tp_1', tp_2', \ldots, tp_m'\}$, 其中 $m\leqslant n$。同理可得 *Go* 中的扩展局部轨迹集合 *To*′。如 3.3.1 节图 7 所示, 右侧函数存在 3 组局部轨迹 {<4>, <6, 7, 8>, <6, 8>}, 合并后存在 2 组扩展局部轨迹{<4, 6, 7, 8>, <4, 6, 8>}。

扩展局部轨迹相比于原始的局部轨迹, 能够有效地识别更多的安全修补。不同于简单合并多组局部轨迹, 我们提出了一种基于删减基本块的扩展局部轨迹提取算法, 通过迭代删减无变化的基本块, 并修复前驱后继关系, 以得到扩展局部轨迹。该算法相比于原始算法更加准确, 详细过程见算法 2。

对于问题 3, 在定位修补基本块的过程中, 原始算法无法识别删减基本块的修补方式。Spain 方法首先识别修补后的有变化基本块, 作为后续安全修补识别的基础。但漏洞修补时可能仅删去一些基本块, 导致没有出现有变化基本块, 最终使得 Spain 认为修补前与修补后无变化, 从而造成安全修补的漏报。基于扩展局部轨迹的安全修补识别中的有变化基本块定位方法可以有效应对删减基本块的修复方式, 本文在 4.4 节对这个问题进行了更详细讨论。

算法 2 扩展局部轨迹提取算法

输入: *init_patch_pair*: 初始补丁对

输出: *pts_op*: 修补前与修补后对应的扩展局部轨迹集合

```
1   FOR each fo, fp ∈ init_patch_pair DO
2       g_fo, g_fp = get_graph_cfg(fo, fp)
3       vs_fo, vs_fp = get_vertex(g_fo),
    get_vertex(g_fp)
4       vs_inv = find_invariant_bbs(g_fo, g_fp)
5       FOR each v ∈ vs_fo DO
6           IF v ∈ vs_inv THEN
7               vs_p, vs_c = v->parents,
    v->children
8               FOR each v_p ∈ vs_p DO
9                   FOR each v_c ∈ vs_c DO
10                      g_fo.add_edge(v_p, v_c)
11                  ENDFOR
```

```
12                  ENDFOR
13                  g_fo.remove_vertex(v)
14              ENDIF
15          ENDFOR
16          Remove the vs_inv node in g_fp in the same way.
17          pts_fo, pts_fp = get_all_paths(g_fo, g_fp)
18          pts_op = pts_op U (pts_fo, pts_fp)
19      ENDFOR
```

整体的扩展局部轨迹提取算法描述如算法 2 所示。第 1～19 行, 分析预处理后的初始补丁对, 得到修补前与修补后对应的扩展局部轨迹集合。第 2～3 行通过 get_graph_cfg 获得补丁对的 CFG, 其中 *g_fo*、*g_fp* 表示修补前和修补后 CFG 构成的图。第 3 行得到图中的顶点集合 *vs_fo*、*vs_fp*, 即基本块节点集合。第 4 行通过 find_invariant_bbs 判断修补前与修补后的两个图中完全相同的节点。第 5～15 行在 *g_fo*、*g_fp* 中删除这些无变化的节点, 同时根据删去节点的父子节点前驱后继关系, 在父子节点修复并建立新的前驱后继, 得到新的 *g_fo*。第 16 行, 同理得到新的 *g_fp*。第 17 行, 在新 *g_fo* 与 *g_fp* 中, 根据出度入度得到开始和结尾节点集合, 并计算在始末节点间的所有执行路径, 即扩展局部轨迹。

最后我们利用与 Spain 相似的语义计算方法完成目标语义状态的计算与安全修补筛选。受 Bingo-E[33]启发, 本文采用了基于 Ghidra 仿真的模拟执行方法, 从而解决了问题 4。与此同时, 相比于 Spain 以及基于 unicorn 仿真的 Bingo-E, 我们所提出的 MDiff 可支持的架构更多。另外, 本文在模拟执行的过程中选取了多组状态初值, 任一输入计算出的语义相似度大于阈值, 该补丁对都被视为安全修补, 以降低安全修补漏报的概率。多样化的输入以及语义摘要计算过程提取了所有可能的执行路径, 即忽略了目标的条件跳转语句, 从而保证了基于扩展局部轨迹计算的有效性。详细过程见算法 3。

算法 3 基于模拟执行的安全修补识别算法

输入: *pts_op*: 修补前与修补后对应的扩展局部轨迹集合

输出: *fs_semantic*: 根据执行语义信息筛选后的补丁对

```
1   FOR each pts_fo, pts_fp ∈ pts_op DO
2       scores_max = 0
3       FOR each init_state ∈ init_states DO
4       cal_poststate(pts_fo, init_state)
5       cal_poststate(pts_fp, init_state)
6       score_max = 0
7       FOR each pt_fo ∈ pts_fo DO
8           FOR each pt_fp ∈ pts_fp DO
9               score_cur = get_score(pt_fo, pt_fp)
10              score_max = score_cur IF THEN score_cur>score_max ELSE score_max
11              ENDFOR
12          ENDFOR
13          scores_max = score_max IF THEN score_max>scores_max ELSE scores_max
14      ENDFOR
15      IF scores_max > threshold THEN
16          fs_semantic = fs_semantic U {f:scores_max}
17      ENDIF
18  ENDFOR
```

基于模拟执行的安全修补识别算法如算法 3 所示, 第 1～18 行为基于模拟执行的安全修补识别算法, 以筛除语义变化大的非安全补丁对。第 2～3 行, 选取多组寄存器内存状态初值 *init_states* 对执行路径进行模拟执行, 任一输入计算出的语义相似度大于阈值, 即最大值 *scores_max* 大于阈值, 补丁对都被视为安全修补。这里对状态初值的设置同 Bingo-E, 选取了 3 组寄存器内存初值 0x0、0x1111、以及全部随机。第 4～5 行, 循环使用相同的初始值对所有执行路径进行模拟执行, 使用 cal_poststate 方法得到执行后的寄存器及内存值, 作为该路径的语义摘要。第 6～13 行, 通过 get_score 方法计算两组语义摘要的相似度, 以两者最高的执行路径语义相似度作为两个函数的语义相似度。第 15～17 行, 根据阈值筛除相似度低的功能性修补, 保留语义相似度高的为安全修补。在实验中, 我们根据经验将阈值 *threshold* 设置为 0.7。

如图 8 所示, 一个算法 2 和算法 3 执行示例如下。(i)首先定位扩展局部轨迹, 易见 (1, 3)和(1′, 3′, 7′)分别为修补前、修补后的有变化基本块, 即图 8 中红色基本块。具体的扩展局部轨迹定位过程如下: 首先在补丁对中对所有基本块进行一轮对比, 得到修补前和修补后无变化的基本块集合 *vs_inv*, 然后依次在控制流图中将 *vs_inv* 内的基本块删除, 同时根据删去基本块的父子节点前驱后继关系, 在父节点、子节点修复并建立新的前驱后继, 最终即可得到修补前后的节点构成的新控制流图。考虑图 8 右侧函数, 有且仅有一个新控制流图 *Gp′*为({1′, 7′, 3′}, {(1′, 7′), (7′, 3′)}, {1′}, {3′}, *ι*)。然后根据出度和入度得到 *Gp′*的开始和结尾节点集合, 并计算 *Gp′*在始末节点间的

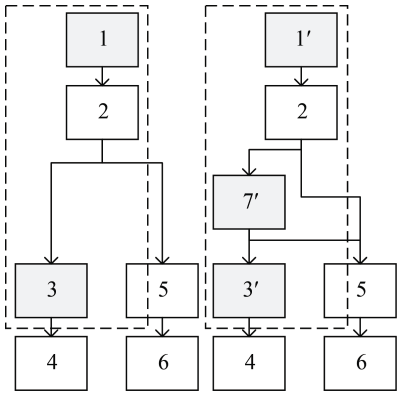


图 8 安全修补识别过程示意图

Figure 8 Schematic diagram of the security patch identification process

所有执行路径, 即扩展局部轨迹。如图 8 右侧函数 *Gp′*有且仅有一条扩展局部轨迹{<1′, 7′, 3′>}。(ii)然后对所有执行路径, 即扩展局部轨迹, 进行模拟执行, 计算修补前与修补后函数的语义相似度。通过相同的初始值对所有执行路径进行模拟执行, 执行后的寄存器及内存值即该路径的语义摘要。如在 1 和 1′处给定一组相同的寄存器和内存初值, 在 3 和 3′的处即可得到执行结束的寄存器和内存值, 执行结果<mem', reg'>即目标执行路径的语义状态表示。同理将两个函数的多组执行路径进行计算和比较, 语义相似度最高的值即这个函数对的语义相似度。另外这里需要选取多组寄存器内存状态初值对执行路径进行模拟执行。若任一输入计算出的语义相似度大于阈值, 则该补丁对都被视为安全修补。由于示例中的仅有一组扩展局部轨迹, 因此{<1, 3>}与{<1′, 7′, 3′>}的语义相似度即这组函数的语义相似度。

(3)基于代码度量的安全修补排序

由于单体式可执行文件规模较大, 在使用基于动态语义相似度的安全修补识别后, 仍然存在很多待人工确认的潜在安全修补, 最差情况下需要分析多达 342 个补丁对。因此, 本文引入了基于函数复杂性、函数脆弱性的静态语义分析, 在不针对特定漏洞的前提下, 对潜在安全修补结果进行进一步的排序。

根据 Leopard 等人[34]的发现, 一个函数如果脆弱性程度更高, 那么针对该函数的修补就更可能是一个安全修补。本文受此启发, 首先利用复杂性度量, 即函数圈复杂度大小 C1、函数循环结构数量 C2 将目标补丁对进行分箱, 以避免遗漏复杂度低但潜在脆弱的函数对。接着利用脆弱性度量对每个分箱中的补丁对进行排序, 如果修补前和修补后函数的控制结构 V1 越多, 那么这个补丁越可能是一个安全补丁。由于敏感函数的使用不当是导致漏洞的重要原因, 本文以是否存在敏感函数的引用作为补充, 即 V2。另外由于大部分安全补丁都是通过添加校验完成修补, 同时借助具有修补前和修补后两个函数的优势, 本文引入了函数圈复杂度增加更可能是加入了校验的脆弱性语义 V3。

表 3 静态语义衡量的维度

Table 3 Dimensions of static semantic measurement

类别	示例
C1	函数圈复杂度大小
C2	函数循环结构数量
V1	函数控制结构数量
V2	函数是否调用敏感函数*
V3	函数是否圈复杂度增加

(注: *仅对有库函数符号的目标而言)

4 实验

我们实现了漏洞定位原型系统 MDiff。该系统使用 Java 语言及 Python 语言实现, 代码量约 5 千行。其中, 基于模块识别的粗粒度定位部分主要由 Python 语言实现, 借助 IDA Pro 完成函数的调用引用、数据引用的分析。我们预先在启发式算法中限制每个函数的可匹配范围, 通过修改 Diaphora 的源码完成的基础补丁比对。基于动静态语义信息的细粒度定位部分主要由 Java 语言实现, 基于 Ghidra 完成预处理、修补基本块识别、函数模拟执行、函数语义相似度计算、静态语义信息提取相关的代码。其中圈复杂度、循环结构数量等代码度量基于 Diaphora 计算得到。

整体系统运行在 MBP 2016 上, 2.6 GHz Quad-Core Intel Core i7, 16 GB 2133 MHz LPDDR3, 256GB SSD, 运行的软件版本为 MacOS Catalina 10.15.1。IDA pro 版本为 7.0, Ghidra 版本为 9.0.4, Diaphora 版本为 1.2.4, Bindiff 版本为 5.0。

本章首先在 4.1 节介绍测试用例; 4.2 节阐述整体分析情况; 4.3 节讨论模块分解结果; 4.4 节讨论漏洞定位效果。

4.1 测试用例

从设备厂商、设备类型、漏洞类型、漏洞发布时间、处理器架构、操作系统类型等角度出发, 我们选择的测试用例如表 4 所示, 主要包括目标设备 1 的 snmp、cmp、smi 等 3 个漏洞; 目标设备 2 的 http、ike、snmp、webvpn 等 4 个漏洞; 目标设备 3 的 cifs 的 1 个漏洞; 目标设备 4 的 sslvpn 的 2 个漏洞; 目标设备 5 的 ssh 的 1 个漏洞; 目标设备 6 的 lldp、dhcp 等 4 个漏洞。其中, 表 4 内序号 1～11 的漏洞所在位置根

表 4 测试用例

Table 4 Test case

序号	CVE	漏洞类型	设备信息
1	CVE-2017	栈溢出	目标设备 1 (厂商 1, PowerPC)
2	CVE-2017	栈溢出	
3	CVE-2018	栈溢出	
4	CVE-2018	空指针解引用	
5	CVE-2016	堆溢出	目标设备 2 (厂商 1, x86)
6	CVE-2016	栈溢出	
7	CVE-2018	Double Free	
8	CVE-2017	堆溢出	目标设备 3 (厂商 1, x86)
9	CVE-2018	路径穿越	目标设备 4 (厂商 4, x86-64)
10	CVE-2018	堆溢出	
11	CVE-2015	后门账号	目标设备 5 (厂商 2, ARM)
12	CVE-2018	格式化字符串	
13	CVE-2018	栈溢出	目标设备 6 (厂商 1, PowerPC)
14	CVE-2018	堆溢出	
15	CVE-2018	整数溢出	

据网上公开信息分析证实, 特别的, 序号 1、2、4、6、8、11 的漏洞所在位置并非漏洞直接披露人给出, 我们综合多方面信息构造 PoC 分析确认这是该漏洞的所在位置。序号 12～15 是利用本系统确认的未公开细节补丁的漏洞所在位置, 为了详细说明这 4 个漏洞的定位情况, 我们将其也纳入了对比范围。

4.2 整体分析情况

我们主要从测试时间、结果数量、结果准确性等方面来评估 MDiff 方法的性能。在实验中, 选取了当前广泛使用的两个补丁分析工具 Bindiff 和 Diaphora 做对比, 其中 Bindiff 为闭源工具, Diaphora 是开源工具。实验结果如表 5 所示。漏洞定位时间一列主要列出了三个工具的分析所需时间, 其中 Mdiff 给出了对比阶段、模块拆分阶段、预处理阶段、语义比较阶段的具体用时, 以及总用时; 待分析修补数量一列主要列出了三个工具分析的结果, 其中 Mdiff 给出了模块拆分前、模块拆分后、预处理后、动态语义筛选后、静态语义排序后各阶段的结果; 漏洞定位结果一列中“√”和“×”代表最终漏洞定位结果的正误, 带有“*”的行表示由于补丁比对过程中限制了函数可匹配范围, 使得拆分前, 即基础补丁比对阶段结果正确。

表 5 整体补丁比对效果对比

Table 5 Comparison of overall patch comparison effects

序号	漏洞定位时间(s)							待分析修补数量							漏洞定位结果		
	B	D	M					B	D	M					B	D	M
			总时间	对比时间	模块拆分	预处理	语义比较			拆分前	仅拆分	预处理	动态语义	静态语义			
1	1126	29244	30242	29160	752	282	48	1651	8271	8145	403	59	56	46	√	×	×*
2	1126	29244	29930	29160	751	7	12	1651	8271	8145	149	22	22	10	×	×	×
3	1126	29244	30190	29160	706	256	68	1651	8271	8145	246	67	53	12	×	√	√
4	1222	22172	24476	21391	2230	589	266	2128	2505	1682	617	260	148	1	√	√	√
5	1222	22172	24368	21391	2182	563	232	2128	2505	1682	667	292	164	85	√	×	√*
6	1222	22172	23562	21391	2131	28	12	2128	2505	1682	26	11	9	2	√	√	√
7	1222	22172	24469	21391	2231	611	236	2128	2505	1682	540	258	144	77	√	√	√*
8	1087	19464	21186	18328	2046	391	421	1622	1169	1083	408	209	137	5	√	√	√
9	3314	28905	32255	28860	2463	596	336	37029	23203	22958	759	440	342	75	×	×	√*
10	3314	28905	32255	28860	2463	596	336	37029	23203	22958	759	440	342	44	×	×	×
11	791	10831	11480	10819	537	118	6	15602	5034	4888	317	11	10	6	√	√	√
12	1197	29284	30064	28909	1051	85	19	1398	8138	7758	44	9	9	9	√	√	√
13	1197	29284	30064	28909	1051	85	19	1398	8138	7758	44	9	9	3	√	√	√
14	1197	29284	30329	28909	1136	206	78	1398	8138	7758	284	62	46	42	√	×	√*
15	1197	29284	30329	28909	1136	206	78	1398	8138	7758	284	62	46	41	√	√	√

(注: B 代表 Bindiff、D 代表 Diaphora、M 代表 MDiff)

在测试时间方面, 我们提出的 MDiff 方法平均分析用时与所修改的基础工具 Diaphora 相当。而 Diaphora 和 MDiff 的测试用时是 Bindiff 的 19 倍, 主要原因是 Diaphora 由 Python 语言编写, 运行速度较慢。在具体的对比用时方面, 由于限定了启发式算法可匹配的函数空间, MDiff 在对比时间上与原工具

Diaphora 相比有微弱优势。在具体的粗粒度模块识别和细粒度模拟执行的用时方面, MDiff 平均用时为 1977s, 仅占整体用时的 7%, 说明模块拆分与细粒度识别的高效性。

在结果数量方面, 借助多级的筛选排序机制, MDiff 最终给出的平均结果数量为 Bindiff 和 Diaphora 的 1%, 最差的情况下能将结果数量降低到原本问题的 4%以下。MDiff 平均仅在 31 个差异中完成定位, 最差的情况下仅在 85 个差异中完成定位, 使得需要专家确认的函数量大幅度下降, 增强漏洞定位的自动化程度。

在结果准确性方面, MDiff 比 Bindiff 多准确定位 1 个漏洞, 比 Diaphora 多准确定位 3 个漏洞, 在大大降低研究人员工作量的同时, 提升了漏洞定位的准确率。

4.3 模块分解结果

在模块分解方面, 我们对目标设备 1、2、3、4、5、6 的模块分解结果总结如表 6 所示。其中各单体式可执行文件大小、文件内函数个数、识别的所有模块入口函数个数如表格第 2、3、4 列所示。CVE 描述相关的目标模块名如第 5 列所示。目标模块通过调用图获得的函数个数、通过局部性获得的函数个数如表格第 6、7 列所示。第 8 列计算了初始模块集合与局部模块集合的并集, 通过与第 9 列的公共模块函数集合求差集, 可以得到最终的目标模块函数个数, 如第 10 列所示。最后, 第 11 列计算了目标模块函数占文件内所有函数的比例。

表 6 具体模块拆分情况

Table 6 Specific module decomposition results

序号	单体式可执行文件大小(MB)	单体式可执行文件函数集合	单体式可执行文件入口函数集合	目标模块名	初始模块函数集合 A	局部模块函数集合 B	初始与局部模块并集	公共模块函数集合 C	目标模块函数集合 T	目标模块函数占比(%)
1				snmp	3368	3585	6246	2706	3540	4.5
2	32.6	79005	423	cmp	2872	656	3332	2706	626	0.8
3				smi	3736	1793	5061	2706	2355	3.0
4				http	19262	3311	20805	2413	18392	32.2
5	43.4	57053	390	ike	20108	3241	22270	2413	19857	34.8
6				snmp	3087	847	3592	2413	1179	2.1
7				webvpn	19210	2515	20401	2413	17988	31.5
8	47.7	50208	369	cifs	17373	2795	18057	2564	15493	30.8
9	49.7	91107	130	sslvpn	20857	2349	22118	19439	2679	2.9
10										
11	34	43271	78	ssh	4965	1351	5874	3479	2395	5.5
12	36.3	86375	485	lldp	1741	495	2235	1740	495	0.6
13										
14	36.3	86375	485	dhcp	3271	2445	5256	1740	3516	4.1
15										

(注: 目标模块函数集合 T =(A∪B)–C, 即 T= $\{x| x\in A$ 或 $x\in B$, 但 $x\notin C\}$))

从表 6 的最终结果, 即最终目标函数所占比例来看, 模块分解达到了良好的效果, 通过模块入口函数与局部性补充方法, 将待分析函数数量平均降低到原本问题规模的 11%, 最差情况下降低到原本问题规模的 35%, 在后续的分析过程中效果明显。

从漏洞定位结果来看, 存在部分漏洞发生漏报的情况, 如序号 1 漏洞对应的 snmp 模块的漏洞函数远离识别的特定模块入口函数局部, 不完全符合高内聚低耦合的特点, 同时为间接调用, 从而在模块分解过程中发生漏报。

为了更准确衡量模块函数漏报误报情况, 我们根据函数引用字符等信息对模块函数进行了人工标记, 通过对结果进行定量分析发现, 基于模块识别的粗粒度定位方法所识别的模块函数, 平均能涵盖 69%的标记模块函数, 最好能况下, 如 smi 能涵盖 100%的标记模块函数, 但是也存在少数较差情况, 如 ssh 模块仅能涵盖 39%的标记模块函数, cmp 模块仅能涵盖 42%的标记模块函数。我们具体探究了其中的原因发现, 除了某些命名质量较差的字符信息会使标记结果存在误差外, 如果目标模块与其他模块耦合程度高, 即不符合高内聚低耦合的特点, 也会使得漏报率提升。因此, 如何更准确地对目标模块

函数进行拆分和衡量, 如引入更多的实体关系, 是未来的一个研究方向。

最后, 我们对更多型号的固件进行了拆解, 以验证模块分解方法对于不同品牌、不同大小固件的适用性。我们对厂商 1～5 的主流设备进行了模块识别与代码拆分, 结果如表 7 所示。结果表明, 模块分解方法对于主流厂商固件都能够有效完成识别和拆分, 体现出对不同品牌的良好适用性。在不同固件规模方面, 如表 7 所示, 固件平均大小为 176MB, 包含 22 万个函数, 内部含有 560 个入口函数。同时固件越大, 内部包含的功能越多, 我们所识别的子系统入口函数也越多, 验证了对大固件具有很好的扩展性。以最大 513.3MB 的厂商 1 固件为例, 可以识别出 1315 个入口函数, 平均每个入口函数覆盖的局部函数数量为 850, 平均每个入口函数通过调用图识别的函数数量为 23178, 同时在模块代码提取过程中, 单个入口函数进行调用图分析平均仅需 50s。显示出模块代码识别的可扩展性。

表 7　不同品牌及不同大小文件的扩展性

Table 7　Scalability of files of different brands and sizes

设备品牌	单体式可执行文件大小(MB)	单体式可执行文件函数个数	单体式可执行文件入口函数数量
厂商 1	33.7	79298	429
厂商 1	441.7	485795	1145
厂商 1	59	87842	391
厂商 1	305.5	381219	1123
厂商 1	513.3	559717	1315
厂商 2	34	44443	94
厂商 3	144.4	240664	232
厂商 4	35.5	68043	115*
厂商 5	24.7	39890	203

(注: *的模式和之前略有区别)

虽然我们给出的子系统入口模式特征十分普适, 守护进程模式作为一种编程实现模式, 几乎存在于所有的单体式设备固件中, 但是进程启动方法上, 不排除极少数系统会采取另类的上层进程创建方式, 这时候不得不在进程函数识别阶段结合一些手工逆向来进行指导。对于厂商 4 固件, 在自动识别守护进程函数后, 我们通过 data 段上存在指引关系的结构体数组找到了所有 115 个进程入口函数及其进程函数名。另外, 关于少数情况下漏洞公告与目标进程子系统名对应困难的问题, 我们通过分析各个模块入口函数附近的局部模块函数引用的字符信息, 将特定目标模块名出现次数在前列的视为目标模块(实验中, 我们将次数排序后的 top3 都纳入分析范围)。例如, 序号 2 漏洞的漏洞公告中显示的模块名为 CMP/Cluster Management Protocol, 但系统内识别的模块名为 Cluster, 后者非缩写但不完整, 通过对各入口函数相关的局部模块函数的分析, 即对 CMP 字符的引用计数和排序, 我们确认了最终的 3 个相关入口函数。

4.4　漏洞定位效果

实验表明, 从漏洞比对的效果来说, 大部分的漏洞都可以通过 MDiff 方法鉴别出。从序号 5、9、14 漏洞可以看出, Bindiff 和 Diaphora 都存在无法正确识别的补丁对, 原因正在于其他模块存在更相似的函数, 内置的启发式算法出现了误匹配。而本文提出的局部性对比方法由于更少和更贴合的对比空间, 给出了正确的结果。另外, 由于局部性对比方法保留了完整的函数调用链, 使得序号 7 漏洞得到了正确的比对结果。序号 1 漏洞虽然在粗细粒度定位中发生漏报, 但依旧受益于基于局部性的二进制代码比对技术, 在比对阶段结果正确。另外, 序号 4、11 漏洞的修复模式都为删除基本块, 我们所提出的基于扩展局部轨迹模拟执行的方法有效识别了这类修补。

同时, 可以发现仍存在部分漏洞无法正常筛选和定位, 如序号 2、10 漏洞, 我们具体探究了其中的原因之后发现, 该厂商修复时候对漏洞函数修改较多, 语法结构变化过大, 导致了漏报。基于局部性的二进制代码比对虽然一定程度上降低了大文件层面的漏报可能, 但如何提升针对这些语法、语义本身变化很大的补丁识别能力是我们下一步需要考虑的问题。另外, 序号 1 漏洞虽然能够正常比对, 但在筛选过程中无法通过调用图和局部补充方法完成提取, 可见在误报消除的过程中也可能引入新的漏报, 如何进一步应对间接调用等问题, 如对间接调用进行更准确的分析, 是未来的研究重点。

接下来我们讨论误报消除阶段各方法的有效性问题。经统计, 除了 4.3 节中的模块级粗粒度定位方法将问题规模平均缩减为原始问题规模的 11%, 函数级细粒度定位方法将问题规模进一步平均缩减为输入规模的 8%, 其中, 平均预处理模块贡献 66%, 安全修补度量贡献 13%, 代码度量贡献 21%。最高的情况下, 安全修补度量贡献 24%, 代码度量贡献 41%。更细致的代码度量方面, C1、C2、V1、V3 效果较为普适, 在大多数情况下都能起到排序作用。其中 V2、V3 效果较为突出, 但 V2 受到目标函数符号的限制, 最终仅在序号 9、10 漏洞中起到作用, 如果能够提前识别更多的库函数, 将会带来更好的结果。

总体上看, 多层次的误报消除方法体现了良好的协同效果。

最后, 我们利用该系统确认了 4 个未公开补丁细节, 包括 LLDP 协议栈溢出漏洞 CVE-2018, LLDP 协议格式化字符串漏洞 CVE-2018, DHCP 协议堆溢出漏洞 CVE-2018, DHCP 协议整数溢出漏洞 CVE-2018。在最终漏洞确认过程中, 主要依赖人工进行 PoC 的构造, 如何进一步提高 PoC 生成阶段的自动化程度是下一步值得研究的问题。

另外, 由于我们针对的是已披露 CVE 的漏洞定位, 需要漏洞公告描述等信息, 然而现实中可能存在厂商没有披露但静默修补的漏洞, 如何针对这部分漏洞进行有效分析, 是未来值得研究和改进的方向。

5 总结与展望

具有单体式操作系统的骨干级设备固件解包后, 待分析的巨型单体式可执行文件令人望而却步。我们认为固件拆分后, 至少有以下三点突出应用: 辅助人工进行目标二进制功能理解与逆向分析; 加速漏洞静态扫描、补丁比对工作, 使得并行化成为可能; 辅助先进的反馈式模糊测试等。

针对缺少漏洞详情且公开漏洞分析较少的问题, 我们首先进行了自动化补丁比对的相关技术研究。相较于原始的直接二进制比对技术, 我们提出了一种基于局部性的二进制代码比对技术, 并通过单体式网络设备固件的内部子进程特性进行了模块代码粗粒度识别, 结合基于动静态语义的细粒度筛选排序方法, 实现了一套针对大型固件版本差异的高效漏洞定位方案, 在对比时间几乎不变的前提下, 大大缩减了待确认的对比结果数量, 且自动化程度较高, 同时借助原型系统验证了多个公开漏洞。

目前的研究中, 我们主要基于局部性原理、函数调用关系、函数数据引用关系等易于提取的实体关系进行模块拆分, 结果趋向保守。下一步工作将尝试研究更多的可行技术来提高模块识别的准确度, 如引入更多的实体关系、进一步解决间接调用问题等。

参考文献

[1] Free sampling of files from the purported Equation Group hack. https://github.com/nneonneo/eqgrp-free-file, Aug.2016.

[2] Liu J, Su P R, Yang M, et al. Software and Cyber Security—A Survey[J]. *Journal of Software*, 2018, 29(1): 42-68.
(刘剑, 苏璞睿, 杨珉, 等. 软件与网络安全研究综述[J]. *软件学报*, 2018, 29(1): 42-68.)

[3] Bindiff. https: //www.zynamics.com/bindiff.html, Oct. 2019.

[4] Diaphora. http://Diaphora.re/, Oct. 2019.

[5] Zhang J, Zhang C, Xuan J F, et al. Recent Progress in Program Analysis[J]. *Journal of Software*, 2019, 30(1): 80-109.
(张健, 张超, 玄跻峰, 等. 程序分析研究进展[J]. *软件学报*, 2019, 30(1): 80-109.)

[6] IDA Pro, https://www.hex-rays.com/products/ida/. Feb. 2020.

[7] Nguyen M H, Binh Nguyen T, Quan T T, et al. A Hybrid Approach for Control Flow Graph Construction from Binary Code[C]. *2013 20th Asia-Pacific Software Engineering Conference*, 2014: 159-164.

[8] Shoshitaishvili Y, Wang R Y, Salls C, et al. SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis[C]. *2016 IEEE Symposium on Security and Privacy*, 2016: 138-157.

[9] L. Xu, F. Sun, Z. Su. Constructing Precise Control Flow Graphs from Binaries. Technical Report. University of California, Davis, 2009:28.

[10] Zhu K L, Lu Y L, Huang H, et al. Construction Approach for Control Flow Graph from Binaries Using Hybrid Analysis[J]. *Journal of Zhejiang University* (*Engineering Science*), 2019, 53(5): 829-836.
(朱凯龙, 陆余良, 黄晖, 等. 基于混合分析的二进制程序控制流图构建方法[J]. *浙江大学学报(工学版)*, 2019, 53(5): 829-836.)

[11] Bao T, Burket J, Woo M, et al. BYTEWEIGHT: Learning to Recognize Functions in Binary Code[C]. *The 23rd USENIX conference on Security Symposium*, 2014: 845-860.

[12] Zeng J Y, Lin Z Q. Towards Automatic Inference of Kernel Object Semantics from Binary Code[C]. *The 18th International Symposium on Research in Attacks*, *Intrusions*, *and Defenses - Volume 9404*, 2015: 538-561.

[13] He J X, Ivanov P, Tsankov P, et al. Debin: Predicting Debug Information in Stripped Binaries[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 1667-1680.

[14] David Y, Alon U, Yahav E. Neural Reverse Engineering of Stripped Binaries Using Augmented Control Flow Graphs[EB/OL]. 2019: arXiv: 1902.09122. https://arxiv.org/abs/1902.09122.

[15] Lacomis J, Yin P C, Schwartz E, et al. DIRE: A Neural Approach to Decompiled Identifier Naming[C]. *2019 34th IEEE/ACM International Conference on Automated Software Engineering*, 2020: 628-639.

[16] Votipka D, Rabin S M, Micinski K, et al. An Observational Investigation of Reverse Engineers' Processes[EB/OL]. 2019: arXiv: 1912.00317. https://arxiv.org/abs/1912.00317.

[17] Anquetil N, Lethbridge T C. Experiments with Clustering as a Software Remodularization Method[C]. *Sixth Working Conference on Reverse Engineering* (*Cat. No.PR00303*), 2002: 235-255.

[18] Mitchell B S. A heuristic search approach to solving the software clustering problem[D]. Drexel University. 2002.

[19] Maletic J I, Marcus A. Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding[C]. *Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence*. *ICTAI*, 2002: 46-53.

[20] Kuhn A, Ducasse S, Gîrba T. Semantic Clustering: Identifying Topics in Source Code[J]. *Information and Software Technology*,

2007, 49(3): 230-243.

[21] Tzerpos V, Holt R C. ACCD: An Algorithm for Comprehension-Driven Clustering[C]. *Proceedings Seventh Working Conference on Reverse Engineering*, 2002: 258-267.

[22] Karande V, Chandra S, Lin Z Q, et al. BCD: Decomposing Binary Code into Components Using Graph-Based Clustering[C]. *The 2018 on Asia Conference on Computer and Communications Security*, 2018: 393-398.

[23] Newman M E J. Fast Algorithm for Detecting Community Structure in Networks[J]. *Physical Review E*, 2004, 69(6): 066133.

[24] Haq I U, Caballero J. A Survey of Binary Code Similarity[J]. *ACM Computing Surveys*, 2022, 54(3): 1-38.

[25] Gao D B, Reiter M K, Song D. BinHunt: Automatically Finding Semantic Differences in Binary Programs[M]. Information and Communications Security. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008: 238-255.

[26] Xu Z Z, Chen B H, Chandramohan M, et al. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills[C]. *2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017: 462-472.

[27] Egele M, Woo M, Chapman P, et al. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components[C]. *The 23rd USENIX conference on Security Symposium*, 2014: 303-317.

[28] Liu B C, Huo W, Zhang C, et al. ADiff: Cross-Version Binary Code Similarity Detection with DNN[C]. *The 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018: 667-678.

[29] Xiao R Q, Liu S L, Yan M, et al. Comparison Technology of Binary Files Based on Hierarchical Nodes[J]. *Computer Engineering and Applications*, 2017, 53(21): 144-150.
(肖睿卿, 刘胜利, 颜猛, 等. 节点层次化的二进制文件比对技术[J]. *计算机工程与应用*, 2017, 53(21): 144-150.)

[30] Andriesse D, Chen X, Van Der Veen V, et al. An In-Depth Analysis of Disassembly on Full-Scale X86/X64 Binaries[C]. *The 25th USENIX Conference on Security Symposium*, 2016: 583-600.

[31] Chandramohan M, Xue Y X, Xu Z Z, et al. BinGo: Cross-Architecture Cross-OS Binary Search[C]. *The 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016: 678-689.

[32] Pewny J, Garmany B, Gawlik R, et al. Cross-Architecture Bug Search in Binary Executables[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 709-724.

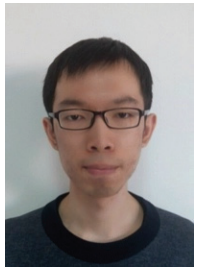
[33] Xue Y X, Xu Z Z, Chandramohan M, et al. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation[J]. *IEEE Transactions on Software Engineering*, 2019, 45(11): 1125-1149.

[34] Du X N, Chen B H, Li Y K, et al. LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics[C]. *2019 IEEE/ACM 41st International Conference on Software Engineering*, 2019: 60-71.

王琛 于 2017 年在哈尔滨工业大学(威海)信息安全专业获得学士学位。现在中国科学院信息工程研究所网络空间安全专业攻读硕士学位。研究领域为网络设备安全。研究兴趣包括: 网络设备的漏洞挖掘与利用等。Email: wangchen@iie.ac.cn

邹燕燕 于 2014 年在中国科学技术大学计算机系统结构专业获得硕士学位。现在中国科学院大学网络空间安全专业攻读博士学位。研究领域为软件安全分析。研究兴趣包括: 漏洞挖掘、模糊测试、程序分析。Email: zouyanyan@iie.ac.cn

刘龙权 于 2018 年在吉林大学网络与信息安全专业获得学士学位。现在中国科学院大学信息工程研究所攻读硕士学位。研究领域为漏洞挖掘、漏洞利用。研究兴趣包括 IoT 安全、漏洞关联。Email: liulongquan@iie.ac.cn



彭跃 于 2017 年在北京科技大学信息安全专业获得学士学位, 现在中国科学院大学计算机技术专业攻读硕士学位。研究领域为程序分析, 漏洞挖掘。研究兴趣包括二进制程序分析, 软件漏洞挖掘。Email: pengyue@iie.ac.cn

张禹 于 2016 年在吉林大学计算机科学与技术专业获得学士学位。现在中国科学院信息工程研究所攻读网络空间安全专业博士学位。研究领域为网络设备漏洞挖掘与利用, 软件模糊测试等。Email: zhangyu2@iie.ac.cn

卢昊良 于 2017 年在吉林大学网络与信息安全专业获得学士学位。现在中国科学院信息工程研究所网络空间安全专业攻读硕士学位。研究领域为软件安全分析。研究兴趣包括: 漏洞挖掘、漏洞利用。Email: luhaoliang@iie.ac.cn

王鹏举 于 2015 年在北京邮电大学电子科学与技术专业获得硕士学位。现任中国科学院信息工程研究所助理研究员。研究领域为网络空间安全。研究兴趣包括: 漏洞挖掘、深度学习。Email: wangpengju@iie.ac.cn

霍玮 于 2010 年在中国科学院计算技术研究所获得博士学位。现任中国科学院信息工程研究所研究员。主要研究领域包括软件漏洞挖掘、利用和安全评测、基于大数据及知识图谱的软件安全分析、信息系统安全分析等。Email: huowei@iie.ac.cn

郭涛 于 2004 年在华中科技大学计算机学院获博士学位。现任中国科学院信息工程研究所研究员。研究领域为软件漏洞分析、安全测评等。Email: guotao@iie.ac.cn