基于物联网设备局部仿真的反馈式模糊测试技术

卢昊良[1,2,3,4], 邹燕燕[1,2,3,4], 彭 跃[1,2,3,4], 谭凌霄[1,2,3,4], 张 禹[1,2,3,4], 刘龙权[1,2,3,4], 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 北京 中国 100049

摘要 近几年物联网设备数量飞速增长, 随着物联网的普及, 物联网设备所面临的安全问题越来越多。与物联网设备相关的安全攻击事件中, 危害最大的是利用设备漏洞获得设备最高权限, 进而窃取用户敏感数据、传播恶意代码等。对物联网设备进行漏洞挖掘, 及时发现物联网设备中存在的安全漏洞, 是解决上述安全问题的重要方法之一。通过模糊测试可有效发现物联网设备中的安全漏洞, 该方法通过向被测试目标发送大量非预期的输入, 并监控其状态来发现潜在的漏洞。然而由于物联网设备动态执行信息难获取以及模糊测试固有的测试深度问题, 使得当前流行的反馈式模糊测试技术在应用到物联网设备中面临困难。本文提出了一种基于物联网设备局部仿真的反馈式模糊测试技术。为了获取程序动态执行信息又保持一定的普适性, 本文仅对于不直接与设备硬件交互的网络服务程序进行局部仿真和测试。该方法首先在物联网设备的固件代码中自动识别普遍存在并易存在漏洞的网络数据解析函数, 针对以该类函数为入口的网络服务组件, 生成高质量的组件级种子样本集合。然后对网络服务组件进行局部仿真, 获取目标程序代码覆盖信息, 实现反馈式模糊测试。针对 6 个厂商的 9 款物联网设备的实验表明, 本文方法相比 FirmAFL 多支持 4 款物联网设备的测试, 平均可以达到 83.4%的函数识别精确率和 90.1%的召回率, 针对识别得到的 364 个目标函数对应的网络服务组件共触发 294 个程序异常并发现 8 个零日漏洞。实验结果证明了我们方法的有效性和实用性。

关键词 物联网设备; 模糊测试; 机器学习
中图法分类号 TP309.1 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2023.01.06

Feedback-driven Fuzzing Technology Based on Partial Simulation of IoT Devices

LU Haoliang[1,2,3,4], ZOU Yanyan[1,2,3,4], PENG Yue[1,2,3,4], TAN Lingxiao[1,2,3,4], ZHANG Yu[1,2,3,4], LIU Longquan[1,2,3,4], 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 In recent years, the number of IoT devices has grown rapidly. With the popularity of the IoT, the security issues of IoT devices are increasing. In the related security attack incidents, the main attack method is to use device vulnerabilities to obtain the highest authority of the device, and then steal the user's sensitive data and spread malicious code. Mining vulnerabilities in IoT devices and discovering security vulnerabilities in IoT devices in time is one of the important ways to solve the above security problems. Vulnerabilities in IoT devices can be effectively discovered by fuzzing. Fuzzing discovers potential vulnerabilities by sending a large number of unexpected inputs to the target being tested and monitoring its status. However, due to the difficulties of obtaining dynamic execution information of IoT devices and the inherent depth of fuzzing tests, the current popular feedback-driven fuzzing technology is difficult to apply to IoT devices. This paper proposes a feedback-driven fuzzing technology for IoT devices based on partial simulation. In order to obtain program dynamic execution information and maintain a certain universality, this paper only performs partial simulation and testing on network service programs that do not directly interact with device hardware. The method automatically identifies a certain type of function, i.e., network data analysis functions, and generates a set of high-quality component-level seed samples for network service components that take such functions as the entrance. It performs partial simulation, and

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: 2020-01-26; 修改日期: 2020-04-13; 定稿日期: 2022-11-14

obtains code coverage information. Experiments on 9 IoT devices from 6 manufacturers show that the method is able to support 4 more IoT devices than FirmAFL does. It achieves an average function recognition accuracy of 83.4% and a recall rate of 90.1%, and triggers a total of 294 crashes for objective components corresponding to the identified 364 objective functions. We have discovered 8 0-day vulnerabilities finally. Experimental results show the effectiveness and practicability of our method.

Key words IoT Devices; fuzzing; machine learning

1 引言

物联网(IoT)被认为是继计算机、互联网之后信息技术产业发展的又一次重要革命。近几年物联网设备数量飞速增长, 据统计, 2018 年全球共有 70 亿台物联网设备, 并保持每年20%左右的增长速度, 到 2020 年, 物联网设备预计可达 100 亿台, 到 2025 年将增加到 220 亿台[1]。随着物联网的普及, 物联网设备所面临的安全问题越来越多。根据卡巴斯基 IoT 安全报告[2]显示, 近年来捕获到的物联网恶意样本数量呈现爆炸式的增长。

据 Gartner[3]调查, 近 20%的企业或者相关机构在过去三年内遭受了至少一次基于物联网的攻击。物联网安全在企业中受到越来越多的重视, 为了防范安全威胁, Gartner 预测 2020 年全球物联网安全支出将达到 26 亿美元。与物联网设备相关的安全攻击事件中, 危害最大的是利用设备漏洞获得设备最高权限, 进而窃取用户敏感数据、传播恶意代码等。近几年, 物联网设备漏洞频繁被曝光, 2018 年 8 月 13 日, 在第 26 届全球黑客大会 Defcon 上[4], 安全研究人员破解了全世界最畅销的智能音箱——亚马逊 Echo 智能音箱, 通过控制音箱进行录音并将数据回传实现窃听, 在无需任何用户交互的情况下窃取用户资料和隐私数据; GeekPwn 国际安全极客大赛[5]上, 多个影响数十亿设备的物联网设备漏洞被发现并成功利用; 2018 年 7 月, 安全研究人员发现三星智能家居平台 SmartThings Hub 存在多个安全漏洞[6], 利用这些漏洞可以实现任意代码执行, 攻击第三方智能家居设备。

对物联网设备进行漏洞挖掘, 及时发现物联网设备中存在的安全漏洞, 是解决上述安全问题的重要方法之一。在物联网设备的众多攻击面中, 物联网设备处理并解析网络数据包部分存在的漏洞, 占了物联网设备漏洞的大多数, 据我们统计的 2019 年物联网设备 CVE 漏洞信息[7], 其中发生在网络数据包解析部分的漏洞占比约为 52%, 因此本文主要针对物联网设备的网络数据包解析部分进行漏洞挖掘。

众所周知, 模糊测试是针对软件和系统十分有效的漏洞挖掘方法, 该方法通过向被测试目标发送大量非预期的输入, 并监控其状态来发现潜在的漏洞。由于模糊测试技术可以在不对目标软件进行详细分析的前提下进行自动化测试, 因此该技术获得了安全研究人员的广泛使用。模糊测试根据程序执行信息的获取情况, 通常分为白盒、灰盒和黑盒测试。其中基于覆盖率制导的反馈式灰盒模糊测试技术是模糊测试中具有代表性的一类技术, 也是当前研究的热点。该技术主要依靠动态获取的目标程序覆盖信息来决定新的测试用例生成策略, 其中的代表工作有 American Fuzzy Lop(AFL)[8]、Honggfuzz[9]、LibFuzzer[10]等。这些工具都已在大量实际项目中检测到了数以百计的零日漏洞。

然而, 将反馈式模糊测试应用于物联网设备仍然存在许多挑战。首先, 程序的动态执行信息很难有效获取, 从而无法有效进行覆盖率制导的测试。大部分物联网设备都是定制的完整系统, 缺少在其实体设备中获取执行信息的手段和工具。而基于设备仿真的方式虽然可以潜在获取到相关的执行信息, 但因为物联网设备的异构性与多样性, 大量的设备无法实现系统级的仿真。其次, 模糊测试的效率和深度仍然存在不足。通过网络抓包获取初始测试种子样本并进行变异, 由于变异的盲目性导致大量无效或重复测试, 这对于基于设备仿真的模糊测试影响更大, 另外也会使得输入次数存在限制的程序难以充分测试[11]。同时, 多数物联网设备在网络通讯中都存在厂商自主实现的私有协议, 缺少协议逻辑的理解很难变异出触发深层代码执行的测试用例, 这些都导致了模糊测试无法高效地进行。

针对上述挑战, 本文提出了一种基于物联网设备局部仿真的反馈式模糊测试方法。为了获取程序动态执行信息又保持一定的普适性, 本文仅对于不直接与设备硬件交互的网络服务程序进行局部仿真和测试。我们发现物联网设备中的网络服务程序, 一般由多个网络服务组件组成。为了识别一个网络服务组件, 需要自动识别组件的入口函数, 即网络数据解析函数, 并通过调用关系获取网络服务组件中的函数。我们分析了大量的网络数据包解析关键函数, 发现大部分遵从特定的处理流程, 本文称之为 TLD 模式。基于该模式, 本文提出一种基于机器学习的网络数据解析函数识别方法。针对所识别出来的

每一个网络服务组件, 利用符号执行等技术生成高质量的组件级种子样本, 并在此基础上进行变异, 生成测试用例, 提高物联网设备模糊测试的效率和深度。针对 6 个厂商的 9 款物联网设备的实验表明, 本文方法相比 FirmAFL 多支持 4 款物联网设备的测试, 平均可以达到 83.4%的函数识别精确率和 90.1%的召回率, 针对识别得到的 364 个目标函数对应的网络服务组件共触发 294 个程序异常并发现 8 个零日漏洞①, 获批 2 个 CVE 编号和 1 个 CNVD[12]编号。

本文的主要贡献如下:

(1) 提出了一种针对物联网设备的反馈式模糊测试方法, 通过局部仿真执行实现了执行信息获取并提高了该方法的普适性;

(2) 提出了一种基于机器学习的网络数据解析函数识别方法, 适用于有符号及无符号的设备代码, 并以识别出来的函数为入口获取网络服务组件, 生成高质量的组件级种子样本, 可进一步提高模糊测试的效率和深度;

(3) 基于以上方法, 实现了系统原型 TLDFuzzer, 实验表明本方法相对当前代表性工具在支持设备数量和漏洞发现能力上均有提升, 并发现了零日漏洞, 验证了本文方法的有效性。

本文第 2 节对所提出的方法进行概述; 第 3 节介绍所提方法的详细设计; 第 4 节介绍本文的实验部分, 并对实验结果进行评估; 第 5 节介绍本文的相关工作; 第 6 节对所提方法进行总结。

2 方法概述

2.1 总体方法

反馈式模糊测试需要及时获取目标程序的执行信息。但是物联网设备作为定制化的系统, 缺少获取执行信息的工具, 比如缺少通用计算系统中的 pin[13]、valgrind[14]等。同时即使编写了动态信息获取工具, 由于物联网设备的非开放性, 这类工具也很难植入到设备中, 因此基于设备仿真实现反馈式模糊测试成为物联网设备漏洞挖掘的一种潜在技术途径。但因为物联网设备的异构性与多样性, 基于设备仿真很难做到具有普适性的仿真, 具体来说主要是因为进行系统级的固件仿真, 需要一系列特定的硬件外设条件等。

随着物联网设备功能更加丰富, 所处的网络环境更加多样, 物联网设备在使用过程中需要复杂的交互操作与管理配置, 如使用 HTTP 服务、UPnP 服务对物联网设备进行日常管理与参数配置, 以及通过 Cisco 的私有协议 CDP, 在设备间共享相关操作系统信息、IP 地址等, 所以设备中存在大量的公开或者私有的协议处理程序。通过对 2019 年已披露的物联网设备漏洞分布情况统计分析, 我们发现物联网设备披露的大部分漏洞存在于网络数据包处理程序中。在本文中, 网络数据包处理程序是指对外提供网络服务、接收并解析外部传入网络数据包的程序(本文称为网络服务程序, 即目标程序)。通常情况下, 网络服务程序由于不直接与硬件交互而可以相对容易地仿真执行[15], 因此针对这类程序实施反馈式模糊测试, 既可以覆盖物联网设备重要的攻击面, 又可以避免由于设备硬件依赖等因素导致系统级仿真失败的问题。

本文提出了一种面向物联网设备局部仿真的反馈式模糊测试方法。该方法在物联网设备的固件代码中识别网络数据解析函数(更加准确的定义参见 3.1 小节, 本文中也称为目标函数), 并以该函数为入口对网络服务组件进行局部仿真, 支持执行动态信息的获取。然后对目标组件生成高质量的组件级种子样本集合。最后使用所生成的种子样本对目标组件进行模糊测试, 并基于局部仿真获取目标程序控制流覆盖信息, 实现针对目标组件的反馈式模糊测试。该方法主要结构如图 1 所示。我们的方法需要解决两个主要的问题:

(1) 目标函数识别。一方面网络数据解析函数没有统一的命名规范, 另一方面实际的设备代码中存在大量无符号函数, 因此不能简单的通过函数名称来进行识别。我们通过分析实际物联网设备的网络数据解析函数, 发现函数处理流程大部分遵循基本相同的处理流程: 从网络数据包中读取相关字段类型及长度, 在长度检查后读取字段的数据值。在本文中用 TLD 模式刻画上述流程, 并建立了 TLD 模式的特征。本文对 2022 个网络数据解析函数提取 TLD 特征并构建训练集, 基于决策树(Decision Tree)算法[16], 生成目标函数预测模型, 用于目标函数的识别。

(2) 对目标网络服务组件进行深度的反馈式模糊测试。为了对网络服务组件进行深度测试, 本文采用当前广泛使用的反馈式模糊测试技术。为了获取组件动态执行信息, 本文基于 Qemu[17]对目标组件所在的网络服务程序进行仿真执行, 并通过控制执行流程, 在构建目标函数上下文及输入环境的基础上, 以该类函数为入口, 仅对目标组件进行仿真执行, 该方法在本文中称为局部仿真。为了有效触发目标组件中深层的代码, 一方面要构造满足网络数据解析函数, 即入口函数输入参数要求的种子样本, 另

① 其中 5 个为厂商已经确认，暂未分配漏洞标识的零日漏洞。

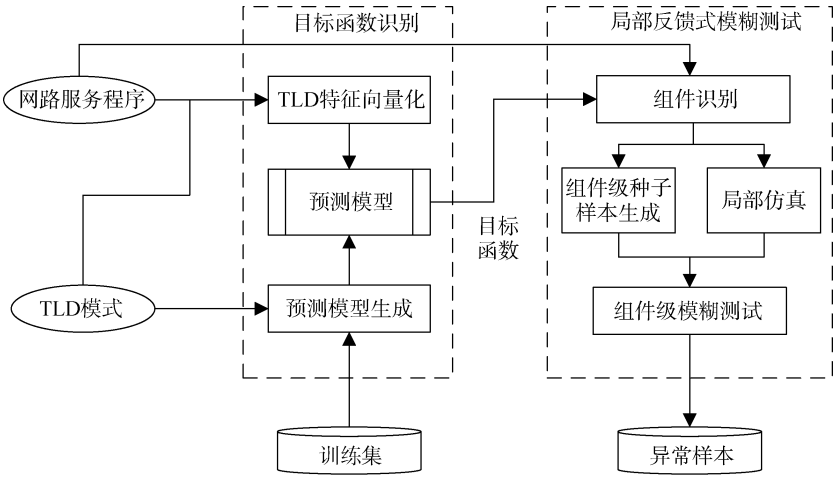


图 1　TLDFuzzer 总体结构

Figure 1　The structure of TLDFuzzer

一方面也要使用高质量的种子样本。因此本文针对目标组件使用符号执行等技术生成高质量种子样本。使用所生成的种子样本，组件级模糊测试引擎对种子样本进行变异，并接受局部仿真反馈的动态执行信息，完成反馈式模糊测试。

2.2　实例分析

依据图 1 所示的总体结构，本节将以某商业设备代码片段(如图 2 所示)为例来介绍本文方法。

预测模型生成

收集各主流品牌设备固件并标注其中的网络服务程序，从程序的函数中提取特定函数(如 strcpy、memcpy 等函数)的调用次数作为特征向量，构建训练集，并基于决策树算法生成目标函数预测模型，用于目标函数的识别。

TLD 特征向量化

针对图 2 所示函数，提取 TLD 模式所定义的特定函数调用次数(如第 11~12 行、第 15~16 行 memcpy 函数调用共计 4 次；第 18、20、22 调用 sprintf 函数共计 3 次)，构成特征向量为[4,3,0,4,0,47,5](向量中的 47 和 5 分别为 parseInfo 函数的基本块个数和父函数的个数，其余为特定函数的调用次数，具体说明见 3.1.2 小节)，利用训练生成的目标函数预测模型来对向量化后的函数进行识别，判定图 2 所示函数为网络数据解析函数，作为目标函数输出。

组件级种子样本生成

以识别出来的目标函数为入口获取网络服务组件，针对其使用符号执行等技术来生成高质量测试用例，就图 2 所示代码片段而言，可以触发程序执行到第 11~12 行代码的测试用例，即为高质量测试用例。因为传入数据的相关偏移需要满足一系列特定的 *MAGIC BYTES* 值，才能执行到这类深层次的代码。具体来说，*user_input[1e,1b,1f,1d,...,16,1f,...]*即为一个可以执行到 parseInfo 函数中第 11-12 行代码的高质量测试用例。

```
1   int parseInfo(char *user_input)
2   {
3       ...
4       if((user_input[0] != 0x1e) || (user_input[1] != 0x1b) ||
        (user_input[2] != 0x1f) || (user_input[3] != 0x1d))
5       {
6           return 0;
7       }
8       ...
9       if((user_input[14] == 0x16) && (user_input[15] == 0x1f))
10      {
11          memcpy(probuf_1,user_input+16,100);
12          memcpy(probuf_2,info+8,4);
13      }
14      ...
15      memcpy(v1,v8,16);
16      memcpy(v2,v9,10);
17      ...
18      sprintf(cTemp, "Device ProductID : %s", devicePI);
19      fputs(cTemp,fp);
20      sprintf(cTemp, "Device MacAddress : %s", deviceMA);
21      fputs(cTemp,fp);
22      sprintf(cTemp, "Device Firmware version : %s", deviceFV);
23      fputs(cTemp,fp);
24      ...
25      puts("Write to conf!");
26      ...
27  }
```

图 2　某商业设备网络数据解析函数代码片段

Figure 2　The code snippet of a commercial device network data parsing function

反馈式模糊测试

利用模糊测试引擎对生成的种子样本进行变异，并通过控制程序执行流程构建好目标组件的入口函数，即 parseInfo 函数的上下文及输入环境，接受局部仿真反馈的动态执行信息，完成反馈式的灰盒模糊测试，最后获取触发程序异常的样本。

3　详细设计

3.1　目标函数识别

我们关注能够通过网络协议等方式与外界交互

的程序。因为不接收外部输入数据的程序, 即使存在安全漏洞, 也很难被攻击者利用。网络服务程序的主要处理流程如图 3 所示, 程序首先会进行初始化配置工作, 然后会利用 socket 建立网络连接, 接下来在接收到网络数据包后会分发给不同的 Action 进行处理, 如解析外部输入数据用来构建内部数据结构、转化为记号流(具体定义参见 3.2.1 小节)、执行特定功能(如智能路由器中进行网络诊断的功能, 见图 4)等。我们将实现上述功能的代码称为网络服务组件。网络服务组件通常有一个入口函数, 本文称为网络数据解析函数。在网络服务程序的调用图上, 由网络数据解析函数及以该函数为根的子树节点上函数组成的代码, 即为网络服务组件, 是本文模糊测试的对象。

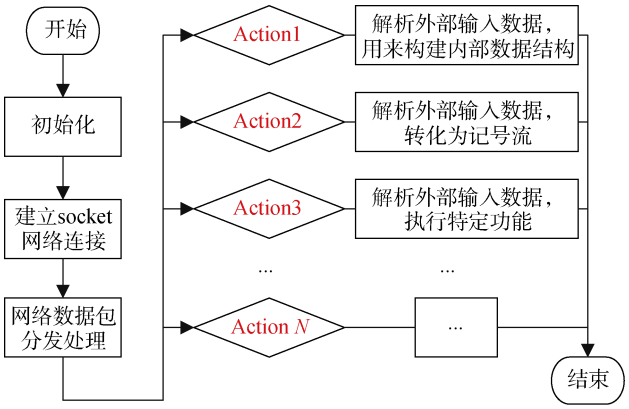


图 3 物联网设备中网络服务程序的主要处理流程

Figure 3 The main process flow of network service program in IoT device

3.1.1 TLD 模式构建

由于固件程序对外部输入数据解析错误导致的安全漏洞是常见的、涉及范围广泛的一类漏洞。尤其在物联网设备内负责提供网络服务的程序中, 在对外部输入的解析过程中由于边界校验不严格、前后不一致、处理逻辑错误等导致的漏洞在所有物联网程序漏洞中占比很大。我们发现网络数据解析函数基本遵循相同的处理流程, 如图 5 所示。

本文用 TLD 模式刻画这个处理逻辑。TLD 中三个字母的含义分别是: T — Type(类型), L — Length(长度), D — Data(实际数据值)。T 代表接收数据的字段类型或关键字, L 代表数据中字段的长度, D 代表数据中字段的实际数值。TLD 模式可以抽象成如下三个步骤:

1. 读取字段类型或关键字 T, 判断是否满足要求, 满足则继续执行;

2. 读取字段的长度 L, 根据 L 进行与长度相关的数据操作, 如长度检查、赋值大小、拷贝长度等;

3. 读取字段的实际数值 D(污点数据), 将污点数据写到之前分配好的相关内存中。

在程序对外部输入数据的解析过程中, 如果对 T-L-D 三者处理不当将会导致一系列的安全问题, 如越界读写等。如图 4 代码片段所示, 第 13 行进行字段关键字 T 的检查, 存在“ping”类型字段, 进行后续操作, 第 19 行并没有对 L 做出任何限制, 直接将格式化后的数据 D 写入栈中, 从而导致了栈溢出漏洞。

```
1   // 执行网络诊断功能的函数
2   int network_diagnosis(char *a)
3   {
4       char buf[0x100];
5       ...
6       if (v28 ==1)
7       {
8           //判断是否存在非预期字符
9           if ( !strchr(src,";") && !strchr(src,"$") && !strchr(src,".."))
10          {
11              sprintf(s1,"touch %s","tmp/.web_diag.txt");
12              system(s1);
13              if (strstr(src,"ping"))
14              {
15                  ...
16                  strcpy(v2,v1);
17                  //用以将数据格式化为一段合法的网络诊断
    shell命令
18                  //此处存在栈溢出漏洞
19                  sprintf(s2,"ping -c 4 %s -l %s 1>>%s
    2>>&1",v2,v8,"tmp/.web_diag.txt");
20                  system(s2);
21                  ...
22              }
23              if (strstr(src,"traceroute")) {...}
24          }
25          else
26          {
27              sprintf(s3,"echo \"Unexpected characters exist!\n\"
    >>  %s","tmp/.web_diag.txt");
28              system(s3);
29              puts("There are unexpected characters.");
30          }
31          ...
32      }
33      else
34      {
35          ...
36          if ( !strncmp(s4,"start",5) ) {...}
37          else if ( !strncmp(s4,"output",6) ) {...}
38          else if ( !strncmp(s4,"stop",4) ) {...}
39          else {...}
40          ...
41      }
42  }
```

图 4 智能路由器中执行网络诊断功能的代码片段

Figure 4 Code snippets in smart routers that perform network diagnostics

```
1   type = Locate_Read_Type();
2   if(type == T)
3   {
        //依据相关字段的数据类型或类别进行数据的定位与获取
4       Operate_T(type);
        //获取数据相关字段的长度
5       length = Locate_Read_Length();
        //依据length进行相关长度的数据操作，如长度检查、确定
    分配内存的大小、拷贝长度等
6       Operate_L(length);
        //依据获取到的type和length进行相关数值操作
7       Operate_D(data);
8   }
```

图 5 TLD 模式解析流程伪代码

Figure 5 Pseudocode of analytical process according with TLD model

3.1.2 目标函数预测模型生成

为建立网络数据解析函数的预测模型，我们首先深入分析了来自 20 个实际物联网设备的网络数据解析函数，发现目标函数会调用大量字符与字符串定位操作的相关函数来对网络数据中的关键字段进行定位(用以确定 TLD 模式中的 T、L 值)；在解析外部输入数据后，不论是构建内部数据结构还是将数据转化为记号流，都存在大量的内存操作，尤其以内存拷贝操作居多(进行 TLD 模式中的 D 相关操作)；在解析外部输入数据并执行特定功能的函数中，存在字符串格式化相关的操作，用以将数据转化为特定格式(TLD 模式中的 T、D 操作)。因此我们选取待分析函数中所调用的字符串处理和标准输入输出函数的调用次数作为特征。目前，我们在嵌入式系统标准库函数中选取了 string 类与 stdio 类[18]共 24 个函数(见表 1 第 3 列前 5 行)作为 TLD 模式识别的特征函数，并分为 5 个特征函数类，分别是内存拷贝操作、字符与字符串定位操作、字符与字符串比较操作、数据格式化操作、标准输出操作。本方法统计待分析函数中调用每个特征函数的次数，并按照特征函数类进行累加，最后把累加和作为目标函数识别特征的一维。由于网络数据解析函数通常规模较大且会进行一系列的复杂操作，一般也会被其他函数大量调用以分别解析不同的网络数据包，因此把基本块的个数和父函数的个数分别作为目标函数识别特征的另外的 2 个维度。

以图 4 所示函数为例，我们来具体说明如何依据表 1 中的特征对函数进行向量化。network_ diagnosis 函数分别调用 strchr、strstr 函数 3 次和 2 次，故字符与字符串定位操作 *STR_LOCATE_ITEM* 对应的数值为 5，同理我们可以获取到其他特征对应的数值，形成 1 个特征向量(如图 6 所示)。

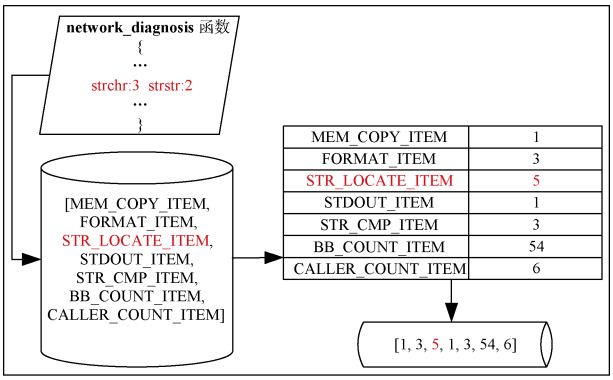


图 6 TLD 特征向量化

Figure 6 Vectorization of TLD features

表 1 TLD 模式的特征

Table 1 The features of TLD model

特征名称	特征定义	TLD 模式涉及的函数集合
MEM_COPY_ITEM	内存拷贝操作	strcpy,memcpy,strncpy,strcat,strncat
FORMAT_ITEM	数据格式化操作	printf,sprintf,snprintf,fprintf
STR_LOCATE_ITEM	字符与字符串定位操作	strstr,strchr,strrchr,strtok,strspn,strcspn,strsep
STDOUT_ITEM	标准输出操作	puts,putchar,fputs,fputc
STR_CMP_ITEM	字符与字符串比较操作	strcmp,strncmp,memcmp,strcasecmp
BB_COUNT_ITEM	基本块的数量	/
CALLER_COUNT_ITEM	调用者的数量	/

本文选择决策树算法[16]对训练数据集进行训练生成目标函数预测模型。决策树算法是一种通过一系列规则解决数据分类问题的算法。由于目标函数识别问题本质上是一个分类问题，决策树算法具有分类精度高、生成模式简单、所需样本数量少、训练时间短等特点。故选择决策树算法作为本文目标函数预测模型的生成方法，其可以在有限的训练数据集基础上，得到较好的分类结果。

3.1.3 目标函数识别算法

为了识别出目标函数，本文从固件的网络服务程序中提取出待识别函数的地址，对待识别函数依据 3.1.2 小节所述方法形成特征向量，利用训练好的目标函数预测模型来进行目标函数识别。

目标函数识别方法如算法 1 所示，输入为目标网络服务程序，输出为识别得到的目标函数在网络服务程序中的地址。第 1 行为遍历网络服务程序 P 中的所有非导入函数作为待识别的函数集合；第 3 行为获取待识别函数中调用的所有标准库函数；第 4 行和第 5 行为统计待识别函数中各标准库函数(即表 1 中前 5 类特征涉及的函数集合)的调用次数，以形成特征向量 *featureVec[]*；第 7 行和第 8 行分别统计待识别函数中的基本块数量和被调用的次数；第 9 行为使用目标函数预测模型来识别特征向量对应的函数是否为目标函数，如果是目标函数，则在第 10 行

将目标函数对应的地址保存并输出, 如果不是, 则返回执行第 2 行的代码。

算法 1. 识别目标函数.
输入: 目标网络服务程序 *P*
输出: 目标函数的地址 *Addr[]*
1: *functionList* ← *AllFunctions*(*P*)//获取程序中所有非导入函数
2: FOR *function* in *functionList*
3: *libList* ← *LibAPIs*(*function*)//获取待识别函数中调用的所有标准库函数
4: FOR *libFunc* in *libList*
5: *featureVec[]* ← *FeatureCount* (*libFunc*)//统计待识别函数中各标准库函数的调用次数, 以形成特征向量
6: END FOR
7: *featureVec[]* ← *BlockCount* (*function*)//统计基本块数量, 添加到特征向量
8: *featureVec[]* ← *CallerCount* (*function*) //统计待识别函数父函数的数量, 添加到特征向量
9: IF *PredictionModel*(*featureVec[]*)
10: *Addr[]* ← *FunctionAddr* (*function*)//如果目标函数预测模型识别向量为目标函数, 则将目标函数地址输出
11: ELSE
12: continue
13: END IF
14: END FOR
15: RETURN *Addr[]*

3.2 反馈式模糊测试

目前主流的模糊测试方法是基于覆盖率制导的反馈式模糊测试技术, 即通过判断测试过程中执行路径的信息来指导测试用例的生成, 使其尽可能多地覆盖程序的路径。目前广泛使用的反馈式模糊测试方法主要基于遗传算法, 在测试用例的执行过程中获取代码插桩后的路径信息或程序异常信息, 通过预先定义的适应函数和选择策略变异输入数据, 具有执行效率高的优点, 但对于输入的测试用例仍存在盲目性变异, 可能导致深层次代码路径很难或无法被覆盖等问题。

本文采用局部反馈式模糊测试策略, 针对网络服务程序中的网络服务组件进行反馈式模糊测试, 可以有效地减少模糊测试的时间。网络数据解析函数和它调用的子函数组成的模块, 本文称其为网络服务组件。采用局部仿真技术得到目标组件的路径覆盖信息。同时针对传统反馈式模糊测试存在盲目性变异的问题, 采用组件级种子样本生成技术来指导高质量种子样本的生成, 使其尽可能多的覆盖程序执行路径。局部反馈式模糊测试流程如图 7 所示, 针对目标组件进行测试用例生成, 同时通过控制网络服务程序的执行流程, 仅对目标组件进行模拟执行来获取执行动态信息, 从而进行反馈式模糊测试, 最终得到异常样本。

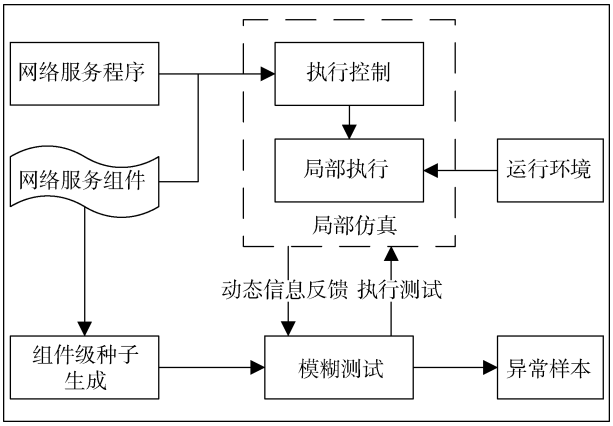


图 7 局部反馈式模糊测试流程
Figure 7 Process of feedback-driven fuzzing based on partial simulation

3.2.1 组件级种子样本生成技术

为实现局部反馈式模糊测试, 有效触发目标组件深层次代码的执行, 不仅需要构造满足入口函数输入参数要求的种子样本, 同时需要使用高质量的种子样本。图 2 所示为解析某私有协议网络数据包的函数代码片段, 我们可以发现, 当输入数据前 4 个字节的值为“\x1e\x1b\x1f\x1d”, 同时第 15、16 个字节的值为“\x16\x1f”时, 才能执行到 parseInfo 函数的第 11、12 行代码, 否则函数将会提前退出, 无法继续探索更多的路径。即使基于覆盖率制导的变异也难以生成能够覆盖到上述代码位置的测试用例。因此, 针对目标组件本文使用符号执行等技术来生成高质量测试用例。

算法 2. 组件级种子样本生成.
输入: 目标函数 *Func*
网络服务程序对应的真实网络数据包 *Packet*
输出: 种子样本 *SeedSpec*
1: *networkComponent* ← *ComponentIdentify*(*Func*)
2: *protocolContent* ← *ReadNetworkData* (*Packet*)//读取网络数据包内容

3: IF all *characters* in *protocolContent* ∈ Printable ASCII or CR or LF//判断目标程序解析的协议类型

4: *protocol_is_text* ← TRUE

5: ELSE

6: *protocol_is_text* ← FALSE

7: END IF

8: IF *protocol_is_text* //文本格式协议

9: *key[]* ← *ExtractKey* (*networkComponent*)//遍历组件汇编指令, 提取关键字

10: *SeedSpec* ← *ConstructRequest* (*key[]*)//构造测试用例

11: ELSE//非文本格式协议

12: *constraintCondition* ← *SymbolicExec* (*networkComponent, protocolContent*)//符号执行

13: *SeedSpec* ← *ConstraintSolving* (*constraintCondition*)//约束求解

14: END IF

15: RETURN *SeedSpec*

组件级种子样本生成如算法 2 所示, 输入目标函数 *Func* 与网络服务程序对应的真实网络数据包 *Packet*, 输出种子样本 *SeedSpec*。我们发现物联网设备应用层的网络协议数据包主要有 2 种格式, 一种是文本格式(例如图 4 代码片段对应解析的 HTTP 协议), 另外一种是非文本类的二进制格式(例如图 2 代码片段对应解析的某私有协议)。所以第 1 行我们首先根据目标函数的函数调用关系做组件识别, 识别出目标函数及其调用的子函数为网络服务组件; 第 2 行按字符读取目标网络服务程序的真实网络数据包中的内容, 作为算法判别目标网络服务程序处理协议类型的依据; 第 3 行判断第 2 行读取的字符若均为可打印 ASCII 字符或回车符(CR)、换行符(LF), 那么第 4 行定义目标网络服务程序处理的协议类型为文本格式协议, 反之为二进制格式协议, 见第 5、6 行。依据目标网络服务程序处理协议类型的不同, 算法分为两个分支来生成种子样本, 算法 8~15 行将分别说明两种格式种子样本的生成。

在网络服务程序处理文本格式协议时, 程序针对如图 8 上半部分所示的文本协议数据进行对多个键(key)赋值(value)操作, 得到如图 8 下半部分所示的键值信息。所以当目标网络服务程序处理的协议类型是文本格式的协议时, 本算法需要构造网络数据中的键用来生成种子样本。在算法第 8 行判断网络服务组件是否为处理文本格式协议, 如果是, 则第 9 行依据目标网络服务组件遍历汇编指令, 获取指向常量字符串的地址, 提取仅由数字、字母、下划线的连续字符组合成的字符串, 进而获得目标组件处理的所有关键词 *key*; 第 10 行为构造种子样本, 由于这类文本格式协议 *key* 的顺序不影响处理结果, 故我们可以直接将得到的 *key* 按获取顺序组合并构造形如 "*key_1=value&key_2=value&…&key_n=value*" 的合法请求, 得到高质量种子样本, 因为本算法意在构造网络数据中的键(*key*), 故其中的值(value)为任意字符串。

如果是二进制格式协议, 在第 12 行对网络数据包内容进行符号化后, 针对网络服务组件进行符号执行, 得到约束条件; 第 13 行为对约束条件进行约束求解, 得到高质量种子样本。为进行符号执行, 首先我们将识别到的目标网络服务组件作为符号执行的入口, 将参数进行符号化, 通过将所有符号化参数变量的约束条件置为空, 以此作为一个初始状态, 然后将其加入到状态集合中, 从初始状态开始进行符号执行。当符号执行遍历过相关路径后, 会产生大量的约束, 借助约束求解引擎, 计算得到满足这些约束的输入, 形成高质量的测试用例。

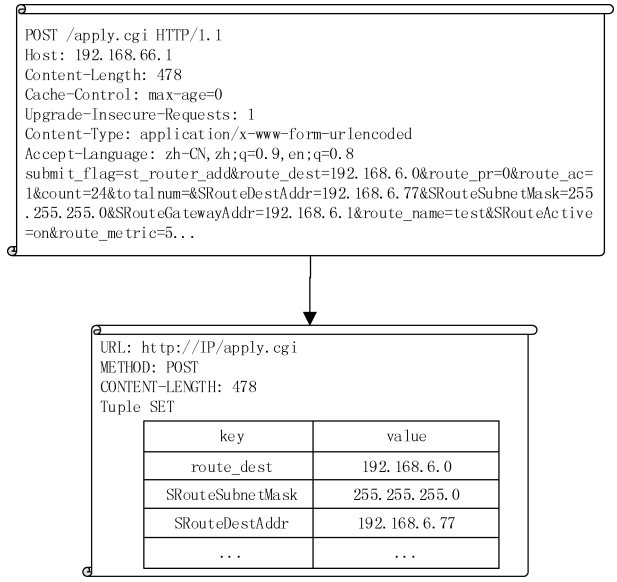


图 8 HTTP 的 POST 数据转化示例

Figure 8 The transformation instance of POST data in HTTP

3.2.2 局部仿真技术

仅对目标组件而不是整个固件进行仿真, 需要解决两个问题: 1)如何将控制流跳转到目标组件的入口函数, 即网络数据解析函数; 2)如何正确地构建目标组件的执行环境。

本文使用动态链接库劫持技术实现执行控制。一般的进程在执行主函数前, 会首先执行初始化函数, 根据环境变量中定义的顺序, 依次加载动态链

接库。我们可以通过修改环境变量中定义的动态链接库加载顺序, 优先加载包含测试床程序的动态链接库, 在测试床程序中利用执行控制程序, 改变原来调用流程, 在完成目标函数调用环境构建的基础上, 将控制权交给测试入口函数, 即网络数据解析函数, 从而完成局部仿真, 如图 9 所示。我们以 Linux 系统为例来具体说明上述流程。首先将测试床程序编译成测试床动态链接库, 通过修改 Linux 系统中环境变量优先加载该动态链接库。在该库中, 劫持 ELF 程序的初始化函数, 完成执行控制和目标函数调用环境构建(其中目标函数调用环境参见 4.1 节的描述)。在目标组件执行过程中, 需要依赖原有系统中的库文件等。我们通过装载原有设备文件系统的方式, 正确构建目标组件执行环境, 保证目标组件的正常执行。

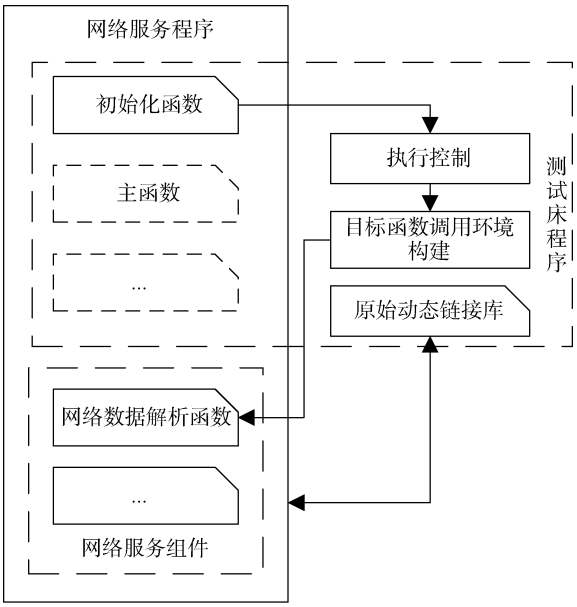


图 9 局部仿真技术示意图

Figure 9 Schematic diagram of partial simulation technology

3.2.3 局部反馈式模糊测试算法

算法 3. 局部反馈式模糊测试.

输入: 目标函数 *Func*

输出: 异常样本 *SeedSpec*

1: *networkComponent*←*ComponentIdentify*(*Func*)

2: *seedQueue*←*TestCaseBuilder* (*networkComponent*)//生成测试用例

3: *patchedComponent* ← *ProgramPatch*(*Func*)//构建目标函数上下文及输入环境

4: *targetComponent*←*Instrument* (*patchedComponent*)//插桩

5: REPEAT

6: *seed*←*seedQueue.top*

7: *mutatedFiles*←*Mutate*(*seed*)//变异

8: FOR *file* in *mutatedFiles*

9: *coveragePath*, *isCrash* ← *execute*(*targetComponent*, *file*)//使用测试用例执行程序

10: IF *isCrash*

11: *Spec*←*file*

12: RETURN *Spec*//输出异常样本

13: END IF

14: IF *coveragePath* not in *coverageMap*

15: *coverageMap*←*coverageMap* + *coveragePath*//添加覆盖路径

16: *seedQueue*←*seedQueue* + *file*//添加种子队列

17: ELSE

18: continue

19: END IF

20: END FOR

21: UNTIL *keyboardInterrupt*

本文提出的局部反馈式模糊测试方法, 将识别出来的网络数据解析函数作为目标网络服务组件的入口, 逐次分别针对目标网络服务组件进行组件级的模糊测试。

局部反馈式模糊测试如算法 3 所示, 输入算法 1 识别出的目标函数, 即种子的输入口, 输出异常样本。第 1 行为根据目标函数的函数调用关系做组件识别, 识别出目标函数及其调用的子函数为网络服务组件; 第 2 行为生成合法的高质量测试用例, 形成种子队列; 第 3 行为构建目标函数上下文及输入环境; 第 4 行为在局部仿真模式下对目标组件进行插桩; 第 6 行和第 7 行为从种子队列中取出一个测试用例并对其进行变异; 第 8 行和第 9 行为使用每个变异后的测试用例执行插桩后的程序, 得到程序的覆盖路径和是否出现 crash; 第 10 行到第 12 行为如果程序执行时出现 crash, 则认为此测试用例为一个异常样本, 保存并输出; 第 14 行到第 16 行为如果执行当前测试用例过程中触发了新的执行路径, 则将程序执行路径添加到覆盖路径集合中, 同时将测试用例添加到种子队列中; 第 17 行到第 18 行为如果没有触发新的路径, 则继续执行下一个测试用例; 第 21 行

为重复第 6 行到第 20 行的行为, 直到用户手动中断模糊测试为止。

4 实验验证

4.1 系统原型框架实现

基于物联网设备局部仿真的反馈式模糊测试技术, 本文使用 Python 和 C 实现了系统原型框架 TLDFuzzer。框架分为目标函数识别和模糊测试两大模块, 其中目标函数识别模块包括固件预处理、特征提取、模型生成与函数识别 4 个部分, 模糊测试模块包括局部仿真、组件级种子样本生成和局部反馈式模糊测试 3 个部分。

对物联网设备进行分析, 首先要获取设备的固件。本文获取设备固件主要通过如下 4 种方式: 从厂商的官网进行爬取下载; 抓取通过 OTA(Over-the-Air Technology) [19]方式升级的请求网络数据包进而获得固件的下载链接; 利用设备的硬件调试接口将系统文件打包上传间接获取固件; 利用编程器直接读取固件存储芯片。

一个固件的组成结构如图 10 所示, 主要分为固件头部(Header)、引导程序(Bootloader)、内核(Kernel)、文件系统(RootFs)等。文件系统是设备固件运行的基础, 包含了网络服务程序在内的实现设备各种功能的基础应用程序。因此获得固件后, 提取其中的文件系统是后续对其进行漏洞挖掘的基础。Binwalk[20]是一个固件分析工具, 用于协助研究人员进行固件分析、文件提取等工作。FMK(Firmware Mod Kit) [21]工具的功能和 Binwalk 工具类似, 它包含了自动化分析固件的一系列脚本。我们利用上述两种工具从固件中解析并提取出对应的文件系统并手工找到具有通过网络协议与外界交互功能的网络服务程序。这类网络服务程序中通常包含一系列与处理网络协议相关的关键字, 它们通常存放于 /usr/sbin、/usr/bin、/sbin、/bin 目录下, 而且命名通常以字母 d 结尾, 如 httpd、snmpd、tftpd 等。所以我们可以比较容易地发现这类程序作为我们的目标程序。

特征提取模块利用 IDA Pro[22]辅助脚本完成。模型训练与识别模块使用 Python 的 sklearn 库[23], 目标函数参数信息借助 Ghidra[24]确定。判别网络服务程序处理协议类型的部分基于 Python 实现。测试用例生成模块, 对于非文本类的二进制格式协议, 基于 angr[25], 并对其进行了扩展。对于文本格式协议, 我们同样基于 IDA Pro 辅助脚本进行提取并构建。

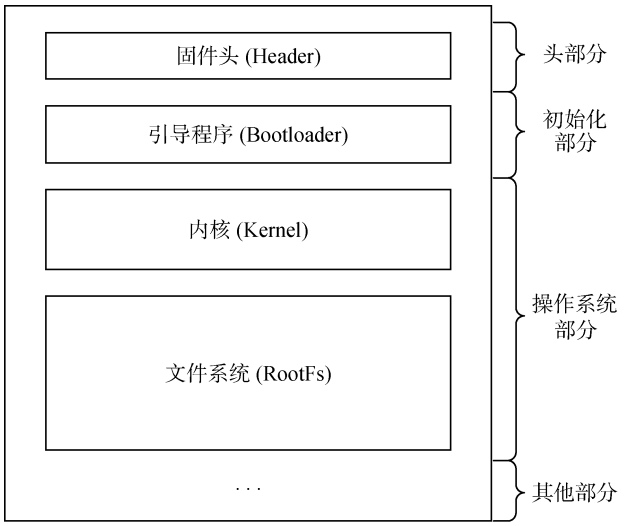


图 10 固件组成结构

Figure 10 The component structure of firmware

模糊测试基于 AFL 的 Qemu 模式, 局部仿真借助 LD_PRELOAD 机制, 通过控制程序执行流程, 仅对目标组件进行仿真执行。我们通过利用 LD_PRELOAD 机制加载自定义的动态链接库, 进而覆盖原始初始化函数的处理逻辑来间接控制程序执行流程的方法, 来对目标组件进行仿真执行。为使 AFL 可以应用到目标组件的模糊测试, 我们在目标函数调用环境构建的过程中, 使目标函数可以将标准输入作为接收数据的入口, 并为输入数据分配缓冲区, 进而将接收的数据传给目标函数。

4.2 实验环境及数据集

本文的实验环境如下: Intel Core i7, 3.0GHz CPU, 操作系统使用 Ubuntu 16.04, 实验中用到的 sklearn 版本为 0.18.1, Qemu 和 AFL 的版本分别为 2.10 和 2.52。

我们选取了市面主流品牌的物联网设备作为实验目标, 具体的厂商及相应的网络服务程序见表 2。

4.3 目标函数识别

我们从 Netgear、Trendnet、ASUS、D-Link、TP-Link 和 Tenda 6 个主流品牌设备的服务程序中选取了共计 6122 个函数作为训练集。通过函数关键字及字符串定位结合人工分析的方式, 共标记了 2022 个函数作为目标函数样本, 即正样本集, 选取了 4100 个函数作为负样本集。将表 2 所示 9 个程序中的所有函数作为验证集(与训练集独立), 其中包含 396 个目标函数。依据表 1 所示的特征, 我们分别针对训练集和验证集中的数据进行了特征化, 如图 6 所示, 将标记好的函数按照 TLD 模式的相应特征形

表 2 实验目标物联网程序

Table 2 The target IoT program of experiment

	厂商	型号	版本	程序	类别	CPU 指令集
1	Netgear	R7000	V1.0.5.70	httpd	智能路由器	ARM
2	Netgear	R8500	V1.0.2.100	httpd	智能路由器	ARM
3	Trendnet	TV-IP110WN	V1.2.2	cgibin*	智能摄像头	MIPS
4	ASUS	AC66	3.0.0.4_380	networkmap	智能路由器	ARM
5	Dlink	DAP-2695	1.11.RC044	httpd	智能路由器	MIPS
6	Dlink	DIR-825	2.02	httpd	智能路由器	MIPS
7	Dlink	DSL-3782	1.01	tcapi	智能路由器	MIPS
8	Tenda	AC9	US_V15.03.05.19	httpd	智能路由器	ARM
9	TP-Link	AC1900	V1_18	tddpd	智能家庭网关	ARM

(注: cgibin* 表示固件中的 cgi 程序集合)

成特征向量。

我们利用决策树算法进行训练, 生成目标函数预测模型。利用训练好的模型对目标程序进行识别, 具体结果如表 3 所示, 其中精确率 P_1 的计算公式为

$$P_1=\frac{\text{识别正确的个数}}{\text{识别到目标函数的个数}}\times 100\% \tag{1}$$

由结果可见, 9 组实验的平均精确率为 83.4%。我们分析了实验中识别结果不准确的情况, 主要是因为存在一些误报。具体来说, 实验中的一些非目标函数存在较多 TLD 模式的特征(如 strcmp, sprintf 调用数目较多), 如一些非目标函数的功能为将一些用户不可控的数据格式化后重定向到日志文件中, 所以影响了对函数识别的精确率。其中召回率 P_2 的计算公式为

$$P_2=\frac{\text{识别正确的个数}}{\text{实际目标函数的个数}}\times 100\% \tag{2}$$

实验的平均召回率为 90.1%, 由于部分目标函数 TLD 模式相关的特征函数(如字符与字符串比较操作涉及的函数、数据格式化操作涉及的函数等)调用较少, 具体来说, 例如程序中存在不需要将网络数据格式化成特定的命令, 且不依赖字符与字符串比较操作相关的函数来决定程序控制流的走向的目标函数, 它们更多地是调用 strcpy、memcpy 等函数进行内存拷贝操作, 所以未能将其识别出来。

4.4 测试用例生成

依据目标函数识别方法从实验程序中识别得到网络数据解析函数, 我们以这些目标函数为入口, 针对网络服务组件, 利用测试用例生成模块生成对应的测试用例, 平均每个目标组件生成 2.2 个测试用例。分别将利用测试用例生成模块生成的测试用例和抓取对应程序的网络数据包作为模糊测试的种子, 对目标进行模糊测试, 以此作为一组对比实验, 每组的测试时间为 1 h。针对每个实验程序我们随机选取 10 个识别得到的目标组件进行说明(如不足 10 个, 则选取所有识别得到的目标组件)。

表 3 决策树算法识别目标函数有效性评估

Table 3 Validity evaluation of decision tree algorithms on target function

	厂商	型号	程序	CPU 指令集	实际目标函数个数(个)	识别到的目标函数个数(个)	正确的个数(个)	精确率(%)	召回率(%)
1	Netgear	R7000	httpd	ARM	83	89	77	86.5	92.8
2	Dlink	DIR-825	httpd	MIPS	49	53	45	84.9	91.8
3	ASUS	AC66	networkmap	ARM	15	16	13	81.3	86.7
4	Dlink	DSL-3782	tcapi	MIPS	8	9	7	77.8	87.5
5	Netgear	R8500	httpd	ARM	80	87	75	86.2	93.7
6	Trendnet	TV-IP110WN	cgibin	MIPS	45	48	41	85.4	91.1
7	Dlink	DAP-2695	httpd	MIPS	19	21	17	80.9	89.5
8	TP-Link	AC1900	tddpd	ARM	13	13	11	84.6	84.6
9	Tenda	AC9	httpd	ARM	84	93	78	83.9	92.9
平均	/	/	/	/	44	47.7	40.4	83.4	90.1

图 11 显示了实验结果, 其中 x 轴代表测试的时间, y 轴代表选取的组件在某个时间点 ENTRY 数目(即通过变异生成的能够覆盖新代码路径的有效测试用例数目)的平均值。红色曲线代表使用测试用例生成模块生成的测试用例作为种子产生的 ENTRY 数目, 蓝色代表未使用测试用例生成模块产生的 ENTRY 数目。依据图 11 所示的实验结果, 使用本文测试用例生成模块生成的测试用例来进行模糊测试, 多数情况能够显著提高产生的 ENTRY 数目。具体来说, 使用本文方法生成的测试用例在相同的时间内产生的 ENTRY 数目最高可以达到未使用情况下的 4.7 倍, 平均可达 3.0 倍。因为本文方法可以生成覆盖到诸如图 2 代码片段所示的复杂条件的测试用例。而且从图中可以看出, 多数情况下蓝色曲线在产生较少的 ENTRY 数目后, 增长趋势就趋于平缓, 而红色曲线的增长速率一直高于蓝色曲线, 且趋于平缓时, 产生的 ENTRY 数目也高于蓝色曲线。因为未经处理的测试用例通过简单的变异无法覆盖到复杂条件的代码路径, 很快就会被程序丢弃掉, 很难进入到更深层次的路径, 也很难探索更广的代码路径。我们具体分析了 D-Link DSL3782 的 TCAPI 程序, 在目标组件中, *MAGIC BYTES* 这类复杂条件的代码路径较少, 在目标组件中占比不足 2%, 所以实验结果与另外 8 组相差较大, 但从总体上来看, 针对目标组件生成高质量测试用例的方法相较于未使用的情况下在代码覆盖能力上有显著提升。

4.5 1-day 漏洞触发时间

本节通过在真实设备程序中触发 1-day 漏洞的情况来评估 TLDFuzzer 的效果。对于 TLDFuzzer, 我们针对识别出的网络数据解析函数, 分别以该类函数为入口对网络服务组件进行模糊测试。

实验结果见表 4, 其中第 9 列表示 TLDFuzzer 成功识别出漏洞所在的组件, 进而以目标函数为入口对网络服务组件进行模糊测试, 且触发漏洞所用的时间。TLDFuzzer 的 1-day 漏洞触发时间最长为 248 s, 最短仅为 8 s, 平均时间为 80.6 s, 由实验结果可见我们的方法可以在较短的时间内触发程序的漏洞。由于 TLDFuzzer 是组件级模糊测试, 其 1-day 漏洞触发的平均时间为分钟级, 而 FirmAFL[26]是系统级模糊测试, 实验中其 1-day 漏洞触发的平均时间为小时级, 所以在 1-day 漏洞触发时间开销方面, 本文所提方法更有优势。表格中的 14 个 1-day 漏洞存在于 8 个

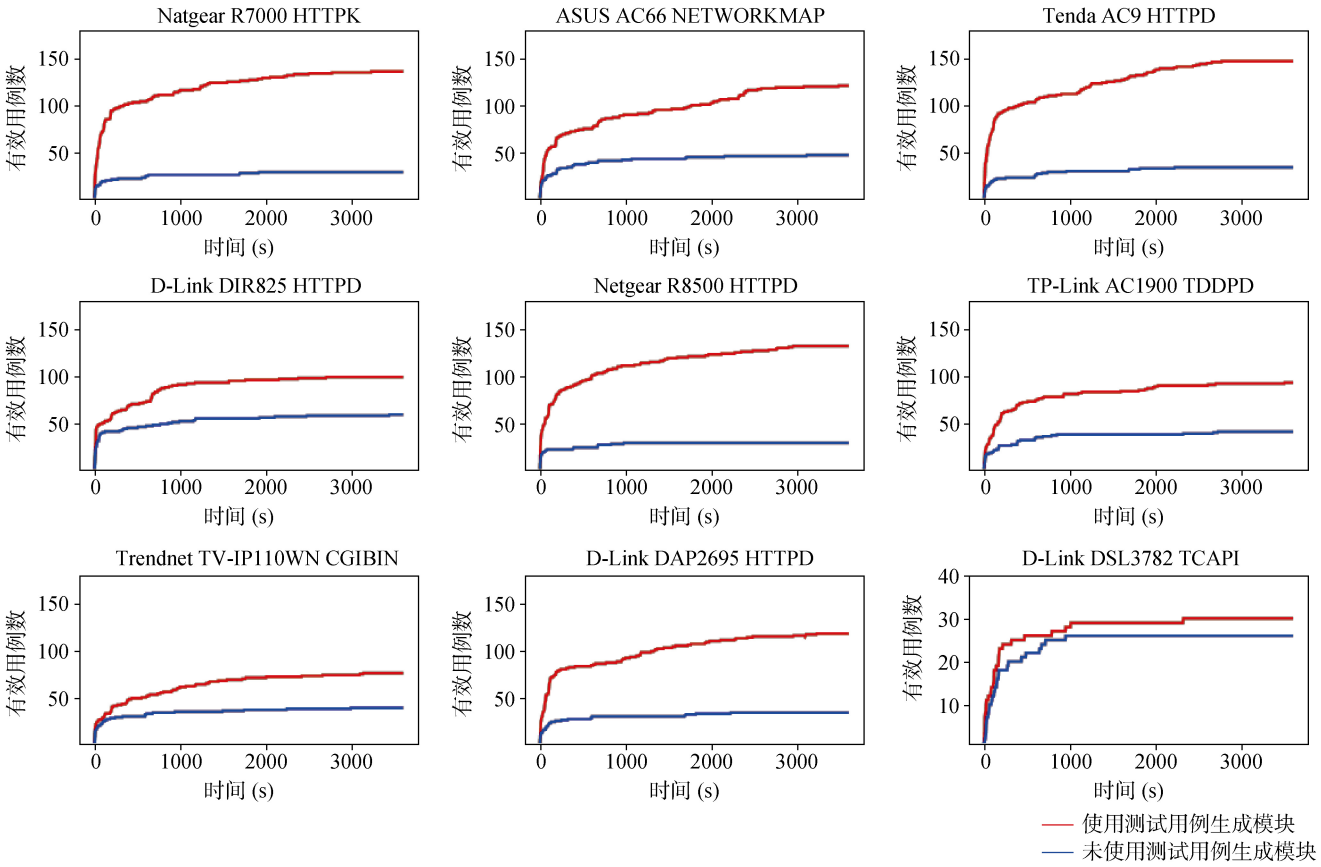


图 11 代码覆盖能力对比

Figure 11 Construction of code coverage ability

不同的程序中, 其中有 4 个程序 FirmAFL 不支持对其进行模糊测试, 而 TLDFuzzer 可支持对所有测试程序进行模糊测试。所以 TLDFuzzer 在针对物联网设备漏洞挖掘方面不仅具有高效性, 而且具有较好的普适性。

4.6 TLDFuzzer 总体实验情况

在 4.2 节的实验环境下, TLDFuzzer 从 9 个测试程序中识别出来共计 364 个目标函数, 分别以该类函数为入口对网络服务组件进行 1 小时(依据图 11 所示结果, 近 90%的测试程序在 300 s 后 ENTRY 数目的增长趋于平缓, 故针对每个目标组件进行 1 h 的模糊测试相对充足)的模糊测试, 共发现了 294 个 crash, 平均每个测试程序发现 32.7 个 crash。经我们后续分析, 其中大部分 crash 均由相同的已知 1-day 漏洞引起。有 7 个 crash 在原生程序中无法触发, 经过我们的分析, 均为程序在将网络数据传入目标函数之前调用了额外的函数进行检查, 且该函数在执行时对输入数据的限制致使目标函数未被执行。由于我们的方法是以网络数据解析函数为入口进行模糊测试, 该种情况的 crash 在原生程序中无法触发, 所以模糊测试的结果存在一定的误报性。在分析与去重后, 确认系统共发现了 22 个漏洞, 包括 14 个 1-day 漏洞(见表 4)和 8 个 0-day 漏洞。由表 5 可见, 8 个 0-day 漏洞获批了 2 个 CVE 编号和 1 个 CNVD 编号, 涉及了 D-Link、TP-Link 和 Tenda 共 3 个厂商的物联网设备, 漏洞类型涉及栈溢出与堆溢出漏洞(本文方法同样适用于其他类型内存破坏类漏洞的挖掘), 且所在函数均符合 TLD 模式, 他们都是由于 T-L-D 三者处理不当造成的。

5 相关工作

由于物联网设备漏洞会带来巨大的危害, 越来越多的安全研究人员将精力投入到物联网设备漏洞挖掘的研究工作中。在静态分析方面, Shoshitaishvi 等人提出关于在固件二进制程序中检测认证绕过漏洞的方法, 并实现了一个二进制分析框架 Firmalice[27]。然而, 该方法仅针对少数的几种商用设备的固件进行了分析, 数据集较少;同时利用该方法进行检测时, 需要安全研究人员事先对固件进行分析并人工提取一份安全策略, 严重降低了工具的自动化性能, 并且需要大量的人力成本。Costin 等研究人员在[28]中针对公开的物联网设备固件进行了大规模的静态分析用以发现漏洞,其存在较高的误报性。Cojocar 等提出了识别嵌入式设备固件中协议解析器的方法[29], 但其仅依据程序代码结构相关的特征用以识别目标。

表 4 1-day 漏洞触发情况

Table 4 1-day vulnerability triggering

	漏洞标识	厂商	型号	版本	程序	漏洞触发		TLDFuzzer 触发时间(秒)
						FirmAFL	TLDFuzzer	
1	CVE-2016-1558	Dlink	DAP-2695	1.11.RC044	httpd	√	√	92
2	CVE-2018-19241	Trendnet	TV-IP110WN	v1.2.2	cgibin	√	√	42
3	EDB-ID-38718	Dlink	DIR-825	2.02	httpd	√	√	77
4	CVE-2018-10749	Dlink	DSL-3782	1.01	tcapi	√	√	9
5	CVE-2018-10748	Dlink	DSL-3782	1.01	tcapi	√	√	8
6	CVE-2018-10747	Dlink	DSL-3782	1.01	tcapi	√	√	11
7	CVE-2018-8941	Dlink	DSL-3782	1.01	tcapi	√	√	14
8	CVE-2017-6548	ASUS	AC66	3.0.0.4_380	networkmap	N/A	√	143
9	PSV-2017-2427	Netgear	R8500	1.0.2.100	httpd	N/A	√	48
10	PSV-2017-2428	Netgear	R8500	1.0.2.100	httpd	N/A	√	98
11	PSV-2017-2226	Netgear	R8500	1.0.2.100	httpd	N/A	√	114
12	PSV-2017-2460	Netgear	R7000	V1.0.5.70	httpd	N/A	√	248
13	CVE-2018-8747	Tenda	AC9	US_V15.03.05.19	httpd	N/A	√	135
14	CVE-2018-8743	Tenda	AC9	US_V15.03.05.19	httpd	N/A	√	89
平均	/	/	/	/	/	/	/	80.6

(注: EDB 为 Exploit-DB 漏洞库中的漏洞标识; PSV 为 NETGEAR 厂商为自家产品中的漏洞设定的漏洞标识。第 7、8 列漏洞触发表示是否成功触发 1-day 漏洞。N/A 表示不支持对该设备进行漏洞挖掘。)

表 5 0-day 漏洞发现

Table 5 0-day vulnerability discovery

	厂商(型号)	TLD	漏洞类型	分配漏洞标识
1	Dlink(DIR825)	√	栈溢出	CVE-2018-9170
2	Tenda(AC9)	√	栈溢出	CVE-2018-9164
3	TP-Link(AC1900)	√	栈溢出	CNVD-2020-00019
4	Tenda(AC9)	√	栈溢出	*
5	Tenda(AC9)	√	栈溢出	*
6	Tenda(AC9)	√	栈溢出	*
7	Tenda(AC9)	√	栈溢出	*
8	Tenda(AC9)	√	堆溢出	*

(注: 第 3 列表头的 TLD 表示漏洞所在函数是否满足 TLD 模式。“*”表示厂商已经确认, 暂未分配标识的零日漏洞。)

动态分析方面需要真实的物联网设备或对物联网设备进行仿真。Chen 等人[30]提出 Firmadyne, 它是一款针对嵌入式设备固件的系统级仿真框架, 但是在我们的实践中, 其支持仿真的设备范围存在较大的局限性, 尤其受限于对 ARM 架构设备的支持。Zaddach 等[30]通过将硬件相关请求从模拟器重定向到真实设备, 提出了基于混合仿真的 Avatar 系统, 在此基础上利用符号执行技术来发现设备的漏洞。由于需要连接仿真器和实体设备, 该系统的时效性较差。Firmadyne 和 Avatar 均不支持对未知漏洞的挖掘。

在针对物联网设备的模糊测试方面, Chen 等[32]提出黑盒模糊测试方案 IoTFuzzer,该工具采用 APP 分析与模糊测试相结合的方式来挖掘物联网设备的漏洞, 但对于很多没有对应 APP 的物联网设备无法进行测试。该工具需要真实物理设备作为模糊测试目标且吞吐率不超过 1 个/s, 存在很大的性能问题。基于覆盖率的反馈式灰盒模糊测试[8-10,33,34]通过收集程序运行时的信息来指导测试用例的生成, 由于物联网设备固件的程序没有相应的源代码, 只有二进制层面上的代码可用于分析, 且程序执行的硬件依赖程度高, 所以上述几种灰盒模糊测试方法都无法直接应用于物联网设备的漏洞挖掘中。Zheng 等[26]提出的方法能够针对部分物联网设备进行灰盒模糊测试, 但由于物联网设备的异构性与多样性, 无法做到普适性的模糊测试, 仅能针对 Firmadyne 所支持的设备进行测试, 并且漏洞挖掘的效率不高。

6 总结

本文提出了一种基于物联网设备局部仿真的反馈式模糊测试方法, 解决了物联网设备动态执行信息难获取以及模糊测试固有测试深度问题。我们在物联网设备的固件代码中识别网络数据解析函数, 然后以该类函数为入口定位到网络服务组件, 并针对目标组件生成高质量的组件级种子样本集合, 进行局部仿真, 获取目标程序代码覆盖信息, 实现反馈式模糊测试。通过实验表明, 本文提出的方法平均可以达到 83.4%的函数识别精确率和 90.1%的召回率, 并在实际商业设备中发现了 8 个零日漏洞, 证明了本文方法的有效性和实用性。

参考文献

[1] State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating. https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-io. 2018.

[2] New trends in the world of IoT threats. https://securelist.com/new-trends-in-the-world-of-iot-threats/87991. 2018.

[3] Gartner. https://www.gartner.com/en.

[4] DEFCON. https://www.defcon.org.

[5] GeekPwn. http://www.geekpwn.org/en/index.html.

[6] Vulnerability Spotlight: Multiple Vulnerabilities in Samsung SmartThings Hub. https://blog.talosintelligence.com/2018/07/samsung-smartthings-vulns.html?m=1. 2018.

[7] Common Vulnerabilities and Exposures. https://cve.mitre.org.

[8] American fuzzy lop. M. Zalewski. http://lcamtuf.coredump.cx/afl.

[9] honggfuzz, a security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options. http:// honggfuzz.com/.

[10] libfuzzer-alibraryforcoverage-guidedfuzztesting. http://llvm.org/docs/LibFuzzer.html.

[11] Zhang Y, Huo W, Jian K P, et al. SRFuzzer: An Automatic Fuzzing Framework for Physical SOHO Router Devices to Discover Multi-Type Vulnerabilities[C]. *The 35th Annual Computer Security Applications Conference*, 2019: 544-556.

[12] China Nation Vulnerability Database. https://www.cnvd.org.cn.

[13] Luk C K, Cohn R, Muth R, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation[C]. *The 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005: 190-200.

[14] Nethercote N, Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation[C]. *The 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007: 89-100.

[15] Costin A, Zarras A, Francillon A. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces[C]. *The 11th ACM on Asia Conference on Computer and Communications Security*, 2016: 437-448.

[16] J. R. Quinlan. C4.5: Programs for Machine Learning[J]. *Morgan Kaufmann*, 1993, 16: 235-240.

[17] Bellard F. QEMU, a Fast and Portable Dynamic Translator[C]. *The annual conference on USENIX Annual Technical Conference*, 2005: 41.

[18] uClibc. https://uclibc.org.

[19] OTA. https://searchmobilecomputing.techtarget.com/definition/OTA-update-over-the-air-update.

[20] Binwalk. ReFirm Labs. Inc. https://github.com/ReFirmLabs/ binwalk.

[21] Firmware Mod Kit. Bitsum. https://bitsum.com/firmware_mod_kit.htm.

[22] Ida: About - hex-ray. https://www.hex-rays.com/products/ida/.

[23] Scikit-learn Machine Learning in Python. https://scikit-learn.org/.

[24] Ghidra. https://www.nsa.gov/resources/everyone/ghidra/.

[25] 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.

[26] Zheng Y W, Davanian A, Yin H, et al. FIRM-AFL: High-Throughput Greybox Fuzzing of Iot Firmware via Augmented Process Emulation[C]. *The 28th USENIX Conference on Security Symposium*, 2019: 1099-1114.

[27] Shoshitaishvili Y, Wang R Y, Hauser C, et al. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware[C]. *Proceedings 2015 Network and Distributed System Security Symposium*, 2015: 1-15.

[28] Costin A, Zaddach J, Francillon A, et al. A Large-Scale Analysis of the Security of Embedded Firmwares[C]. *The 23rd USENIX conference on Security Symposium*, 2014: 95-110.

[29] Cojocar L, Zaddach J, Verdult R, et al. PIE: parser identification in embedded systems[C]. *The 31st Annual Computer Security Applications Conference*, 2015: 1-10.

[30] 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.

[31] Zaddach J, Bruno L, Francillon A, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares[C]. *Proceedings 2014 Network and Distributed System Security Symposium*, 2014: 1-16.

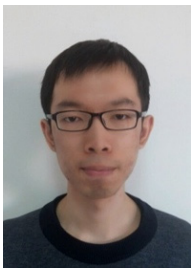
[32] Chen J Y, Diao W R, Zhao Q C, et al. IoTFuzzer: discovering memory corruptions in IoT through app-based fuzzing[C]. *Proceedings 2018 Network and Distributed System Security Symposium*, 2018: 1-15.

[33] Choronzon-anevolutionaryknowledge-basedfuzzer. https://github.com/CENSUS/choronzon.

[34] syzkaller-linuxsyscallfuzzer. https://github.com/google/syzkaller.

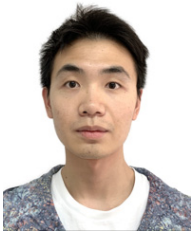
卢昊良 于 2017 年在吉林大学网络与信息安全专业获得学士学位。现在中国科学院信息工程研究所网络空间安全专业攻读硕士学位。研究领域为软件安全分析。研究兴趣包括: 漏洞挖掘、漏洞利用。Email: luhaoliang@iie.ac.cn

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



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

谭凌霄 于 2018 年在上海交通大学网络空间安全专业获得学士学位。现在中国科学院信息工程研究所攻读硕士学位。研究领域为模糊测试、Web 安全。Email: tanlingxiao@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。研究领域为嵌入式设备漏洞挖掘与利用。Email: zhangyu2@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

霍玮 博士, 研究员、博士生导师, 中国科学院青年创新促进会成员。2010 年在中国科学院计算技术研究所获得博士学位。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