# 基于定义可达性分析的固件漏洞发现技术研究

梅润元[1,2], 王衍豪[3], 李子川[1,2], 彭国军[1,2]

[1]武汉大学 空天信息安全与可信计算教育部重点实验室 武汉 中国 430072
[2]武汉大学 国家网络安全学院 武汉 中国 430072
[3]蔚来 安徽 中国 230031

**摘要** 随着物联网领域的快速发展, 大量物联网设备暴露在互联网中, 存储着文件系统的物联网设备固件却经常被曝出具有安全漏洞, 带来严重安全问题。为应对物联网安全问题, 国内外安全研究者们在自动化漏洞发现方面进行了广泛的研究, 但是现有研究中漏洞发现的误报率与漏报率仍不理想。本文提出了一种基于定义可达性分析的物联网设备固件自动化漏洞发现技术, 基于定义可达性分析方法, 结合函数调用路径分析生成的启发式信息, 设计了一种反向污点跟踪方法, 降低了自动化漏洞发现过程中的误报率。与此同时, 在漏洞的漏报率方面, 本文通过识别用户输入 API 函数的函数调用参数特征对用户输入进行扩充, 降低了漏洞挖掘系统的漏报率, 并通过识别厂商自定义库函数内漏洞的触发点进一步扩大了漏洞的识别范围。基于上述方法, 本文设计并实现了一个自动化漏洞挖掘系统 FirmRD, 经实验测试, 在由来自 Netgear、TP-Link、D-Link、Tenda 四个厂商的 49 款固件组成的对比数据集中, FirmRD 的漏洞识别正确率相较前沿的 SaTC 框架提高了 1.8 倍, 能够生成数量更多的漏洞警报, 且经过人工分析共发现了 4 个中高危的 0-day 漏洞; 在由 6 款 TOTOLINK 固件组成的扩展数据集中, FirmRD 以 82.93%的正确率发现了 68 条正确漏洞警报, 其中 58 条警报与 1-day 漏洞存在关联, 其余 10 条 0-day 漏洞警报中已有 8 条得到了厂商的确认。

**关键词** 物联网设备; 漏洞挖掘; 静态分析; 污点分析; 数据流分析
**中图法分类号** TP309.1;TN915.08 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2025.03.01

# Research on Firmware Vulnerability Discovery Technology Based on Reaching Definition Analysis

MEI Runyuan[1,2], WANG Yanhao[3], LI Zichuan[1,2], PENG Guojun[1,2]

[1] Key Laboratory of Aerospace Information Security and Trust Computing, Ministry of Education, Wuhan University, Wuhan 430072, China
[2] School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China
[3] NIO, Anhui 230031, China

**Abstract** With the rapid development of the Internet of Things(IoT) field, a large number of IoT devices are exposed to the Internet, but the firmware of the IoT devices that stores the file system is often found to have security vulnerabilities, causing serious security problems. In order to deal with the security problems of IoT firmware, security researchers from home and abroad have conducted extensive research on automatic vulnerability discovery, but the false positive rate and false negative rate of existing vulnerability discovery methods are still not ideal. In this paper, we propose an automatic vulnerability discovery technology for IoT firmware based on reaching definition analysis method. Based on reaching definition analysis method, a backward tracing taint analysis method is designed with the help of the heuristic information generated by function call path analysis, and the method can reduce the false positive rates of the vulnerability discovery process. At the same time, in term of reducing the false negative rate of the vulnerability discovery process, we expand the user-input by identifying the parameter characteristics of the function calls of user-input API functions. Furthermore, we expand the scope of vulnerability identification by identifying the trigger points of the vulnerabilities in the vendor-defined library functions. Based on the above methods, we designed and implemented an automatic vulnerability discovery system FirmRD. In the experiments, in a comparative dataset composed of 49 firmware from four manufacturers: Netgear, TP-Link, D-Link, and Tenda, the accuracy rate of the vulnerability discovery method of FirmRD has increased by 1.8 times comparing with the cutting-edge framework SaTC, and FirmRD can discover more vulnerability alerts at the same time. After manual analysis, we found 4 middle-risk or high-risk 0-day vulnerabilities in the comparative dataset. In an extensive dataset composed of 6 TOTOLINK firmware, FirmRD found 68 correct vulnerability alerts with an accuracy rate of 82.93%, 58 of which were related to 1-day vulnerabilities, and 8 of the remaining 10 0-day vulnerability alerts have been confirmed by the manufacturer.

**Key words** Internet of Things devices; vulnerability discovery; static analysis; taint analysis; data flow analysis

---

**通讯作者:** 彭国军, 教授, Email: guojpeng@whu.edu.cn。
本课题得到国家自然科学基金(No. 62172308, No. 61972297, No. 62172144), 网络安全学院学生创新资助计划资助。
收稿日期: 2023-05-29; 修改日期: 2023-08-22; 定稿日期: 2025-01-08

## 1 引言

近年来, 物联网行业快速发展, 物联网设备的普及率越来越高。根据 IoT Analytics 的研究报告统计, 截至 2022 年, 接入互联网的物联网设备数量已增长至约 144 亿台[1]。在物联网设备不断增多的同时, 物联网安全问题也变得愈发重要。为了更有针对性地应对来自物联网的安全威胁, 国内外的安全研究人员们总结了物联网的攻击维度。根据相关文献[2], 针对物联网设备的攻击可以被划分为云、管、端三个攻击维度, 云安全主要关注物联网厂商搭建的云平台的安全性, 管安全主要关注通信协议的安全性, 端安全主要关注终端设备的安全性。

本文所研究的二进制固件漏洞发现技术隶属于物联网端安全的范畴, 固件是存储在物联网终端硬件设备中的软件, 固件漏洞挖掘主要研究物联网终端设备软件系统中的漏洞隐患。

物联网设备的固件中始终存在着严重的安全问题。暴露在互联网上的终端设备数量众多, 并且存在更新滞后的问题, 这些设备一旦遭到漏洞攻击, 将会造成严重后果。例如, 针对 CVE-2021-20090 和 CVE-2021-20091 的在野利用攻击行为已被检测到 20 余万次, 这一在野利用可能影响全球数百万台的路由器设备[3]。

针对这一现状, 国内外安全研究者们在二进制固件漏洞挖掘领域进行了广泛的研究。与传统平台中漏洞发现方法相似, 二进制固件的漏洞发现方法也可以被分为面向执行过程的动态分析方法以及面向二进制程序的静态分析方法两大类别。

在常用的动态分析方法中, 以 Firmadyne[4]为代表的固件模拟执行方案难以模拟物联网设备复杂的外设依赖, 以 FIRM-AFL[5]为代表的模糊测试方案则存在着误报率高的问题, 模拟存在缺陷与分析误报率高成为了固件漏洞挖掘领域动态分析方法的重要缺陷。

另一方面, 静态分析方法在应对上述缺陷方面具有天然优势。静态分析方法可以在不具体运行程序的情况下发现漏洞, 无需进行模拟执行, 并且可以在源代码层面精确地解析漏洞触发路径, 能够解决模糊测试误报率较高的问题。因此, 本文选择从静态分析方法出发进行固件漏洞挖掘研究。

固件漏洞挖掘领域现有静态分析方法主要采用了污点分析与符号执行两种技术。污点分析通过分析外部可控数据的流向寻找潜在漏洞, 是一种通用的漏洞挖掘分析思路, 其处理过程一般包括识别污点源与污点汇聚点、污点传播分析以及污点无害处理[6]。污点源与污点汇聚点分别代表不受信任的用户输入以及危害安全的敏感操作, 是使用污点分析进行漏洞挖掘时的分析起止位置。污点传播分析是对污点数据在程序中流向的分析, 污点无害处理则是污点传播过程中对于不再导致危害的污点数据的消除处理, 是污点传播过程中的 1 个子过程。

污点分析方法在固件漏洞挖掘领域得到了广泛的使用[7-11], 在运用该方法时, 决定漏洞识别范围和漏洞检测类型的是污点源和污点汇聚点的识别, 决定发现效率与准确性的关键因素则是负责寻找漏洞路径的污点传播分析方法的设计。

在现有研究工作中, 研究者们往往倾向于使用符号执行替代传统污点分析方法进行污点传播分析。符号执行方法通过使用符号值代替具体值作为程序的输入静态地进行程序执行, 精确地还原了程序运行流程, 是一种精度很高的分析方法。然而, 符号执行存在路径爆炸的问题[12], 其运行时间往往较长, 导致无法在限定的时间内完成漏洞挖掘任务。在精确性方面, 符号执行方法本身具有较高的精确性, 但其运用于污点传播时需要设计复杂的指令级污点传播规则, 且在物联网领域中由于需要考虑物联网设备的多源异构差异, 设计复杂性进一步提高, 这可能导致更多设计缺陷的产生, 进而影响漏洞挖掘的精确性。

另一方面, 集值分析、定义可达性分析(Reaching definitions analysis)[13]等数据流分析方法也可以被用于污点传播, 但是现有的分析方法往往倾向于将此类数据流分析方法用作剪枝或预处理手段, 而非用作进行污点传播的方法本身。这一倾向过分关注符号执行方法理论上的精确性优势, 忽视了其具体实现上的复杂性和精确性问题。

本文选择使用数据流分析方法中的定义可达性分析方法主导污点传播过程, 经实践与分析证明, 该分析思路可以实现比符号执行主导的污点传播更为简单、高效且精确的漏洞挖掘。定义可达性分析作为数据流分析的一类, 通过静态地确定哪些参数定义可以到达代码中的给定参数使用位置, 可以实现简单、精确且高效的数据流分析。该方法在代码审计与漏洞挖掘领域已经得到了较为广泛的应用[14-15]。因此, 本文运用污点分析技术, 使用静态分析框架 angr[16], 基于定义可达性分析方法设计并实现了一个自动化的漏洞挖掘系统, 本文主要贡献如下:

(1) 针对现有基于符号执行的固件漏洞挖掘工作效率与精确度较低的问题, 本文结合函数调用路

径分析生成的启发式信息, 设计了一种基于定义可达性分析方法的污点分析方法, 实现了高效且精确的漏洞路径发现。

(2) 针对现有研究在漏洞识别范围上的缺陷, 本文通过识别函数调用的参数特征定位程序中负责用户输入处理的应用程序接口(Application programming interface, API)函数, 以及通过将可能触发漏洞的自定义动态链接库函数纳入污点汇聚点的识别范畴, 分别扩大了污点源与污点汇聚点的识别范围。

(3) 基于上述方法设计并实现了一套自动化漏洞挖掘系统 FirmRD, 并在由来自 Netgear、TP-Link、D-Link、Tenda 四个厂商的 49 款设备组成的对比数据集以及由 TOTOLINK 的 6 款设备组成的扩展数据集中分别进行了测试。测试结果表明, 在对比数据集中, FirmRD 的漏洞识别正确率相较前沿的 SaTC[7]框架提高了 1.8 倍, 且能生成数量更多的漏洞警报; 在扩展数据集中, FirmRD 以 82.93%的正确率发现了 68 条正确漏洞警报, 其中 58 条警报与 1-day 漏洞存在关联, 其余 10 条 0-day 漏洞警报中已有 8 条得到了厂商的确认。

## 2 相关工作

污点分析是一种精确的漏洞发现方法, 误报率较低, 但相应的, 在代表危险数据输入的污点源与代表危险利用的污点汇聚点识别不完全, 污点传播路径判断有缺漏时, 也可能存在较高的漏报率。在以 DTaint[8]为代表的框架将污点分析技术引入二进制固件漏洞发现领域后, 国内外的研究者们也分别针对污点源与污点汇聚点的识别以及污点传播的方法做出了自己的改进。

对于污点源与污点汇聚点的识别方法的改进主要目的在于扩大漏洞的识别范围, 从而挖掘到更多的漏洞。在对于污点源的改进方面, Chen 等提出的 SaTC[7]框架通过扫描超文本标记语言(HyperText markup language, HTML)、JavaScript 语言(JS)、可扩展标记语言(Extensible markup language, XML)等格式的前端文件中储存的关键字信息, 定位并识别了网络界面与固件中后端程序交互时使用的关键字标识信息, 极大程度地提高了污点源的识别率。

扩大污点汇聚点的识别范围的主要目的是识别更多可能导致漏洞的危险函数, 或是识别更多类型的安全漏洞, 例如, Li Zhang 等提出的 CryptoREX[9]框架通过识别密码学危险函数扩大了污点汇聚点的识别范围, 从而使其框架具备了发现密码学误用类型漏洞的能力。

污点的传播技术是决定污点分析效率的关键技术, 符号执行与数据流分析技术都在这一过程中得到了广泛的应用。运用符号执行进行污点传播的代表性框架有 Nilo Redini 等提出的 KARONTE[10]框架, 其主要贡献在于通过识别污点的跨文件传播提高了污点传播的识别率。

然而, 使用符号执行进行污点传播时通常需要针对不同类型的指令设置污点传播规则以及符号执行约束条件, 从而在发现执行路径之后通过求解约束判断路径的可达性。这一设计过程较为复杂, 且由于物联网设备多源异构, 设计复杂性进一步提高。这可能导致更多设计缺陷的产生, 从而影响漏洞挖掘的精确性。

例如, SaTC[7]框架作者曾在文章中对比了其与 KARONTE[10]的分析结果, 虽然该框架通过扩大污点源识别范围实现了漏洞识别范围的明显扩大, 但是其识别的 2,084 个漏洞警报中只有 683 条警报具备静态分析视角下的真实漏洞路径[7], 总正确率仅有 32.77%。根据其误报情况展示, 该框架污点传播过程中没有考虑 atoi 等类型转换函数对结果的影响[7]。这一误报的消除并不困难, 但是其仍然在一个设计完备的系统之中出现, 体现了基于符号执行的污点传播方法精度可能受制于复杂系统的设计缺陷的问题。在该次对比中, KARONTE[10]的分析结果中只包含 74 条报警, 但正确率也仅有 62.16%[7], 这说明其分析精确程度也仍然具备提升空间。

另一方面, 数据流分析方法也可以被用于污点传播, 漏洞挖掘领域也存在着基于集值分析、定义可达性分析等数据流分析方法的现有工作[11, 15, 17]。

现有的研究往往倾向于将数据流分析方法用作剪枝或预处理手段, 而非用作进行污点传播的方法本身, 例如 SelectiveTaint 框架[11]在运用集值分析剔除污点传播时不需要处理的指令集合之后通过二进制重写技术写入污点分析指令以进行污点传播, HEAPSTER[17]框架运用定义可达性分析方法检测作为潜在堆分配器的指针源并通过符号执行方法进行堆漏洞检测。

另一方面, 由数据流分析方法主导污点传播过程的分析方法设计复杂性往往更低, 如果设计得当, 此类方法也可以进行高精度的漏洞发现。因此, 本文基于数据流分析中的定义可达性分析方法进行污点传播, 结合函数调用路径分析生成的启发式信息对函数间的反向污点传播过程进行了优化, 通过综合考虑各类参数定义情况进行了细粒度的数据流解析, 设计并实现了一种简单, 精确且高效的漏洞路径发

现方法。

## 3 基于定义可达性分析的污点分析方法

### 3.1 定义可达性分析的基本原理

在程序分析理论中，特定语句的可达定义(Reaching definition)为一条语句，其目标变量可以无需中间赋值地到达或分配给目标语句。定义可达性分析是一种决定程序中何处的语句定义了待考虑的数据使用位置读取的值的数据流分析方法[13]，在定义可达性分析过程中，我们通过对特定目标语句(即程序中危险语句)所使用的变量的值的直接定义位置进行查找，从而跟踪危险数据的数据流情况。

图 1 通过本文发现的一个 0-day 漏洞 CVE-2022-30078 展示了定义可达性分析的一个示例，该漏洞通过网络报文向非易失性随机访问存储器(Non-Volatile random access memory, NVRAM)外设中存储可被攻击者控制的数据，再通过 acosNvramConfig_get 函数读取外设设备中可被攻击者影响的变量的值，将其传入 sprintf 后进而传入 system 函数，导致命令注入漏洞产生。

```
1 int __fastcall sub_19884(const char *a1, const char *a2, const char *a3, const char *a4,
2 {
3   const char *v10; // r0
4   int v11; // r5
5   unsigned int v12; // r0
6   int v13; // r0
7   int v14; // r0
8   int v15; // r0
9   int v17; // r0
10  int v18; // r0
11  const char *v19; // r0
12  unsigned int v20; // r0
13  int v21; // r0
14  unsigned int v22; // r0
15  char v23[512]; // [sp+8h] [bp-340h] BYREF
16  char dest[128]; // [sp+208h] [bp-140h] BYREF
17  char s[128]; // [sp+288h] [bp-C0h] BYREF
18  char v26[64]; // [sp+308h] [bp-40h] BYREF
19
20  if ( strcmp(a1, "6to4") && strcmp(a1, "autoconfig") )
21  {
22    sprintf(v23, "ifconfig %s add %s/%s", a2, a3, a4);
23    system(v23);
119   if ( strcmp(v22, "6to4") )
120   {
121     if ( !strcmp(v22, "fixed") )
122     {
123       v13 = (const char *)acosNvramConfig_get("ipv6_wan_ipaddr");
124       v14 = (const char *)acosNvramConfig_get("ipv6_wan_length");
125       v15 = (const char *)acosNvramConfig_get("ipv6_lan_ipaddr");
126       v16 = (const char *)acosNvramConfig_get("ipv6_lan_length");
127       sub_19884("fixed", v21, v13, v14, v15, v16);
128     }
```

**图 1 定义可达性分析示例**

**Figure 1 An example of reaching definition analysis**

图 1 中的箭头符号展示了定义可达性分析依次进行查找的过程，这种从数据使用位置到定义位置的查找过程是一种反向递归式的分析过程，其每次查找只寻找能无需中间赋值地影响目标语句的定义位置。通过在特定函数的上层调用位置继续查找，可以进一步地实现跨函数的数据流跟踪。

除了对变量的使用与定义位置间的数据流依赖关系进行分析外，定义可达性解析过程还可以推断变量定义位置所指示的变量类型，如图 1 中 sprintf 函数的第三至五个参数来源于函数输入参数，而其上层调用者中第 127 行调用语句的 *v13* 参数则来源于 acosNvramConfig_get 函数的返回值。通过对不同的定义类型进行分析，我们可以在分析过程中解析例如拷贝函数长度限制的严谨性的，对漏洞路径可达性有重要影响的相关变量定义情况，进而对各类漏洞路径不可达情况进行筛除，提高分析的准确性。

### 3.2 污点分析方法的总体设计

基于定义可达性分析的污点分析方法是本文实现简单、精确且高效的漏洞路径发现的核心，是由定义可达性分析这一数据流分析方法主导污点传播的一种高精度污点分析方法。根据定义可达性分析的相关原理，该方法从可能导致漏洞的危险函数的危险参数出发反向遍历程序路径，通过分析数据流依赖关系进行污点分析，进而进行漏洞路径发现。在分析过程中，为了防止路径爆炸，本方法运用基于深度优先的函数调用路径分析为定义可达性分析提供了启发式信息，进而使用定义可达性方法进行了基于数据流依赖关系的反向污点跟踪，污点分析方法的总体设计结构如图 2 所示:

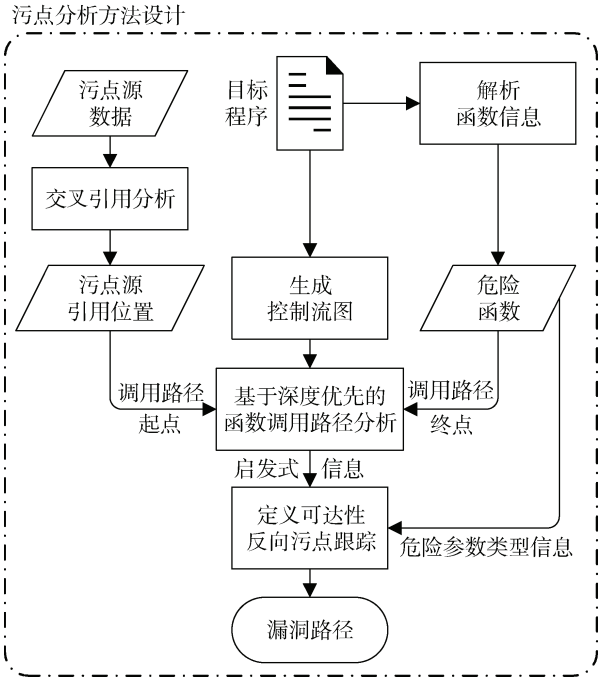


**图 2 污点分析方法的总体设计**

**Figure 2 The overall design of the taint analysis method**

### 3.3 基于深度优先的函数调用路径分析

函数调用路径分析为定义可达性查找提供了反向递归过程中的启发式信息，是本文污点分析方法设计中的一个预处理阶段。

由于定义可达性分析是一种从参数使用位置到定义位置的递归向上定义查找，如果不限制作用范围，该分析可能由于递归层数过深出现路径爆炸的问题。如果在不进行预处理的情况下将定义可达性

分析扩展到程序范围, 则当需要查找某个函数输入参数的定义位置时, 因为无法判定位于何处的上层调用位置可能触及用户输入, 程序必须遍历当前函数的所有调用位置, 并在所有上层调用位置处分别进行进一步地定义可达性查找, 这造成了很多不必要的路径查找。

因此, 对于每一个污点源引用位置, 本方法将通过查找函数间调用关系搜索该引用位置所在函数所调用的所有子函数, 并以深度优先的方式在这些子函数中再次进行函数调用路径分析, 从而找到一组从污点源到污点汇聚点函数的函数调用路径, 在定义可达性分析寻找函数上层调用位置时为其提供决定向何处上层调用位置分析的启发式信息。

诚然, 该启发式信息也可以在执行定义可达性分析之前, 通过反向递归分析获取。但在递归分析的入口方面, 作为正向递归入口的用户输入位置, 相较于作为反向递归入口的危险函数(包括 system,strcpy 等常用函数)所有程序引用位置而言相对较少, 更加适合作为递归起点。因此, 我们选择使用正向递归方法进行函数调用路径分析。

在上述查找过程的前提条件方面, 对于子函数调用的深度优先查找依赖程序控制流图中对函数的后继节点的描述; 污点源数据的查找(将在 4.2 节中详细介绍)以提示用户输入信息位置的关键字字符串为媒介, 因而需要通过交叉引用查找定位位于数据节的关键字的程序引用位置, 再在这些污点源引用位置处进行分析; 作为分析终点的危险函数的查找则需要对程序中的函数信息进行解析, 进而通过预先规定需要关注的危险函数名称列表查找对应函数的程序链接表(Procedure Link Table, PLT)调用位置实现。

由于需要通过库函数的动态加载机制加载, 溢出、命令注入等常见类型的危险函数符号无法去除且其命名需符合动态链接库的命名规范, 因此不同于符号信息可能去除的污点源信息, 危险函数信息的查找可以通过字符串方式进行匹配。该方法的局限在于, 对于使用非常见架构或非常见库函数的厂商, 需要根据对应的情况更新危险函数列表信息。

函数调用路径分析算法如算法 1 所示, 其中调用定义可达性分析方法后的具体处理将在 3.4 节中描述, 算法需要使用的从用户输入位置到危险函数使用位置的函数调用路径启发式信息 *sink_path* 通过在第 20 行进行深度优先递归时在原有路径上拼接后继的调用节点获取, 对于污点汇聚点扩充模块的调用则将在 4.3 节中进行描述。

**算法 1.** 基于深度优先的函数调用路径分析算法

输入: 污点源引用位置 *source_refaddr*、危险函数列表 *SINKS*

输出: 潜在漏洞路径

**过程 1.** *xref_preprocessing*(*source_refaddr*)

```
1:  //查找污点源引用位置所在函数
2:  start_func = search_start_func(source_refaddr)
3:  //深度优先查找函数调用路径
4:  sink_reached = dfs(start_func, sink_path)
```

**过程 2.** *dfs*(*func, sink_path*) //*sink_path* 存储从起始函数到当前函数的调用路径, 过程返回值表示是否查到了危险函数引用

```
1:  IF func.name in SINKS AND len(sink_path) THEN
2:      //触及危险函数, 调用定义可达性分析方法
3:      sinked,depth = Use_Define_Checker(sink_path)
4:  IF sinked THEN output(sink_path)
5:  WHILE depth>=0
6:      IF sink_path[depth] not in sink_reachable THEN
7:          sink_reachable.append(sink_path[depth] )
8:      depth -= 1
9:      RETURN True
10:  IF func is a library function THEN
11:      //触及库函数, 调用污点汇聚点扩充模块
12:      call Lib_Function_Summarizer
13:  IF func in safe_funcs AND func != start_func THEN
14:      RETURN False //局部安全函数剪枝
15:  FOR call_site in func.basic_block.call_sites
16:      //查找当前函数调用的所有函数, 递归调用 dfs
17:      callee_func = get_successor_func(callee)
18:      IF callee_func in sink_path THEN
19:          CONTINUE //防止死循环
20:      sink_reached = dfs(callee_func, sink_path + [callee_func] ) OR sink_reached
21:  IF not sink_reached AND func not in safe_funcs
22:  THEN
23:      safe_funcs.add(func) //第一类局部安全函数剪枝
24:  IF func not in sink_reachable AND func!= startFunc
25:  THEN
26:      safe_funcs.add(func) //第二类局部安全函数剪枝
27:  RETURN sink_reached
```

在深度优先搜索的过程之中, 本方案将记录已分析过的函数的漏洞可达状态信息, 通过记录无危险子函数的函数和危险子函数不可达的函数这两类局部安全函数进行剪枝, 以此避免对于单一函数的重复分析。

具体而言, 如果某一函数不存在任何到达危险函数的调用路径, 则 *dfs* 过程第 15 行的循环结束后, 由

第 20 行返回的过程返回值 *sink_reached* 仍然为 False, 此时, 目标函数没有危险子函数, 第 23 行判断该函数为第一类局部安全函数, 进而对该函数进行剪枝。

另一方面, 如果某一函数可能调用的所有危险函数的危险参数定义来源都不受到该函数输入参数控制, 则对此函数参数的控制也不会导致漏洞产生。该情况对应了本算法第二类局部安全函数剪枝。具体而言, *dfs* 过程第 6-9 行将在 *sink_reachable* 列表中记录定义可达性分析方法的结果显示的输入参数可影响到危险函数的所有函数。在第 26 行处, 如果在分析完某一函数调用的所有函数后, 仍未发现该函数输入参数对任何危险函数的影响, 则目标函数危险子函数不可达, 第 26 行将其判断为第二类局部安全函数, 进而对该函数进行剪枝。

### 3.4 定义可达性反向污点跟踪

定义可达性反向污点跟踪方法是本文实现高精度漏洞发现技术的核心, 是通过跟踪数据流依赖关系检测漏洞路径的一种解决方案。本解决方案通过解析危险函数参数的定义位置分析其是否可能被敏感输入信息控制实现漏洞路径的发现。在解析参数定义位置时, 本方案通过区分不同类型参数定义实现了细粒度的数据流解析, 并通过各类参数定义的递归解析实现了跨函数的, 从参数使用位置到定义位置的反向污点跟踪算法。

从实现原理出发, 首先, 本方案的实现基础是定义可达性分析对于特定参数定义位置的解析操作。图 3 展示了定义可达性分析的参数定义位置解析原理, 参数定义位置解析主要通过在基本块内查找参数对应寄存器或内存位置的定义位置实现, 若基本块内未找到定义, 则需要在基本块前驱节点内继续尝试查找。

因此, 参数定义的查找过程中可以正常解析分支与循环中的定义情况。对于循环而言, 即使参数定义位置位于使用位置后方, 由于循环体中最后一个基本块的前驱节点指向循环起点, 因此寻找前驱节点过程中程序能够正确解析循环情况, 判断位于使用位置后方的定义位置可以影响到目标使用位置。

对于分支情况而言, 如果某一参数定义赋值位于分支之中, 即该基本块的前驱基本块不唯一, 则所有前驱基本块内的定义语句都有可能影响该处参数使用位置, 因此所有分支中的定义情况都需要进行输出。

基于上述参数定义位置解析原理, 本文的反向污点跟踪方案运用定义可达性分析方法, 通过分析作为污点汇聚点的危险函数的参数定义位置反向查找污点路径, 直到发现其可能被作为污点源的危险输入影响为止。本方案对各类参数定义情况进行了细粒度的区分与解析, 方案运行流程如图 4 所示。

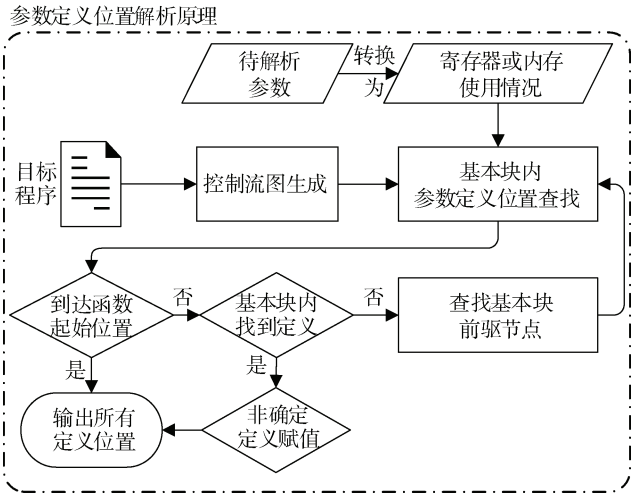


**图 3 参数定义位置解析原理**

**Figure 3 The principle of parameter definition position analysis**

由于定义可达性分析是一种跟踪单一变量定义位置的分析方式, 在用于污点传播时需要从作为污点汇聚点的危险函数危险参数出发, 递归向上跟踪危险参数定义位置, 直到该参数的定义来源触及污点源, 或被判定为不可能到达污点源为止。当以下 2 种情况发生时, 本方案将判断危险参数无法到达污点源, 从而停止递归过程:

(1) 常量、未初始化局部变量或涉及类型转换的定义: 当危险参数内容来源于常量或未初始化局部变量时, 其内容来源将无法被其他变量影响; 当危险参数定义来源于 atoi 等类型转换函数的返回值时, 由于数据类型受限, 其将无法导致缓冲区溢出或命令注入漏洞。

(2) 作用域范围越界: 在污点源的变量作用域范围内无法定位到任何污点源对于目标参数的影响语句。由于单一变量只能在其作用域范围内使用, 一旦作用域范围越界, 则污点源变量不可能再对目标参数产生影响, 此时应当停止继续分析。

在参数定义位置的解析过程中, 定义可达性分析方法可能遇到多种类型的参数定义情况。如图 4 所示, 在解析参数定义位置时, 参数定义的解析结果将会分为以下 6 类, 本方案将对不同类型的解析结果分别进行下述处理:

(1) 常量或未初始化局部变量定义: 该类定义位置无法被污点源影响, 因此可以直接舍弃;

(2) 局部变量定义: 由于函数内部的局部变量的值一般情况下均来自于常量或函数输入参数, 因此, 局部变量定义需要继续递归解析其定义可达性;

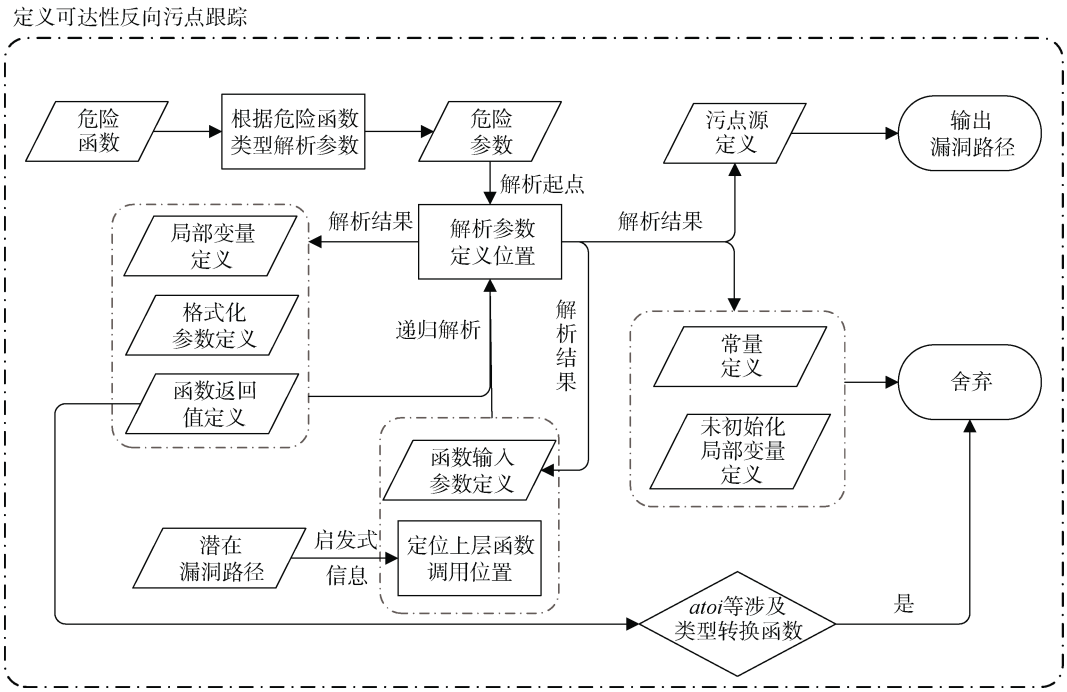


**图 4　定义可达性反向污点跟踪**

**Figure 4　Reaching definition backward taint tracking**

(3) 函数返回值定义: 如果参数的定义来源于某一其他函数的返回结果, 则该定义来源的值可能受到对应函数的所有参数的控制与影响。为了提高分析的覆盖率, 本方案在该情况下将函数所有参数都标记为了可以影响目标参数的上层定义来源, 在此基础上继续递归解析这些参数的定义可达性。如果带长度限制的字符串拷贝函数的长度限制参数来源于源缓冲区长度, 则判断该长度限制失效。如果对应函数涉及字符串至整数、字符串至 IP 地址结构体等类型转换, 则字符串无法继续导致溢出或命令注入漏洞, 此时应当直接舍弃该次定义;

(4) 格式化参数定义: 对于存在格式化字符串参数的危险函数而言, 其输入参数个数往往是可变的。因此, 需要通过查找其格式化字符串参数的定义情况确认该函数可能导致漏洞的危险参数定义位置。具体而言, 若格式化字符串参数为变量定义, 则判断其是否可能被污点源影响, 若为常量定义, 则为检测缓冲区溢出类型漏洞, 需根据常量字符串中的*%s* 字符串参数的个数与位置, 将对应位置的其他输入参数纳入待检测的危险参数列表进行递归解析处理;

(5) 函数输入参数定义: 如果目标参数可能被其所在函数的输入参数影响, 则需要根据潜在漏洞路径所提示的函数上层调用者信息, 在上层函数中查找调用者函数对应输入参数的定义可达性, 在上层函数内部继续递归进行定义可达性解析;

(6) 污点源定义: 如果目标参数的定义位置来源于作为污点源的用户输入 API 函数的返回值, 则污点源与污点汇聚点间存在控制流与数据流依赖关系, 污点源可能影响危险函数的执行结果, 因此, 将当前路径判断为漏洞路径并进行输出。

在剪枝优化方面, 为避免对同一个危险函数引用位置的定义可达性进行重复分析, 本方案在实现时对目标位置引用情况进行了相应的记录, 运用了动态规划的思想, 在涉及相同引用的情况下通过加载先前存储的运行结果实现了剪枝优化, 提高了运行效率。

此外, 对于位于同一函数内部的多处污点源引用情况, 本方案将使用延迟分析的方法, 集中此类引用情况并在针对其所在函数的单次分析之中一次性解析所有引用情况的定义可达性, 从而提高运行效率。

## 4　漏洞挖掘系统 FirmRD 的设计与实现

### 4.1　系统架构

通过将第 3 章所述的基于定义可达性分析的污点分析方法组装为污点分析模块, 进而增加固件预处理、前端分析与后端分析过程中的其他模块, 本文设计并实现了一个自动化的漏洞挖掘系统 FirmRD, 并使用静态分析框架 angr[16]对系统进行了实现, 系统整体架构如图 5 所示。

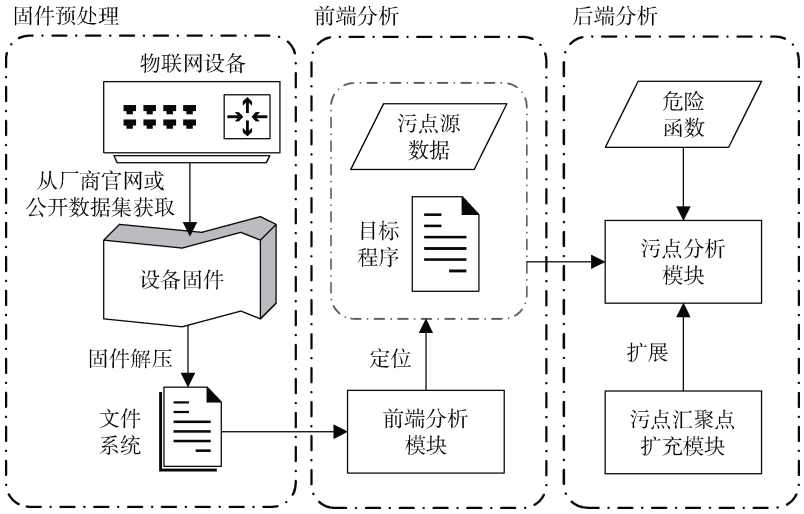


**图 5　FirmRD 系统架构**

**Figure 5　The system structure of FirmRD**

在固件预处理阶段中, 系统需要从物联网设备中获取固件文件系统。FirmRD 从物联网设备厂商官网以及现有漏洞挖掘研究提供的公开数据集中获取物联网设备固件, 并使用 binwalk[18]工具解压获取了固件文件系统。

在前端分析阶段中, 系统需要在提取固件后查找容易受到攻击的目标二进制程序作为分析目标, 并通过标记与用户输入相关的关键字与 API 函数确定污点源的位置。FirmRD 的前端分析模块负责了分析目标与污点源的选取工作, 该模块选取用于处理网络交互的网络边际程序作为分析目标, 将用户输入 API 函数作为污点源, 并通过识别函数调用的参数特征扩大了污点源的识别范围。

在后端分析阶段中, 系统需要根据待检测的危险函数的信息在目标程序中定位污点汇聚点, 并通过识别污点数据从污点源到污点汇聚点的传播过程对漏洞路径进行发现。FirmRD 使用基于定义可达性的污点分析方法完成了这一漏洞路径发现过程, 并通过污点汇聚点扩充模块扩充了污点汇聚点的识别范围。

在组成 FirmRD 的三个分析模块中, 污点分析模块中使用的方法已在第 3 章中进行了论述。因此, 4.2 节与 4.3 节将分别论述前端分析模块以及污点汇聚点扩充模块的设计方法, 介绍前者对于污点源的扩充方法以及后者对于污点汇聚点的扩充方法。在此之后, 4.4 节将对系统的实现方法进行论述。

## 4.2　前端分析模块设计

FirmRD 的前端分析模块基本沿用了 SaTC[7]开源框架的分析方案。该方案通过提取 HTML、JS、XML 等格式的前端文件中与发送网络报文相关的关键字信息, 进而通过查找二进制文件中对于对应关键字的引用情况定位作为分析目标的网络边际程序以及其中的污点源。

然而, 虽然 SaTC[7]的分析方案相较基于硬编码关键字的传统污点源识别方案而言已极大地扩大了污点源的识别范围, 但其污点源识别仍然存在部分遗漏。该框架使用隐式入口查找器(Implicit entry finder), 通过将上下文中具有相似代码模式的函数调用纳入分析, 一定程度上缓解了污点源遗漏的问题[7], 但其缓解程度仍不显著。

图 6 和图 7 展示了 SaTC[7]隐式入口查找器可以解决以及无法解决的污点源遗漏的一个示例, 两段代码分别对应了 Tenda AC18 中各一个 1-day 漏洞, 方框标识语句均为 SaTC[7]前端关键字查找方案所遗漏, 且可触发漏洞的, 作为污点源的用户输入。

```
1  int __fastcall formSetSambaConf(int a1)
2  {
3    int result; // r0
4    int v2; // r0
5    char s[64]; // [sp+14h] [bp-70h] BYREF
6    int v5; // [sp+54h] [bp-30h]
7    int v6; // [sp+58h] [bp-2Ch]
8    int v7; // [sp+5Ch] [bp-28h]
9    int v8; // [sp+60h] [bp-24h]
10   const char *v9; // [sp+64h] [bp-20h]
11   char *s1; // [sp+68h] [bp-1Ch]
12   int v11; // [sp+6Ch] [bp-18h]
13   int v12; // [sp+70h] [bp-14h]
14   int v13; // [sp+74h] [bp-10h]
15
16   memset(s, 0, sizeof(s));
17   v13 = sub_28884(a1, "password", "admin");
18   v12 = sub_28884(a1, "premitEn", "0");
19   v11 = sub_28884(a1, "internetPort", "21");
20   s1 = (char *)sub_28884(a1, "action", &unk_F1338);
21   v9 = (const char *)sub_28884(a1, "usbName", &unk_F1338);
22   v8 = sub_28884(a1, "guestpwd", &unk_F1338);
23   v7 = sub_28884(a1, "guestuser", &unk_F1338);
24   v6 = sub_28884(a1, "guestaccess", &unk_F1338);
25   v5 = sub_28884(a1, "fileCode", &unk_F1338);
26   if ( !strcmp(s1, "del") )
27   {
28     doSystemCmd("cfm post netctrl %d?op=%d,string_info=%s", 51, 3, v9);
```

**图 6　Tenda AC18 污点源识别示例 1**

**Figure 6　The first taint source identification example from Tenda AC18**

在图 6 中, 由于 17 至 25 行的函数上下文中存在多处可被 SaTC[7]识别的, 具有相似代码模式的污点源输入函数调用, SaTC[7]隐式入口查找器将该处调用纳入分析, 解决了遗漏问题。然而, 在图 7 中, 由于函数上下文中的所有用户输入语句对应的关键字都无法被 SaTC[7]识别, 隐式入口查找器无法将此遗漏纳入分析。

```
1  int __fastcall fromDhcpListClient(int a1)
2  {
3    int v1; // r0
4    int v2; // r0
5    int v5[4]; // [sp+10h] [bp-36Ch] BYREF
6    char v6; // [sp+20h] [bp-35Ch]
7    char s[64]; // [sp+120h] [bp-25Ch] BYREF
8    char dest[256]; // [sp+160h] [bp-21Ch] BYREF
9    char v9[256]; // [sp+260h] [bp-11Ch] BYREF
10   int v10; // [sp+360h] [bp-1Ch]
11   const char *v11; // [sp+364h] [bp-18h]
12   char *nptr; // [sp+368h] [bp-14h]
13   int i; // [sp+36Ch] [bp-10h]
14
15   i = 0;
16   memset(s, 0, sizeof(s));
17   nptr = (char *)sub_2B884(a1, "LISTLEN", "0");
18   v11 = (const char *)sub_2B884(a1, "page", "1");
19   v6 = 0;
20   for ( i = 1; ; ++i )
21   {
22     v1 = atoi(nptr);
23     if ( v1 < i )
24       break;
25     v5[0] = 0;
26     v5[1] = 0;
27     v5[2] = 0;
28     v5[3] = 0;
29     sprintf((char *)v5, "%s%d", "list", i);
30     v10 = sub_2B884(a1, v5, &unk_ED730);
31     if ( !v10 || !*(_BYTE *)v10 )
32       break;
33     strcpy(dest, (const char *)(v10 + 1));
34     dest[strlen(dest) - 1] = 0;
35     sprintf(s, "dhcps.Staticip%d", i);
36     SetValue(s, dest);
37   }
38   SetValue("dhcps.Staticnum", nptr);
39   v2 = sprintf(v9, "/network/lan_dhcp_static.asp?page=%s", v11);
```

**图 7　Tenda AC18 污点源识别示例 2**

**Figure 7　The second taint source identification example from Tenda AC18**

因此, FirmRD 通过将图 7 类型的用户输入 API 引用位置正确识别为污点源, 解决了 SaTC[7]的污点源遗漏问题, 对污点源进行了扩充。

污点源扩充算法的关键在于全面识别用户输入 API 函数, 进而通过将用户输入 API 函数的所有引用位置视为污点源实现用户输入位置的全面识别。该识别算法的思想在于: 除通用字符串处理、文件处理以及日志与调试信息输出函数外, 调用多种不同类型关键字的函数通常只有和用户输入或输出相关的 API 函数。

除通用字符串处理、文件处理以及日志与调试信息输出函数外使用字符串关键字的一般只有业务逻辑处理函数以及用户输入与输出 API 函数。处理具体业务逻辑的函数不会使用相同方法处理被“*username*”、“*ipaddress*”等不同类型关键字标识的参数, 因为这些参数的数据类型往往存在较大区别。但是用户输入 API 函数 WebsGetVar 会从多种类型关键字中以字符串形式获取输入内容, 网络处理程序响应报文拼接函数等输出函数也会引用多种类型的关键字信息。

因此, 我们首先通过函数名匹配方法筛除作为常用库函数, 因链接需要保留了符号, 且命名方式相对固定的字符串处理、文件处理以及日志与调试信息输出函数。在此之后, 如果某一函数使用同一参数位置处理被多种关键字标识的参数, 处理的不同类型参数数量超过某一阈值, 我们就将该函数视为一种用户输入 API 函数, 进而将其识别为污点源。

诚然, 该识别方法可能将输出函数, 例如网络响应报文生成函数, 识别为污点源。但是另一方面, 输出函数的参数与返回结果一般也不会在程序中进行进一步处理, 例如网络处理程序拼接完相应报文后, 一般会直接发送这一报文, 不对其进行额外处理。因此, 即使将此类函数调用识别为污点源, 此类函数与危险函数间也不存在数据流依赖, 此类误识别也就不会在后续漏洞发现过程中生成误报。

基于上述思想, 结合具体实践与分析, 我们基于 SaTC[7]框架筛选出的数量丰富的网络服务关键字识别了网络服务关键字引用位置的函数调用参数特征。通过以上特征, 系统筛选出了在同一参数位置引用了多个关键字的函数。

用户 API 发现算法如算法 2 所示, 算法将首先查找网络服务关键字的全部程序引用位置, 由于对网络服务关键字的处理一般都有函数封装, 第 10 行剔除了不以函数参数形式引用的关键字信息。在第 12 行筛除字符串处理、文件处理以及日志与调试信息输出函数后, 算法用 *API_count_dict* 字典记录用 *callee_addr* 标识的每个引用了网络服务关键字的函数在特定参数位置 *reg_name* 引用关键字的数量, 若某一函数位于同一参数位置的关键字引用次数大于某一特定阈值, 系统就在第 27 行将对应函数推测为用户输入 API, 并将其纳入 *API_set*, 进而通过分析其所有引用位置对污点源进行扩充。

**算法 2.** 用户输入 API 发现算法

输入: 网络服务关键字 *key_addr*
输出: 用户输入 API
1:BEGIN
2:　*API_set* = set()
3:　*API_count_dict* = dict()
4:　FOR each *key_addr*
5:　　//查找网络服务关键字的全部程序引用位置
6:　　*xrefs* = *get_xrefs_by_dst*(*key_addr*)
7:　　FOR *xref* in *xrefs*

```
8:          //获取指令所在基本块后继节点的函数名
9:          callee_name = successor_name(xref.ins_addr)
10:         IF callee_name is None OR callee_name is Unresolved THEN //后继节点非函数, 引用非函数参数
11:             CONTINUE
12:         IF callee_name contains func_filter_keywords
13:         THEN //该处函数调用为字符串处理、文件处理以及日志与调试信息输出函数调用
14:             CONTINUE
15:         //记录传递参数时使用的寄存器
16:         reg_name = check_arg(xref.ins_addr)
17:         callee_addr = successor_addr(xref.ins_addr)
18:         IF callee_addr not in API_count_dict.keys()
19:         THEN
20:          API_count_dict[callee_addr] = (1,reg_name)
21:         ELSE
22:          IF callee_addr_count_dict[callee_addr] [1] == reg_name AND calladdr_count_not_added
23:              THEN //如果多个网络服务关键字使用同一参数位置调用某一函数, 增加计数
24:                  API_count_dict[callee_addr] [0] += 1
25:     IF API_count_dict[callee_addr] [0] > threshold
26:         THEN //若计数超过阈值, 记录 API
27:             API_set.add(callee_addr)
28:   RETURN API_set
29:END
```

经实验测试可知, 筛除三类函数之后本算法能够以较低误报率正确识别网络服务 API, 全面且低误报地对网络边际程序中的污点源信息进行识别。此外, 虽然 SaTC[7]框架的关键字来源都是网络服务的前端文件, 但是此类关键字的引用位置处仍可能出现通过配置文件、设备外设、JSON (JavaScript Object Notation)结构等方式读取信息的 API 函数调用。考虑到以上三类数据仍有可能被攻击者影响, FirmRD 将相关 API 也纳入了分析范畴。

### 4.3 污点汇聚点扩充模块设计

物联网设备架构多样, 标准并不统一, 除了使用常见第三方动态链接库外, 物联网设备厂商也会自行制作厂商自定义的动态链接库程序, 这些程序相较开源动态链接库而言往往缺乏评估与检测, 因此其中也可能存在潜在漏洞。

因此, FirmRD 通过识别此类自定义动态链接库中函数对于危险参数的引用情况, 为此类函数生成了可以评判其危险性的函数摘要, 从而扩展了污点汇聚点的识别范围。该功能由污点汇聚点扩充模块实现, 对应模块结构如图 8 所示。

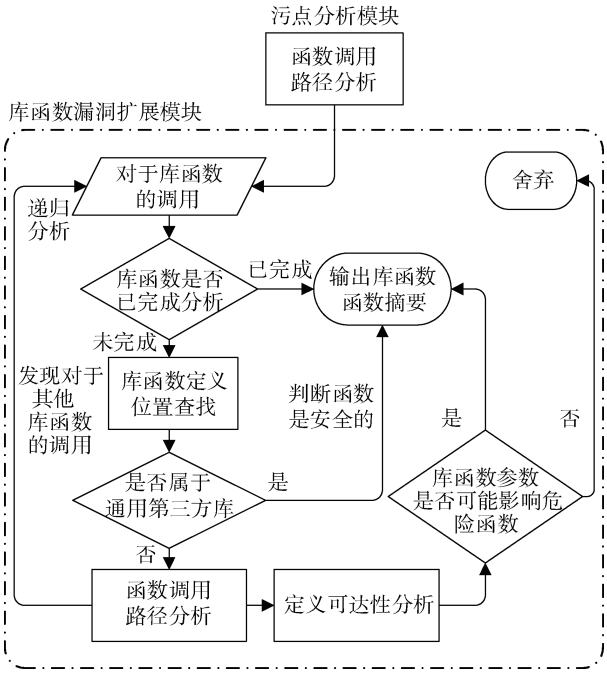


**图 8 污点汇聚点扩充模块**

**Figure 8 The sink point expand module**

由图 8 可知, 本模块在设计上采用了延迟绑定的设计思想, 当污点分析主模块中的函数调用路径分析发现污点源可达的路径之中存在某一未经分析的库函数时, 如果该库函数并不位于 libc, libcrypto 等经过广泛测试, 含有漏洞的可能性较低的第三方动态链接库中, 而是位于厂商自行实现的动态链接库中, 模块将对其进行分析。

在对库函数的分析部分, 本模块沿用定义可达性分析方法的分析思路, 分析该库函数的输入参数到危险函数的危险参数间是否存在定义使用关系, 并将最终的分析结果存储在污点汇聚点扩充信息库中, 在后续过程中如果函数调用路径分析又发现了对于本库函数的调用, 可以直接加载信息库中的分析结果。如果本模块分析过程中发现某一库函数调用了另一个动态链接库中的库函数, 可以递归调用本模块自身进行分析。

### 4.4 系统设计

FirmRD 基于 python 实现, 各类静态分析方法使用了 angr[16]框架提供的 API 接口。FirmRD 的交叉引用查找功能同时提供了使用 angr[16]与 Ghidra[19]实现的两种接口, 并在测试过程中选用了 Ghidra[19]接口。这一选择的原因在于 angr[16]框架无法识别使用多条指令引用特定字符串的交叉引用关系。

图 9 展示了一个 angr[16]框架无法识别的交叉引用示例, 此类问题属于 angr[16]、radare2[20]等常用静态分析框架底层 API 接口的固有缺陷。在识别数据

段中特定字符串在代码段的引用位置时, 如果同时使用多条指令计算字符串地址, 例如图 9 右下侧代码中同时使用 LDR(将特定地址加载到寄存器中)与 ADD(加法操作)两条语句在寄存器 R3 中加载字符串“*stbEn*”所在地址, angr[16]与 radare2[20]倾向于认为字符串地址仅由 LDR 一条语句索引, 因此定位了错误的字符串引用地址“[*pc*, #0x3a0]”, 遗漏了对于“*stbEn*”的交叉引用信息。

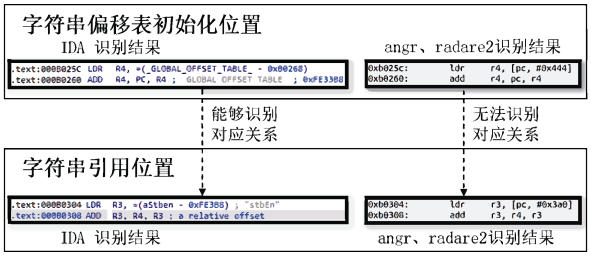


**图 9 Tenda AC18 httpd 程序交叉引用示例**

**Figure 9 A cross-reference example from Tenda AC18 httpd program**

另一方面, 如图 9 左侧所示, IDA[21]和 Ghidra[19]两个反编译器在分析交叉引用关系时正确考虑了此类引用类型。在本示例中, 它们首先在 0xB025C 处关联LDR与ADD两条语句, 计算出了两条指令执行后 R4 中存储的正确地址, 即字符串表 *GLOBAL_OFFSET_TABLE* 所在位置先减去再加上 PC 寄存器取值 0xB0268 后的取值。进一步地, 在 0xB0304 处, 反编译器使用相似方法识别出, 对于“*stbEn*”字符串的引用是使用两条语句计算处的基于字符串表 *GLOBAL_OFFSET_TABLE* 的相对偏移, 从而正确识别到了对于此地址的引用。

因此, 为了不在分析时遗漏上述交叉引用查找类型, 仅就字符串引用位置的查找功能而言, FirmRD 采用 Ghidra[19]框架的 API 接口等效替代了 angr[16]框架中的相关 API。

# 5 实验及测试

本文实验过程中使用的硬件与软件环境分别如表 1 与表 2 所示:

**表 1 硬件环境**

**Table 1 Hardware environment**

| 硬件 | 配置 |
|---|---|
| 主机型号 | Dell PowerEdge R740 |
| CPU | Intel® Xeon(R) Gold 6230R CPU @ 2.10 GHz |
| 内存 | 32GB |
| 储存 | 2TB |

**表 2 软件环境**

**Table 2 Software environment**

| 软件 | 配置 |
|---|---|
| Ubuntu | 20.04.3 LTS |
| Linux Core | 5.15.0-58-generic |
| python | 3.8.10 |
| angr | 9.0.6885 |
| Ghidra | 10.0.4 |

本文实验测试部分主要回答以下几个问题:

(1) 与前沿漏洞挖掘框架相比, FirmRD 基于定义可达性分析的漏洞路径发现方法是否能够进行更加精确, 更加高效的漏洞挖掘?

(2) FirmRD 前端分析模块中实现的污点源扩充算法是否能够在不降低漏洞发现精确程度的前提下扩充污点源识别数量, 进而使系统发现更多漏洞? 该扩充算法对于 API 函数的识别是否能够全面识别不同类型的用于处理用户输入的 API 函数?

(3) FirmRD 的污点汇聚点扩充模块是否能够有效扩充污点汇聚点的识别范围?

(4) FirmRD 是否具备真实漏洞发现能力?

本文与前沿分析框架 SaTC[7]进行了对比实验, 通过对实验数据的分析分别回答了前三个问题, 并通过在对比测试数据集与额外的扩展数据集上的真实漏洞发现效果测试回答了第四个问题。

在对比实验中, 本文使用了 SaTC 与 KARONTE[10]进行对比实验时共同使用的 Karonte 数据集[22], 该数据集包含来自 Netgear、TP-Link、D-Link、Tenda 四个厂商的 49 款路由器固件。在测试真实漏洞发现能力时, 本文也在由 6 个 TOTOLINK 厂商路由器固件组成的扩展数据集上进行了额外的漏洞挖掘测试, 统计了系统在该数据集上发现的 1-day 与 0-day 漏洞警报数量。

## 5.1 与 SaTC 的对比测试

表 3 展示了 FirmRD 与 SaTC 框架在 Karonte 数据集[22]上的对比实验结果, 其中, SaTC 的测试数据来源于其论文实验部分提供的原始数据[7]。在本实验中, 本文将在 5.1.1、5.1.2 与 5.1.3 节中分别回答第 5 章开头时提出的前三个问题。

### 5.1.1 基于定义可达性的污点分析方法能力测试

在本小节中, 我们将验证以下问题: 与前沿漏洞挖掘框架相比, FirmRD 基于定义可达性分析的漏洞路径发现方法是否能够进行更加精确, 更加高效的漏洞挖掘?

(1) 精确程度测试

为验证本文基于定义可达性的污点传播方法的

**表 3 FirmRD 与 SaTC 对比测试实验结果-警报数量与正确率**

**Table 3 The results of the comparison experiment between FirmRD and SaTC-Alert and Accuracy**

| 厂商 | 设备系列 | 样本数 | SaTC | | | FirmRD-禁用污点源扩充算法 | | | FirmRD-启用污点源扩充算法 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 警报 Alert | 正确数* #TP | 正确率 TP rate | 警报 Alert | 正确数* #TP | 正确率 TP rate | 警报 Alert | 正确数* #TP | 正确率 TP rate |
| Netgear | R/XR/WNR | 17 | 1901 | 537 | 28.25% | 493 | 434 | 88.03% | 6305 | 5836 | 92.56% |
| D-Link | DIR/DWR/ DCS | 9 | 32 | 22 | 68.75% | 16 | 7 | 43.75% | 33 | 18 | 54.55% |
| TP-Link | TD/WA/WR/ TX/KC | 16 | 7 | 2 | 28.57% | 55 | 31 | 56.36% | 79 | 45 | 56.96% |
| Tenda | AC/WH/FH | 7 | 144 | 122 | 84.72% | 156 | 142 | 91.03% | 326 | 291 | 89.26% |
| Total | - | 49 | 2084 | 683 | 32.77% | 720 | 614 | 85.28% | 6743 | 6190 | 91.80% |

*本对比实验沿用了 SaTC[7]的判定标准, 如果在静态分析视角下漏洞路径是可达的, 则判断该漏洞警报为正确警报(True Positive, #TP), 反之则判断对应漏洞警报为误报。

漏洞发现精确程度, 我们对比了 FirmRD 与 SaTC 发现漏洞时的正确率。由表 3 可知, 在相同数据集中, 除了对于 D-Link 设备固件外, FirmRD 漏洞发现的正确率都高于 SaTC, 总正确率相较 SaTC 提升了 1.8 倍, 且该正确率提升效果与禁用启用污点源扩充算法基本无关, 这体现了本文污点传播方法相较 SaTC 方法而言具有更高的漏洞发现精度。

FirmRD 在 D-Link 设备固件中识别正确率偏低的原因在于其无法找到足够数量的正确警报。由表 3 可知, 在 D-Link 设备固件中, FirmRD 启用污点源扩充算法时找到的正确警报数量少于 SaTC, 因此在总警报数量基本持平的情况下 FirmRD 的正确率低于 SaTC。该情况产生的主要原因在于 D-Link 厂商通用网关接口(Common Gateway Interface, CGI)处理程序中用户输入处理方法存在特殊性, FirmRD 的污点源扩充算法与该处理方法并不适配, 无法找到足够数量的正确警报。由于该问题与污点分析方法无关, 相关情况将在论证污点源扩充算法能力的 5.1.2 节中详细论述。

另一方面, FirmRD 也在 D-Link 固件的 ncc 等二进制程序中发现了不同于 SaTC 的一些漏洞入口, 该类漏洞入口可能导致与 CVE-2021-45382 模式相似的漏洞产生。由于漏洞复现难度较大, 我们目前暂未生成漏洞概念验证(Proof of concept, POC)代码, 后续也会继续尝试复现分析并与厂商进行沟通。

(2) 运行效率测试

在系统运行效率方面, 我们对比了 FirmRD 以及 SaTC 原始数据中的平均运行时间。由于不同厂商设备之间污点源引用数量存在差距, 不同厂商中系统运行时间效率差异显著, 为归一化运行时间开销, 本测试中还统计了 FirmRD 需要进行处理的污点源引用位置, 并计算了分析单条引用平均需要消耗的时间, 相关测试结果如表 4 所示:

**表 4 FirmRD 与 SaTC 对比测试实验结果-运行效率**

**Table 4 The results of the comparison experiment between FirmRD and SaTC-Efficiency**

| 厂商 | 设备系列 | 样本数 | SaTC | FirmRD-启用污点源扩充算法 | | |
|---|---|---|---|---|---|---|
| | | | 平均分析时间 | 平均分析时间 | 平均污点源引用位置 | 单条引用平均时间 |
| Net-gear | R/XR/WNR | 17 | 16.78h | 28.95h | 7721 | 13.50s |
| D-Link | DIR/DWR/DCS | 9 | 1.95h | 2.12h | 640 | 11.93s |
| TP-Link | TD/WA/WR/TX/KC | 16 | 4.22h | 2.75h | 1017 | 9.73s |
| Tenda | AC/WH/FH | 7 | 12.32h | 4.87h | 1443 | 12.15s |

由表 4 可知, 即使增加污点源扩充算法以及污点汇聚点扩充模块带来了额外的开销, FirmRD 在 TP-Link 以及 Tenda 设备中的总体平均运行效率仍然优于 SaTC, 在 D-Link 设备中 FirmRD 与 SaTC 的运行效率基本持平, 在 Netgear 设备中 FirmRD 平均分析时间相较 SaTC 而言更长。在 Netgear 设备中 FirmRD 分析时间更长的主要原因在于其发现了更多的漏洞警报, 由表 3 可知, FirmRD 在 Netgear 设备中发现的漏洞警报数量是 SaTC 的 3.3 倍。在用单条引用位置平均分析时间归一化运行时间开销之后, FirmRD 在不同厂商中的单条引用位置平均分析时间基本一致, 证明 Netgear 设备中平均运行时间增长可被归因于污点源引用数量的增加。

### 5.1.2 污点源扩充能力测试

在本小节中, 我们将进一步回答以下问题:

1) FirmRD 前端分析模块中实现的污点源扩充算法是否能够在不降低漏洞发现精确程度的前提下扩充污点源识别数量, 进而使系统发现更多漏洞?

2) 该扩充算法对于 API 函数的识别是否能够全面识别不同类型的用于处理用户输入的 API 函数?

(1) 污点源识别数量扩充

为验证污点源扩充算法是否能够在不降低漏洞发现精确程度的前提下扩充污点源识别数量, 进而使系统发现更多漏洞, 我们着重关注表 3 中的以下结果:

1) 启用污点源扩充算法后 FirmRD 与 SaTC[7]的正确警报数量的对比: 除了对于 D-Link 设备固件外, FirmRD 启用污点源扩充算法后, 在各厂商设备中产生的正确警报数普遍多于 SaTC[7]。此结果表明 FirmRD 覆盖了 SaTC[7]未曾覆盖的漏报情况, 成功实现了污点源识别数量以及漏洞识别数量的扩充。

2) 禁用与启用污点源扩充算法前后 FirmRD 的正确警报数量变化: 禁用与启用污点源扩充算法前后 FirmRD 的正确警报数量变化显著, 此结果表明 FirmRD 对正确警报数量的扩充的确来源于污点源扩充算法对 SaTC[7]遗漏的 API 函数引用位置的正确识别。

3) 禁用与启用污点源扩充算法前后 FirmRD 的漏洞识别正确率变化: 启用污点源扩充算法后 FirmRD 的漏洞识别正确率基本不变或略微升高, 此结果表明污点源扩充算法的引入不会降低漏洞发现的精确程度。此处的不会降低精确程度包含 Tenda 厂商固件中污点源扩充后误报率的略微升高, 误报在不同程序片段中的分布情况可能受到随机因素的影响, 通过对相关结果的分析确认, Tenda 厂商固件中误报率的小幅度提升并非来自于对污点源的错误识别, 而是随机因素所导致的细微误差。

需要补充说明的是, 在 D-Link 厂商设备中, FirmRD 启用污点源扩充算法时与 SaTC[7]找到的警报数量基本持平, 但其找到的正确警报数量反而产生了少量下降。通过分析系统漏洞警报, 本文发现 FirmRD 对于 D-Link 设备漏洞产生漏报的主要原因在于该厂商固件中各 CGI 处理程序对于用户输入的获取接口并未封装为 API 函数, 因此本文基于 API 函数引用位置识别污点源的方法无法获取此类污点源引用位置。

图 10 显示了上述漏报的一个示例, 本漏洞的 CVE 编号为 CVE-2018-6530, 是在 D-Link 厂商 DIR-868 设备的 cgibin 程序中发现的一个命令注入漏洞。由方框中代码逻辑可知, 该设备在获取网络报文中用户输入时并未调用任何 API 函数, 而是在匹配到特定用户输入关键字后通过将指向报文内容的指针向后移动直接获取字符串地址。

```
/* Grab a pointer to the request url */
requri = getenv("REQUEST_URI");
/* goto failure if the urquri does not start with "?service=" */
if((query = strchr(requri, "?")) == NULL or strncmp(query, "?service=", strlen("?service=")))
{
    goto failure_condition;
}
/* Point the control type field pointer 9 bytes beyond the requri */
controlType = query + strlen("?service=");

... ...

/* Create/open a shell script using the specified control type string */
sprintf(filename, "%s/%s_%d.sh", "/var/run", controlType, getpid());
fn = fopen(filename "a+");
/* Write self-deleting command into the srcipt and execute it by "sh -c"*/
if(fn)
{
    fprintf(fn, "rm -f %s/%s_%d.sh", "/var/run", controlType, getpid());
    fclose(fn);
    sprintf(filename, "%s/%s_%d.sh", "/var/run", controlType, getpid());
    system(filename);
}
```

**图 10 CVE-2018-6530 漏洞触发点信息**

**Figure 10 The information of the vulnerability trigger point of CVE-2018-6530**

考虑到绝大多数厂商都会在获取用户输入时封装 API 函数, 增强代码复用能力, 避免生成重复代码, FirmRD 使用 API 函数识别引用位置识别污点源的方式仍然具备普遍适用性。为应对此类不封装用户输入 API 的厂商, 后续可以针对具体厂商生成污点源引入模式的特征匹配规则。由于根据测试结果生成针对性特征匹配规则, 进而改善测试结果的行为可能影响对比测试的有效性, 本文在对比测试中暂未引入相关操作, 因此 FirmRD 在本测试中仍会遗漏此类漏洞警报。

此外, 在启用污点源扩充算法后, FirmRD 在 Netgear 厂商中的漏洞警报数量增长数量较多, 原因在于该算法在用户输入关键字信息充足时, 可能定位到通过配置文件、设备外设、JSON 结构等方式读取用户输入信息的 API 函数调用。此类用户输入 API 不与网络输入直接相关, 但是设备外设与配置文件仍有遭到攻击者影响的可能。虽然出于现实攻击难度的原因, 此类需要物理接触与复杂篡改的漏洞警报厂商通常不会受理, 但是本文认为, 由于未来可能产生更加高效的攻击手法, 此类危险使用情况仍有必要被纳入到漏洞报警的范围中。

(2) 污点源识别类型全面性

为证明污点源扩充算法对于用于处理用户输入的不同类型 API 函数的全面识别能力, 表 5 列举了该算法对于不同类型 API 函数的识别结果。由表 5 可知, 本算法可以成功识别从网络报文、配置文件、设备外设以及 JSON 结构中读取用户输入信息的各类

**表 5 API 函数识别测试结果**

**Table 5 The results of the API function identification experiment**

| 类型 | 厂商 | 设备型号 | 二进制程序 | API 函数 |
|---|---|---|---|---|
| 网络报文读入 (WebsGetVar) (含等效无符号函数) | Netgear | R6200v2 | httpd | sub_15C34 |
| | TP-Link | Archer_C5v2 | httpd | sub_B4680 |
| | Tenda | AC18 | httpd | sub_2B884 |
| | TOTOLINK | T10 | cstecgi.cgi | WebsGetVar |
| 配置文件读入 | Netgear | R7500 | net-cgi | config_get |
| | TP-Link | C2600 | openssl | NCONF_get_string |
| | Tenda | AC6 | multiWAN | GetIniFileValue |
| | TOTOLINK | T10 | Sysconf | inifile_get_string |
| 设备外设读入 | Netgear | R6200v2 | httpd | acosNvramConfig_get |
| JSON 结构读入 | TOTOLINK | T10 | cstecgi.cgi | cJSON_AddItemToObject |

API 函数, 其中包括无符号函数, 这体现了本文污点源扩充算法对于不同类型 API 函数的全面识别能力。

由于同一厂商不同型号的设备中 API 函数识别情况较为相似, 完全列举重复性较高, 且本实验主要验证 API 函数识别类型的全面性, 污点源识别数量的扩充能力已在前述实验中进行了分析, 表 5 对每个厂商每类情况至多仅保留一条结果。

### 5.1.3 污点汇聚点扩充能力测试

在污点汇聚点扩充能力方面, 表 6 列举了 FirmRD 发现的各厂商中触及危险函数的部分自定义库函数, 以展示 FirmRD 评估厂商自定义动态链接库中安全隐患的能力。同一动态链接库中触及相同危险函数的结果仅展示一条。

**表 6 污点汇聚点扩充能力测试结果**

**Table 6 The results of the sink point expand experiment**

| 安全隐患 | 厂商 | 所属动态链接库 | 库函数 | 危险参数 | 危险函数 |
|---|---|---|---|---|---|
| 参数触及堆栈溢出危险函数 (长度限制不存在或失效) | Netgear | libacos_shared.so | getTokens | *a1* | strcpy |
| | | | get_wlan_channel | *a1* | sprintf |
| | | libnat.so | agApi_tmschAddConf | *a1-a5* | sprintf |
| | Tenda | libtpi.so | verify_value | *a4* | strcpy |
| | | | restart_dns_proxy | *a1-a3* | sprintf |
| | | | tpi_portfilter_set | *a2* | memcpy |
| 参数触及命令注入危险函数 | Netgear | libacos_shared.so | get_wlan_channel | *a1* | system |
| | Tenda | libtpi.so | tpi_l2tpc_set_nat | *a1* | doSystemCmd |

由表 6 可知, FirmRD 能够发现不同厂商自定义动态链接库中函数触及溢出或命令注入危险函数的使用情况, 并将其纳入污点汇聚点范围之中。该结果验证了污点汇聚点扩展算法的可用性及其对于不同危险函数类型可达性情况识别的全面性。

另一方面, 虽然污点汇聚点识别范围的确进行了有效扩大, FirmRD 并未发现触发点在自定义的厂商库函数中的漏洞警报。这既是由于不同厂商自定义库函数编写的规范程度不同, 也受制于 FirmRD 对于间接调用以及动态加载机制识别能力的不足。

虽然无法成功检测自定义库函数中现有真实漏洞路径, FirmRD 的污点汇聚点扩充模块依旧具备了可用性, 其提高了系统分析面的覆盖率, 并且有助于检测将来可能产生的, 厂商自定义库函数误用导致的安全隐患。

## 5.2 真实漏洞发现能力测试

至今为止, FirmRD 在 Karonte 数据集[22]中共计验证并上报了 4 个 0-day 漏洞, 其中 Netgear 的两个命令注入漏洞已申请了 CVE 编号, Tenda 的两个漏洞也已获得了国家信息安全漏洞共享平台(CHINA NATIONAL VULNERABILITY DATABASE, CNVD)授予的漏洞编号, 但其漏洞细节仍未公开, 对应结果如表 7 所示。在经过广泛验证的数据集中仍能发现新的 0-day 漏洞也能证明本文的方法对于真实漏洞的发现能力。

为了进一步测试系统对于真实漏洞的挖掘能力, 本文收集了 6 个 TOTOLINK 厂商路由器固件并在由

**表 7 对比测试过程中发现的 0-day 漏洞**

**Table 7 The 0-day vulnerabilities discovered in the comparison experiment**

| 设备 | 0-day 漏洞 |
|---|---|
| Netgear R6200v2、R6300v2 | CVE-2022-30078 |
| Netgear R6200v2 | CVE-2022-30079 |
| Tenda AC9 | 未公开缓冲区溢出漏洞 (CNVD-2023-54790) |
| Tenda AC18 | 未公开命令注入漏洞 (CNVD-2023-54791) |

其组成的扩展数据集上也进行了漏洞路径发现测试。由于扩展数据集中固件所包含的 1-day 漏洞数量明确, 因此除了对于漏洞路径进行静态分析验证之外, 本实验中还统计了各漏洞警报对应的 0-day 与 1-day 漏洞数量, 并对影响范围扩展问题与 0-day 漏洞问题分别进行了复现验证, 进一步论证了系统对于真实漏洞的正确识别能力。

TOTOLINK 升级固件中不存在 httpd、goahead 等常见网络服务程序, 因此, 针对其设备的现有漏洞挖掘研究主要关注 cstecgi.cgi 程序的漏洞情况。FirmRD 对于该程序中漏洞的识别结果如表 8 所示, 其中直接对应 1-day 条目一列统计了与本设备中发现的 1-day 漏洞存在直接对应关系的漏洞警报数量; 影响范围扩展条目一列统计了与非本设备中发现的对应同一触发位置的 1-day 漏洞存在对应关系的漏洞警报数量; 对应 0-day 漏洞条目一列统计了与本设备中已复现的由未被 CVE 记录的漏洞触发位置导致的 0-day 漏洞对应的漏洞警报数量; 1-day 编号数一列统计了与 FirmRD 输出结果有对应关系的 CVE 编号数量(同一 CVE 编号可能对应多条漏洞警报)。

**表 8 漏洞发现扩展测试结果**

**Table 8 The result of the extra experiment of vulnerability discovery**

| 设备 | 漏洞程序 | 警报 Alert | 正确数#TP | 直接对应 1-day 条目 | 影响范围扩展条目 | 对应 0-day 漏洞条目 | 1-day 编号数 |
|---|---|---|---|---|---|---|---|
| A720R | cstecgi.cgi | 12 | 10 | 3 | 1 | 6 | 3 |
| A3700R | cstecgi.cgi | 11 | 9 | 4 | 5 | 0 | 4 |
| A7000R | cstecgi.cgi | 11 | 9 | 4 | 5 | 0 | 4 |
| LR350 | cstecgi.cgi | 20 | 16 | 16 | 0 | 0 | 7 |
| NR1800X | cstecgi.cgi | 21 | 17 | 17 | 0 | 0 | 8 |
| T10 | cstecgi.cgi | 7 | 7 | 0 | 3 | 4 | 0 |
| Total | - | 82 | 68 | 44 | 14 | 10 | 26 |

由表 8 可知, FirmRD 在此扩展数据集中以 82.93%的正确率发现了 68 条静态分析视角下的正确漏洞警报, 其中 44 条漏洞警报存在已知 CVE 漏洞对应关系, 共计对应了 26 条 CVE; 其余漏洞警报条目中, 14 条警报为已知漏洞触发点在其他设备上的影响范围扩展, 10 条警报对应于未知触发点导致的 0-day 漏洞。相关 0-day 漏洞都已进行复现验证, 其中的 8 条警报对应的四个漏洞触发点已经得到了厂商的确认, 目前正在进行修复, 影响范围扩展问题也在与厂商进行沟通。

由此可知, FirmRD 在扩展数据集上仍然具备高精度漏洞挖掘能力, 对于 0-day 与 1-day 漏洞对应情况的统计结果也进一步佐证了 FirmRD 对于正确警报评估标准的可靠性。

## 6 结论

通过使用定义可达性分析进行反向污点跟踪, 本文设计并实现了一种基于污点分析的低误报率自动化漏洞挖掘系统, 并分别通过识别函数的参数定义特征定位用户输入 API 函数以及识别库函数内的漏洞触发位置, 扩大了污点源与污点汇聚点的识别范围。实验结果表明, 本文所设计的方法可以进行高精度的漏洞挖掘, 且具备了发现 0-day 漏洞的能力。

本文所实现的系统在固件自动化漏洞挖掘方面已经具备了一定的可用性, 但系统仍然存在可以继续完善的改进方向。由于主要基于 angr[16]实现, 且 angr[16]的符号执行模块易用性较强, 可以通过符号执行方法进一步检验系统输出结果的正确性, 从而进一步降低系统误报率。另一方面, 通过继续扩展系统对于不同漏洞类型的识别范围, 可以进一步扩展系统的全面性与可用性, 例如可以通过扩展系统识别整数溢出类型的漏洞。

## 参考文献

[1] IoT Analytics. IoT 2022 in Review: The 10 Most Relevant IoT Developments of the Year. https://iot-analytics.com/iot-2022-in-review/. Jan. 2023.

[2] Yan H, Peng G J, Luo Y, et al. Survey on Smart Home Attack and Defense Methods[J]. *Journal of Cyber Security*, 2021, 6(4): 1-27. (严寒, 彭国军, 罗元, 等. 智能家居攻击与防御方法综述[J]. *信息安全学报*, 2021, 6(4): 1-27.)

[3] CISRC, New in-the-wild Exploit for Routers Could Affect Millions of Devices [EB/OL]. https://www.ics-cert.org.cn/portal/page/122/1004a030d740441cb5276b821c92ef23.html, Aug. 2021.
(关键基础设施安全应急响应中心, 针对路由器的最新在野漏洞利用, 或影响数百万设备. [EB/OL]. https://www.ics-cert.org.cn/portal/page/122/1004a030d740441cb5276b821c92ef23.html, Aug. 2021.)

[4] Chen D D, Egele M, Woo M, et al. Towards Automated Dynamic Analysis for Linux-Based Embedded Firmware[C]. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016: 1-16.

[5] Zheng Y W, Davanian A, Yin H, et al. Firm-Afl[C]. *The 28th USENIX Conference on Security Symposium*, 2019: 1099-1114.

[6] Wang L, Li F, Li L, et al. Principle and Practice of Taint Analysis[J]. *Journal of Software*, 2017, 28(4): 860-882.
(王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实践应用[J]. *软件学报*, 2017, 28(4): 860-882.)

[7] Chen L B, Wang Y H, Cai Q P, et al. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems[C]. *USENIX Security Symposium*, 2021

[8] Cheng K, Li Q, Wang L, et al. DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware[C]. *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018: 430-441.

[9] Zhang L, Chen J Y, Diao W R, et al. CryptoREX: Large-Scale Analysis of Cryptographic Misuse in IoT Devices[C]. *International Symposium on Recent Advances in Intrusion Detection*, 2019.

[10] Redini N, Machiry A, Wang R Y, et al. Karonte: Detecting Insecure Multi-Binary Interactions in Embedded Firmware[C]. *2020 IEEE Symposium on Security and Privacy*, 2020: 1544-1561.

[11] Chen S, Lin Z, Zhang Y. SelectiveTaint: Efficient Data Flow Tracking with Static Binary Rewriting[C]. *USENIX Security Symposium*, 2021: 1665-1682.

[12] Wu H, Zhou S L, Shi D H, et al. Review of Symbolic Execution Technology and Applications[J]. *Computer Engineering and Applications*, 2023, 59(8): 56-72.
(吴皓, 周世龙, 史东辉, 等. 符号执行技术及应用研究综述[J]. *计算机工程与应用*, 2023, 59(8): 56-72.)

[13] Collard J F. Reasoning about Program Transformations Imperative Programming and Flow of Data[M]. New York: Springer, 2003: 77.

[14] Tekchandani R, Bhatia R, Singh M. Semantic Code Clone Detection for Internet of Things Applications Using Reaching Definition and Liveness Analysis[J]. *The Journal of Supercomputing*, 2018, 74(9): 4199-4226.

[15] Li L, Bissyandé T F, Papadakis M, et al. Static Analysis of Android Apps: A Systematic Literature Review[J]. *Information and Software Technology*, 2017, 88: 67-95.

[16] 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.

[17] Gritti F, Pagani F, Grishchenko I, et al. HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images[C]. *2022 IEEE Symposium on Security and Privacy*, 2022: 1082-1099.

[18] Craig Heffner. Binwalk - Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk, Nov. 2013.

[19] NSA. Ghidra. https://github.com/NationalSecurityAgency/ghidra, Mar. 2019.

[20] Radare2 Team. Radare2 github Repository. https://github.com/ radareorg/radare2, Jul. 2012.

[21] hex-rays. IDA Pro. https://www.hex-rays.com/products/ida/, Dec. 2022.

[22] Karonte dataset. The Experimental Dataset Used by Tool Karonte. https://github.com/ucsb-seclab/karonte#dataset, Oct. 2019.

**梅润元** 于 2022 年在武汉大学信息安全专业获得学士学位。现在武汉大学网络空间安全专业攻读硕士学位。研究领域为软件漏洞挖掘。研究兴趣包括: IoT 安全、漏洞自动化挖掘与利用。CCF 学生会员, Email: maybemei@whu.edu.cn

**王衍豪** 于 2019 年在中国科学院大学计算机应用技术专业获得工学博士学位, 研究领域包括软件安全和物联网安全。Email: yanhao.wang@nio.com

**李子川** 于 2020 年在武汉大学信息安全专业获得学士学位。现在武汉大学网络空间安全攻读硕士学位。研究领域为软件漏洞挖掘。研究兴趣包括: IoT 安全、UEFI 安全。Email: river_li@whu.edu.cn

**彭国军** 于 2008 年在武汉大学信息安全专业获得博士学位。武汉大学国家网络安全学院教授。研究领域为网络安全与系统安全。Email: guojpeng@whu.edu.cn