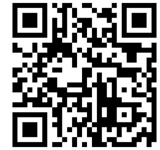
软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2024,35(8):3591−3609 [doi: 10.13328/j.cnki.jos.007117]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

FirmDep: 利用动态分析的嵌入式应用托管方案*



吴华茂[1], 姜木慧[2], 周亚金[1], 李金库[3]

[1](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

[2](香港理工大学 电子计算学系, 香港 999077)

[3](西安电子科技大学 网络与信息安全学院, 陕西 西安 710126)

通信作者: 周亚金, E-mail: yajin_zhou@zju.edu.cn

摘 要: 固件托管(firmware rehosting)是一种对嵌入式设备的软硬件进行建模和仿真, 并在仿真环境中运行和分析嵌入式设备软件的技术. 现有的基于全系统仿真的固件托管方案只能预防性地修复已知的软硬件依赖问题, 而无法解决未知的问题. 为应对这一现状, 提出了一种由动态分析辅助的固件托管方案 FirmDep. 在托管过程中, FirmDep 对被分析应用的执行轨迹和系统状态进行记录. 若目标应用无法被成功托管, FirmDep 对执行轨迹进行信息提取和系统状态补全, 并使用多种执行轨迹分析方法识别和仲裁应用的环境依赖问题. 基于 PANDA 和 angr 实现了 FirmDep 的原型系统, 并使用 217 个来自真实设备固件的嵌入式 Web 应用对其进行了测试. 结果表明: FirmDep 可有效识别嵌入式设备应用的环境依赖问题, 提高固件托管的成功率.

关键词: 嵌入式设备; 固件; 动态分析; 固件托管; 录制重放
中图法分类号: TP311

中文引用格式: 吴华茂, 姜木慧, 周亚金, 李金库. FirmDep: 利用动态分析的嵌入式应用托管方案. 软件学报, 2024, 35(8): 3591–3609. http://www.jos.org.cn/1000-9825/7117.htm

英文引用格式: Wu HM, Jiang MH, Zhou YJ, Li JK. FirmDep: Embedded Application Rehosting Assisted with Dynamic Analysis. Ruan Jian Xue Bao/Journal of Software, 2024, 35(8): 3591–3609 (in Chinese). http://www.jos.org.cn/1000-9825/7117.htm

FirmDep: Embedded Application Rehosting Assisted with Dynamic Analysis

WU Hua-Mao[1], JIANG Mu-Hui[2], ZHOU Ya-Jin[1], LI Jin-Ku[3]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

[2](Department of Computing, The Hong Kong Polytechnic University, Hong Kong 999077, China)

[3](School of Cyber Engineering, Xidian University, Xi’an 710126, China)

Abstract: Through providing a virtual environment modeled from embedded devices, firmware rehosting enables dynamic analysis on embedded device firmware. Existing full-emulation firmware hosting solutions can only preventatively fix known hardware and software dependencies but cannot address undetected dependencies during the rehosting process. This study proposes FirmDep, an embedded application rehosting solution assisted with dynamic analysis. During the rehosting process, FirmDep records the execution trace and system state of the embedded application to be analyzed. If FirmDep fails to rehost the application, FirmDep extracts information and recover system states from the execution trace, then uses several algorithms to identify and arbitrate the unresolved dependency problems. The prototype system of FirmDep is implemented based on PANDA and angr, and it is tested with embedded Web applications from 217 real-world firmware images. The results show that FirmDep can effectively identify unresolved dependencies of embedded application and improve the success rate of rehosting.

Key words: embedded device; firmware; dynamic analysis; firmware rehosting; record and replay

* 基金项目: 国家重点研发计划(2022YFE0113200); 国家自然科学基金重点项目(U21A20464)
本文由“系统与网络软件安全”专题特约编辑向剑文教授、陈厅教授、王浩宇教授、罗夏朴教授、杨珉教授推荐.
收稿时间: 2023-09-10; 修改时间: 2023-10-30; 采用时间: 2023-12-15; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-09

© 中国科学院软件研究所 http://www.jos.org.cn

近年来, 随着物联网技术的发展, 嵌入式设备的种类和数量都在逐步增长. 这些设备不仅被嵌入到各类复杂的系统中, 例如工控系统、汽车、飞机等, 而且还广泛应用于办公场所、公共区域以及个人家居中. 然而, 由于研发周期和成本限制等原因, 嵌入式设备的研发方常常未能为它们提供足够安全的软件. 根据 Yu 等人对嵌入式设备固件进行的大规模分析[1], 他们的数据集中仅有 29.7%的二进制文件包含了用于抵御栈溢出攻击的 Stack Canary, 尽管该安全特性早已被 GCC 等编译工具链支持并默认启用.

嵌入式设备, 尤其是物联网设备的软件安全问题已引起学术界和工业界的关注. 由于架构与性能差异、硬件交互能力不足以及普遍缺乏软件源码等原因, 研究者往往难以直接使用被广泛应用于桌面和服务器系统的安全分析方案分析这些设备, 而需要特别的方法或辅助手段[2]. 固件托管(firmware rehosting)即是一种用于支持动态分析的技术. 简单来说, 此种技术通过为嵌入式设备的固件提供仿真环境, 来实现在原硬件以外的环境运行嵌入式设备的软件. 在此基础上, 研究人员可以使用常见的动态分析手段, 例如应用漏洞扫描和模糊测试等对固件进行分析. 与直接测试真实设备相比, 此种方法不需要真实的设备硬件, 且在使用模糊测试等方法进行测试时, 不会受限于原设备的性能水平.

实现固件托管方案面临的最大挑战之一, 在于如何解决嵌入式设备固件的运行环境依赖问题. 运行于嵌入式设备的软件通常被设计为运行于特定的硬件平台环境, 并依赖特定的硬件外设. 若这些软件未能在与硬件交互时获取其期望的响应, 则将无法正常工作. 现有的固件托管方案主要通过两种方式来解决这一问题: (1) Firmadyne[3], FirmAE[4]等基于全系统仿真的固件托管方案通过尽可能收集并预防性地修复可能存在的硬件依赖问题(例如 NVRAM 闪存的访问等), 提供一个兼容性尽可能高的仿真运行环境; (2) Avatar[5], FIRMCORN[6]等由硬件辅助的固件托管方案则通过将采集自真实硬件的互动结果反馈到仿真环境中的方式, 解决环境依赖问题. 这两类方案各有其优劣性.

- 现有的基于全系统仿真的方案能支持较多嵌入式设备固件, 但此类方案只能解决一些已知的硬件依赖问题; 对于未被该方案支持的硬件依赖, 则需要用户手动解决. 由于嵌入式设备固件往往只提供二进制文件, 且为减少体积去除了符号等便于调试的要素, 因此, 对这些软件的修复通常需要耗费大量精力, 并要求用户具备足够的经验;
- 由硬件辅助的方案可为被分析的嵌入式固件提供更为真实的运行环境, 然而, 此类方案的硬件支持有限, 通常也需要用户对原始硬件有足够的了解, 难以应用到大规模固件分析等场景中.

基于当前的研究现状, 本文提出了一种名为 FirmDep 的固件托管方案. 现有的固件托管方案只对已知的软硬件问题制定了预防措施, 但缺乏发现和处理新问题的机制. 为识别嵌入式应用的环境依赖问题, FirmDep 使用了一种基于录制和回放的分析方法. 具体而言, FirmDep 在尝试托管嵌入式应用的同时记录该应用的执行, 以获取执行轨迹及用于还原仿真执行环境系统状态的执行回放. 如果该嵌入式应用未能被成功托管, FirmDep 将对托管过程中记录的信息进行提炼和补全, 使用执行轨迹分析算法从中提取出导致托管失败的错误根源, 并制定对应的仲裁方案. 此后, FirmDep 将应用仲裁方案, 再次尝试托管. 此过程将持续至目标应用被成功托管或没有更多的仲裁方案可用. 对于可直接顺利托管或直接修复的嵌入式应用, FirmDep 将为其生成一个虚拟机快照, 并在此后的用户使用过程中禁用对仿真执行环境的记录或干涉, 消除对用户的安全分析应用性能的影响.

为实现上述托管方案, 需要解决的问题包括: (1) 如何有效且高效地记录被托管嵌入式应用的行为; (2) 如何从记录的信息中提取应用的行为及其还原系统状态以供分析; (3) 如何从被托管的执行轨迹中识别导致托管失败的故障根源; (4) 如何对被托管应用的环境依赖问题进行仲裁. 针对问题(1), FirmDep 采用了一种按需记录的方式来记录应用的执行轨迹, 并为托管过程生成可用于还原任意时刻系统状态的执行回放. 针对问题(2), FirmDep 可基于执行轨迹生成函数调用记录以实现应用高级语义行为的提取, 且可配合执行回放还原执行轨迹中每个基本块前后的系统状态. 针对问题(3), 本文为 FirmDep 实现了 4 种执行轨迹分析算法, 以识别目标应用在不同运行状态的错误根源. 针对问题(4), FirmDep 的仿真执行环境提供了干预客户机执行的能力, 除了调整网络配置、执行自定义指令等仲裁方案外, 还支持通过改变系统状态来影响目标应用的执行.

本文基于 PANDA 与 angr 实现了适用于 32 位 ARM 平台固件的 FirmDep 原型, 并使用了来自 8 个厂商的 217 个设备固件对 FirmDep 的有效性进行了评估. 根据实验结果, FirmDep 在应用了仲裁措施后, 成功托管了 147 个(67.7%)嵌入式 Web 应用, 优于基于现有经验的现有方案 FirmAE(托管 113 个, 成功率 52.1%)以及 Firmadyne (托管 1 个, 成功率 0.4%). 在此过程中, FirmDep 识别出了 95 个嵌入式 Web 应用的环境依赖问题, 并成功修复了其中的 81 个, 有效提高了托管成功率. 为验证 FirmDep 是否适用于安全分析场景, 本文对成功托管的嵌入式 Web 应用使用 w3af 和 RouterSploit 进行了漏洞扫描测试, 分别检测出 69 个和 9 个有安全隐患的 Web 应用. 此外, 对于未能托管的 Web 应用, 本文还分析了托管失败的原因.

本文第 1 节介绍本工作的背景知识. 第 2 节介绍 FirmDep 固件托管方案的整体设计. 第 3 节和第 4 节分别介绍 FirmDep 的框架设计及分析算法设计. 第 5 节介绍 FirmDep 框架的一些实现细节. 第 6 节对 FirmDep 的有效性进行评估. 第 7 节讨论本文提出方案的不足之处和可能的改进空间. 最后总结全文.

1 相关工作及研究动机

1.1 相关工作

固件托管是一种围绕某个固件镜像, 通过仿真运行所必需的硬件实现固件在虚拟环境中高效运行的技术. 根据 Muench 等人的定义[7], 嵌入式设备大致可被分为以下 3 类: (1) I 型设备, 通常配备带有 MMU 的 SoC 封装的处理器, 运行通用操作系统(如 Linux), 应用通过设备驱动与硬件交互; (2) II 型设备, 此类设备通常配备无 MMU 的 MCU 处理器, 运行专用操作系统(如 VxWorks), 应用程序与软件交互仍然有一层隔离; (3) III 型设备, 此类设备不具备操作系统. 此 3 类设备由于软件、硬件复杂度不同, 因此需要通过不同的方式实现固件托管.

本文主要关注运行于 I 型设备上的嵌入式应用的托管. 2016 年, Costin 等人[8]通过网络爬虫收集了大量嵌入式设备的固件, 通过使用 Qemu 和 chroot 创建较为基础的仿真执行环境, 实现了对嵌入式设备包含的 Web 服务的托管. 此种方案通过使用全系统仿真解决了嵌入式设备软件对指令集的依赖问题, 但却未解决硬件外设交互等依赖问题, 因此托管成功率并不高. 同年, Chen 等人也提出了一种针对嵌入式设备固件的大规模固件托管方案 Firmadyne[3], 该方案通过对关键系统调用及个别 API 函数调用进行拦截, 缓解了嵌入式设备软件对网络配置及 NVRAM 硬件的依赖问题. 一些研究在 Firmadyne 的基础上实现了多种分析平台, 例如用于对嵌入式设备软件进行模糊测试的 FirmAFL[9], FirmFuzz[10]平台等. 为进一步缓解 Firmadyne 无法解决的软硬件依赖问题, FirmAE[4]提出了仲裁式仿真(arbitrated emulation)这一概念, 即通过干涉嵌入式应用程序与系统、硬件的交互来解决其软硬件依赖问题. FirmAE 分析了 Firmadyne 托管失败的原因, 并通过加入一些额外的修复, 进一步提升了固件托管的成功率.

除了从软件层面采取措施外, 一些固件托管方案选择通过优化固件托管过程的硬件仿真来解决嵌入式设备固件的环境依赖问题. 此类方案可被进一步分为两类.

- 一类方案通过分析嵌入式设备固件和与其交互的物理外设的联系, 对其依赖的硬件外设进行抽象和建模, 生成用于替代真实硬件的软件组件. 例如: HALucinator[11]通过劫持嵌入式设备固件对硬件抽象层(hardware abstraction layer)的 API 调用来解决固件的外设依赖问题, P2IM[12]通过对 MMIO 操作进行建模来仿真通过 MMIO 连接的外设, FirmGuide[13]和 ECMO[14]分别通过为仿真器生成和替换外设的方式实现 I 型设备 Linux 内核的托管;
- 另一类方案在托管的流程中加入真实硬件的辅助, 当嵌入式设备软件需要与硬件发生交互时, 通过采集于硬件设备的响应并反馈给嵌入式设备软件来解决这些软件的环境依赖问题. 例如: Avatar[5]通过使用 UART 和 JTAG 总线将仿真器无法处理的硬件 I/O 请求转发到真实硬件上处理, 并将结果反馈到仿真器中; Pretender[15]则基于嵌入式设备固件与物理外设之间的交互记录通过机器学习为每个外设训练一个外设模型, 并将该模型应用到仿真器中. 此类方案并不局限于 II 类设备, 例如: FIRMCORN[6]通过同步 CPU 模拟器与真实硬件上的软件的上下文状态, 实现对 I 型设备嵌入式设备

© 中国科学院软件研究所 http://www.jos.org.cn

软件的高效模糊测试.

1.2 研究动机

实现一个有效的固件托管方案的关键在于解决嵌入式设备固件的环境依赖问题. 现有的基于全系统仿真的托管方案有一个共同的不足之处: 嵌入式设备固件的运行环境依赖问题需要被固件托管框架的开发者手动识别和解决. 若托管框架中未集成某个依赖问题的仲裁措施, 则固件的托管很可能会因此失败. 由于嵌入式设备固件往往难以获得源码, 且为了减小体积常常还会去掉符号表等信息, 修复运行于嵌入式设备的软件往往很困难, 需要研究者对设备有充分的了解. 以常见的 NVRAM 依赖问题为例, 为了减少对设备闪存的磨损, 设备厂商常常会选择将配置文件信息存放于 NVRAM 闪存中, 由于不同设备的 NVRAM 硬件均有区别且通常不提供相关参数资料, Qemu 等仿真器很难对其进行仿真. 现有的全系统仿真方案对该问题的应对方式是分析用于与 NVRAM 交互的库文件, 并使用库函数劫持来处理嵌入式设备软件对 NVRAM 的访问. 若固件托管框架未支持某个 NVRAM 库提供的 API, 用户需要推断该 API 函数的行为并推断出返回值, 并通过劫持该 API 函数或者是改写嵌入式设备软件的二进制文件来解决. 毫无疑问, 这一过程具有门槛且耗时耗力.

与基于全系统仿真的固件托管方案相比, 由硬件辅助的固件托管方案使用来自真实硬件的互动反馈来应对固件与硬件的交互行为, 在根源上解决了固件的硬件依赖问题. 然而, 当前大多数此类方案都只适用于 II 型和 III 型嵌入式设备的固件托管. 像 FIRMCORN 这类方案虽然实现了对 I 型设备的软件托管, 但是只适用于用户具备同一型号硬件设备的情况, 且需要对被分析的软件有足够的了解, 限制了此类方案的应用.

2 方案设计

本文提出了一种使用研究 I 型设备被托管嵌入式设备应用行为的方法, 并基于此方法实现了一个名为 FirmDep 的嵌入式应用托管方案. 针对第 1.2 节中阐述的托管嵌入式设备固件的挑战, FirmDep 记录被托管应用的行为, 并通过多种执行轨迹分析算法识别并仲裁目标应用被托管过程中遇到的环境依赖问题.

FirmDep 包含两个主要部件: 仿真执行环境和执行轨迹分析器, 其工作流程如图 1 所示. FirmDep 采用基于 chroot 的固件托管方案, 在基于固件文件系统的仿真环境中使用预处理阶段提取出的启动参数运行需要被分析的嵌入式应用. 为识别和解决目标应用潜在的环境依赖问题, 仿真执行环境在对嵌入式应用进行托管的同时, 对目标应用的执行进行记录. 若目标应用被顺利托管且可被宿主机访问, 仿真执行环境将生成一个快照, 以便此后进行模糊测试等安全分析应用; 若目标应用未能被成功托管, FirmDep 将其执行轨迹传至执行轨迹分析器, 并使用多种执行轨迹分析算法识别出环境依赖问题并提出仲裁方案. 在此过程中, 执行轨迹分析器依照算法的需求, 从目标应用的托管过程进行还原和提炼. 此后, 仿真执行环境将应用仲裁措施, 并再次尝试托管目标应用. 这一过程将持续至目标应用被成功托管或无更多的仲裁措施可用.

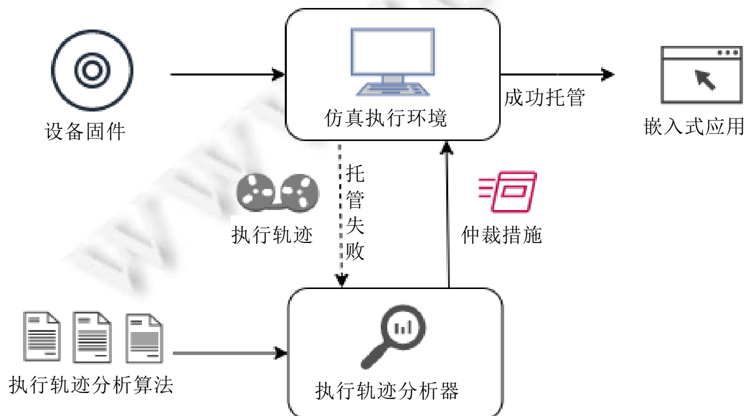


图 1 FirmDep 的工作流程

© 中国科学院软件研究所 http://www.jos.org.cn

运行在 I 型设备上的应用通常使用系统调用或动态链接库提供的 API 函数与环境交互, 而非直接操作硬件. 因此, 它的环境依赖问题通常会表现为异常返回的外部函数调用或系统调用. 此外, 当被托管的应用遇到环境依赖问题时, 其错误处理机制可能会导致程序进入不同于正常工作路径的执行路径. 本文针对可能出现的异常执行状态制定了不同的分析算法, 用于在执行轨迹中寻找可归咎的函数调用, 并将其作为应用托管失败的根源.

要实现上述的嵌入式应用托管方案, 需要解决两个问题: (1) 如何高效记录嵌入式应用托管过程, 并在执行轨迹分析过程中还原应用运行时的系统状态; (2) 如何从应用的执行轨迹中识别出其遭遇的环境依赖问题. 下文通过介绍 FirmDep 的部件实现和执行轨迹分析算法来解答这两个问题.

3 FirmDep 框架设计

3.1 仿真执行

FirmDep 的仿真执行环境需要满足 3 个功能需求: (1) 提供嵌入式应用的仿真执行环境; (2) 记录目标应用的执行过程; (3) 提供应用托管过程中需要的仲裁措施, 以缓解环境依赖问题. FirmDep 使用 PANDA[16]作为嵌入式应用的仿真执行环境, 该框架不仅可用于仿真嵌入式硬件环境, 还提供了对客户机系统进行动态插桩的能力. FirmDep 在仿真执行环境的仿真硬件层和运行时层分别加载了额外的组件, 以获取执行轨迹和分析环境依赖问题所需的其他信息. FirmDep 的仿真执行环境系统结构如图 2 所示.

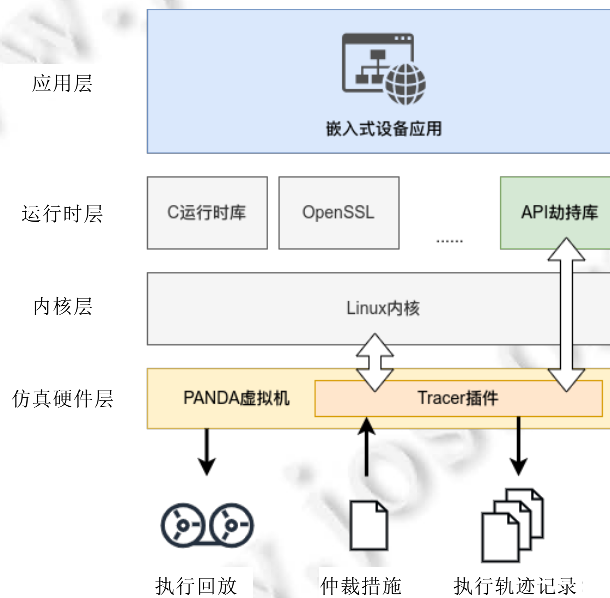


图 2 FirmDep 仿真执行环境的系统结构

PANDA[16]是一个基于 Qemu[17]的全系统动态分析引擎, 其为用户提供了两种层次的记录客户机内部执行的能力.

- 首先, PANDA 提供了录制/重放的功能, 此功能起源于 Mozilla 的 rr 调试器项目[18], 该功能允许用户录制程序在一段时间内的执行及对该回放进行重放, 以方便对程序进行调试, 其录制的原理是: 以一个包括完整寄存器及内存状态的虚拟机快照为起点, 在客户机运行过程中记录录制期间执行过的基本块数量及发生的非决定性输入, 此类输入包括硬件中断、键盘输入和网络包等; 重放则是通过加载快照, 并精确重放所有非决定性事件来实现. 在重放过程中, 若用户对回放进行暂停, 即还原了回放记录当前点的系统状态;

© 中国科学院软件研究所 http://www.jos.org.cn

- 此外, PANDA 提供了一系列的事件钩子, 用户通过使用现成的插件或者执行编写插件并注册对应的事件钩子在执行或重放过程中捕获、记录特定事件或者是获取当前系统状态.

为了提取嵌入式设备应用的高级语义行为并从中找出导致托管失败的环境依赖问题, 仿真执行环境记录的执行轨迹需要满足两个需求: (1) 该记录需要包含足够的信息来还原函数调用记录, 以便解析目标应用与外界的互动; (2) 为了了解目标应用与外界交互的内容和结果, 该记录需要提供还原任意点系统状态的能力, 以满足对函数调用参数和返回值进行解析的需求. 若单独使用 PANDA 的两种记录客户机执行的方法, 均不能完全满足这些要求: 首先, PANDA 的录制/重放功能并不直接记录应用的执行, 而且受限于其实现原理, 重放过程只能按顺序从头开始进行, 无法直接读取任意点的状态; 如果使用事件钩子记录托管过程中嵌入式应用每个时刻的完整系统状态, 则会带来巨大的性能开销和空间占用. 因此, 本文为 PANDA 实现了一个名为 Tracer 的插件, 使用结合了这两种方法的记录机制来记录嵌入式应用的托管过程. 具体来说, 当 Tracer 检测到与目标应用相匹配的新线程时, 它调用 PANDA 的录制/重放功能开始对托管过程进行录制, 生成执行回放文件. 当目标应用运行结束, Tracer 播放执行回放文件并注册事件钩子, 在目标应用的每个基本块执行前和执行后记录执行指令计数和当前系统寄存器状态, 生成基本块粒度的执行轨迹记录. 通过配合使用执行轨迹记录和执行回放文件, 使用者可以还原执行轨迹记录中任意点的系统状态.

尽管在托管阶段不需要记录完整的系统状态, 但由于应用运行时可能需要执行大量基本块, 因此记录完整执行轨迹仍会为仿真和分析带来很大的性能开销. 此外, 由于执行回放无法实现乱序读取, 在分析阶段从执行回放中提取信息可能有困难. 为了缓解这些问题, FirmDep 使用一种按需记录的机制, 在执行轨迹生成阶段先对基本块进行过滤, 并获取一些基本块记录以外的粗粒度信息, 以减少对执行回放的需求. 为了提高记录的效率和精度的平衡, FirmDep 默认不生成目标应用所加载的外部链接库的执行轨迹, 仅记录应用二进制文件本身的基本块执行记录和应用的内存布局. 这种按需记录方式足以还原目标应用的内部运行及外部 API 函数的调用行为, 大大减少了记录的基本块数量. 此外, FirmDep 通过预加载共享链接库拦截常见的 POSIX API 函数和关键系统调用, 直接获取目标应用调用这些函数的返回结果、错误代码(errno)等信息, 以减少在执行轨迹分析阶段对这些信息的获取需求, 提高分析效率.

除了具备记录客户机系统运行的能力, PANDA 还提供了对客户机运行的干预能力, FirmDep 利用 PANDA 的这些能力对被托管的嵌入式应用进行仲裁. 具体来说, FirmDep 支持在托管目标应用之前调整网络设置和执行自定义命令. 此外, FirmDep 支持一种称为"强制执行"的仲裁措施, 以处理某些难以解决的外部 API 调用带来的问题. 在托管目标应用的过程中, 当系统状态满足一定条件时(例如即将执行某条指令), Tracer 将修改特定寄存器或特定内存地址的值, 以改变应用的执行路径.

3.2 执行轨迹分析

如果仿真执行环境无法成功托管嵌入式应用, FirmDep 的执行轨迹分析器将根据执行轨迹分析算法的要求对仿真执行环境产生的执行轨迹进行处理, 并应用执行轨迹分析算法来查找目标应用的环境依赖问题. 具体来说, 执行轨迹分析器首先在基本块粒度的执行轨迹上提取函数调用粒度的执行轨迹, 以满足用户或者轨迹分析算法对提取应用高级语义的需求. 此外, 在仿真执行轨迹中收集的系统状态信息仅包括执行基本块前后的寄存器状态, 无法满足某些执行轨迹分析算法的需求. 针对这些需求, 执行轨迹分析器通过重放托管过程的执行回放, 来为执行轨迹中的每个基本块补全系统状态信息.

3.2.1 提取函数调用记录

为从执行轨迹中提取函数调用记录, 执行轨迹分析器对执行轨迹中的基本块进行解析, 对于每个基本块, 识别其执行前是否返回自某个函数调用, 以及其最终是否进行了函数调用. 对于 ARM 等主要使用寄存器传递参数及返回值的架构而言, 通过配合分析基本块代码的行为及基本块执行前后的寄存器状态, 即可获得每个函数调用所传递的参数指针及返回值.

FirmDep 使用 angr[19]完成执行轨迹中基本块的解析. angr 是一个二进制文件的分析平台, 支持对多种硬件架构、多种格式的二进制文件的加载和反汇编、将二进制文件转换为易于分析的中间语言形式、动态符号执

行及控制流图、反向切片、约束求解等多种静态分析功能. 为实现对执行轨迹基本块的解析, 执行轨迹分析器首先使用angr的CLE加载器加载被分析应用的二进制文件, 并根据仿真执行环境记录的内存布局将目标应用所依赖的动态链接库加载到被分析应用的内存模型中. 然后, 执行轨迹分析器通过调用 angr 的 PyVEX 模块, 将执行轨迹中记录的基本块转换为 VEX IR[20], 以确定每个基本块的出口类型. 最后, 从中提取函数调用记录.

算法 1 展示了针对 32 位 ARM 架构的函数调用记录的提取过程. 其中, *BBTrace* 为基本块粒度的执行轨迹, 每个基本块 *BB* 由 3 个部分组成: 基本块执行前的系统状态 *BeforeState*、执行后的系统状态 *AfterState* 及由基本块的二进制汇编翻译而来的 VEX IR 超级块. *BeforeState* 和 *AfterState* 均包含当前的执行指令计数及通用寄存器(*R*0–*R*15)的状态. 此外, 算法 1 中的 *CallTrace* 为函数调用粒度的执行轨迹, 其中的每一项 *CallRecord* 由 3 个部分组成: 函数调用发生时的系统状态 *CallState*、返回时的系统状态 *ReturnState* 和用于确认函数调用间包含关系的栈深度值 *depth*. 算法 1 维护了一个影子堆栈 *ShadowStack*, 每当基本块中包含返回或者是调用的指令, 即更新其内容. 根据 32 位 ARM 架构的函数调用规范, 当程序调用函数 *f* 时, 该基本块执行结束时 *PC* 寄存器(*R*15)值为函数 *f* 的入口地址, *LR* 寄存器(*R*14)值为返回地址. 当一个二进制基本块被转换为 VEX IR 超级块类型时, 其 *jumpkind* 属性指示了该基本块的出口类型. 若基本块包含对函数调用的发起, 其内容被转换为 VEX IR 后, 出口类型为 *Ijk_Call*; 若基本块包含对系统调用的发起, 基本块结束时, *R*7 寄存器将用于存放系统调用号, 出口类型为 *Ijk_Sys_syscall*. 根据以上原则, 即可判断某个基本块是否包含了函数调用.

算法 1. 从基本块粒度执行轨迹中提取函数调用记录.

Input: BBTrace——基本块粒度的执行轨迹;

Output: CallTrace——函数调用记录.

1. *ShadowStack*←∅, *CallTrace*←∅, *Depth*←0;
2. **foreach** *BB*∈*BBTrace* **do**
3.   **if** *ShadowStack*≠∅ **then**
4.   **if** BB.*BeforeState*.*PC*==*ShadowStack*[*last*] **then**
5.   *CallTrace*[*last*].*ReturnState*←*BB*.*BeforeState*;
6.   *ShadowStack*.*pop*(·);
7.   *Depth*−−;
8.   **end if**
9.   **end if**
10.   **if** *BB*.*jumpkind*∈{*Ijk_Call*,*Ijk_Sys_syscall*} **then**
11.   *Depth*++;
12.   *NewCallRecord*.*CallState*←*BB*.*AfterState*;
13.   *NewCallRecord*.*Depth*←*Depth*;
14.   *CallTrace*.*append*(*NewCallRecord*);
15.   *ShadowStack*.*append*(*BB*.*AfterState*.*LR*);
16.   **end if**
17. **end for**

得到函数调用记录后, 执行轨迹分析器通过解析每个函数调用的目标地址、参数和返回地址来确定目标应用在运行过程中执行了哪些函数调用以及对应的返回结果. 函数调用记录的调用前/后系统状态信息可用于粗略呈现函数调用的参数和返回值, 其包含的回放计数信息则用于在之后的分析过程中控制回放进度, 还原仿真执行环境的完整系统状态. 此外, 用户可以通过预览函数调用记录中各个函数的执行长度, 排除执行轨迹中不感兴趣的部分, 以减少执行轨迹分析所需的时间. 图3 展示了一个嵌入式 Web 应用的函数调用记录, 从中可以看到, 该应用在退出之前执行了一个 *open*64(·)函数调用且该操作失败了.

© 中国科学院软件研究所 http://www.jos.org.cn

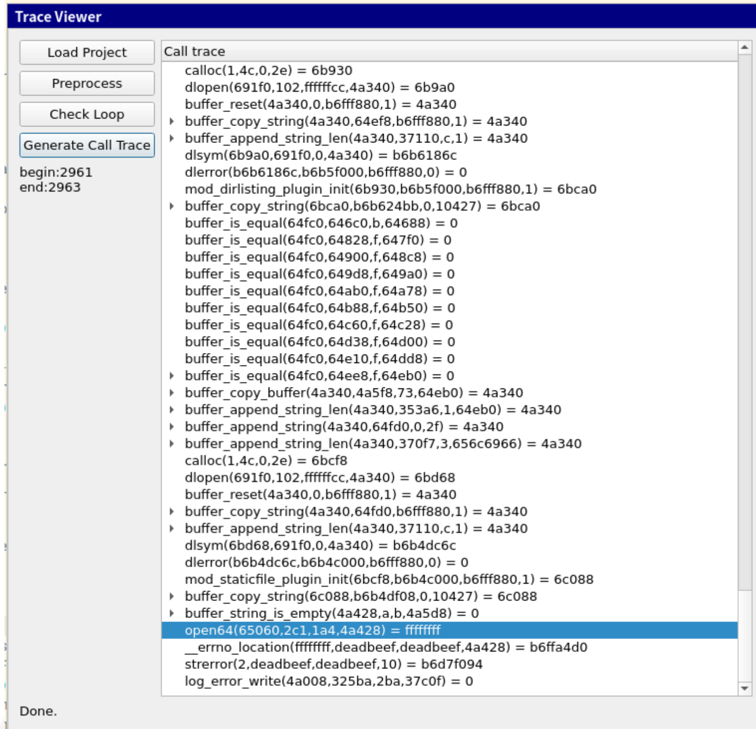


图 3　执行轨迹分析器提取的函数调用记录示例

3.2.2　填补系统状态信息

为了解析函数调用的指针型参数和返回值内容，执行轨迹分析器需能够在函数调用及返回时获取仿真执行环境对应内存地址的内容．此外，以基本块执行为单位的系统状态取样可能不足以满足某些执行轨迹分析算法的需求．例如：一些基本块中可能存在条件读或条件写的行为，如果执行轨迹分析算法需要分析污点传播，则需要获取该基本块执行过程中的具体内存操作和条件指令选择．为满足获取更健全系统状态信息的需求，执行轨迹分析器提供了为执行轨迹中的基本块填补系统状态信息的能力．

FirmDep 的执行轨迹分析器通过使用模拟执行来补全执行轨迹分析所需的系统状态信息，并还原基本块内部的具体内存操作和条件指令选择．具体来说，对于执行轨迹中的每个基本块，执行轨迹分析器将该基本块执行前的寄存器状态放入 angr 的 VEX 执行引擎中，并对该基本块进行模拟执行．同时，执行轨迹分析器使用 PANDA 重放目标应用的托管过程，并将回放进度与模拟执行的进度同步．在模拟执行过程中，执行轨迹分析器记录基本块中的所有条件操作和内存操作．若应用有读取内存的行为，执行轨迹分析器将使用 PANDA 获取指定内存地址的内容，并将其填充到模拟执行引擎的内存模型中．当执行完基本块后，其对应的内存模型将被用作下一个基本块的初始内存模型．通过这种方式，FirmDep 为执行轨迹中的每个基本块获取足够用于分析应用行为的内存状态等信息．

图 4 为一个运行于 FirmDep 的执行轨迹分析算法的运行结果，该算法用于基于执行轨迹生成带有参数解析的文本格式的函数调用记录．如图所示：目标应用绑定了一系列的信号处理函数，创建了一个套接字并尝试将该套接字绑定到某个网络地址．该算法通过对 *bind*(·)函数调用的 *sockaddr* 格式参数进行解析，得到了其尝试托管时尝试绑定的地址及端口．最终，该 *bind*(·)函数未能将套接字与该地址绑定，应用输出了一条错误信息并退出．

© 中国科学院软件研究所 http://www.jos.org.cn

```
In [6]: p.gen_callgraph()
[!] note: you should only use this after pass_2.
sub_9ef0(0xbefffcf8,0xbefffe44,0xbefffe4c,0xb6fdf408)
| +--atoi(8080) = 8080
| +--strcmp(192.168.2.1,0.0.0.0) = 1
| +--inet_aton(192.168.2.1,0xbefff8d4) = 1
+-- = 0
getcwd(0x15ce0,0x100,0x200,0x1f) = 89312
signal(0xd,0x1,0x200,0x1f) = 0
signal(0x2,0x9d54,0xbefff8b0,0x8) = 1
signal(0x1,0x9d54,0xbefff8b0,0x8) = 0
signal(0xf,0x9d54,0xbefff8b0,0x8) = 0
socket(2,1,0) = 3
fcntl((3, '<new socket>'),2) = 0
setsockopt((3, '<new socket>'),1,2,0xbefffd8c,4) = 0
bind((3, '<new socket>'),('AF_INET', '192.168.2.1', 8080),16) = 4294967295
perror(bind) = 38
fprintf(0xb6fdb178,%s:can't bind to any address) = 35
__errno_location() = 3070079920
exit()
```

图 4　执行轨迹分析算法示例: 带参数解析的函数调用记录生成

4　执行轨迹分析算法设计

在提取函数调用记录的过程中, 执行轨迹分析器将对被托管应用的最终运行状态进行简单的确认. 对未能被成功托管的嵌入式应用而言, 其最终运行结果可分为以下几类: 1) 阻塞(执行轨迹结束于未返回的函数调用); 2) 进入循环; 3) 主动结束运行; 4) 因段错误等原因被终止运行. 针对应用不同的运行状态, FirmDep 使用不同的分析策略来识别环境依赖问题, 并提供不同的仲裁方案.

- 针对阻塞状态的分析策略

此种策略将该函数最后执行且未返回的函数调用视作是目标应用托管失败的直接原因. 此函数调用的相关信息可以通过获取函数调用记录中的最后一项得到. 依照最后一个函数调用的情况, FirmDep 会采取对应的措施.

1) 结束于某个未知语义的外部函数调用: 由于在生成初始执行轨迹时默认不记录嵌入式应用程序所加载的动态链接库的执行, 因此当程序异常源于其加载的动态链接库时, 记录信息是不足的. 在这种情况下, FirmDep 会将该外部函数所属的动态链接库的地址范围添加到录制执行轨迹的地址范围内, 并通过回放执行以获得新的执行轨迹;
2) 结束于 *select*(·), *poll*(·)等已知语义的函数调用: FirmDep 将根据该函数调用的参数, 并给出调整仿真执行环境参数的仲裁措施. 对于 *select*(·), *poll*(·)等用于等待请求的函数而言, FirmDep 将通过系统调用记录获取其关联的套接字文件节点信息, 并将该信息提供给用户.

- 针对进入循环及主动结束运行的分析策略

由于存在错误处理机制, 当嵌入式应用程序遭遇错误时, 可能会选择一条不同于正常执行的执行路径. 图 5 展示了两种常见的错误处理机制的具体实现. 图中的 3 栏分别为基于原始二进制生成的伪代码、汇编代码及对应的 VEX IR 中间代码. 在图 5(a)所展示的例子中: 如果应用执行 *listen*(·)获得错误结果, 程序将输出一条错误信息并跳转至清理退出的代码. 这种机制通常用于处理致命性的错误. 另一种机制是对失败的操作进行重试, 直到满足某种条件. 以图 5(b)中展示的代码为例, 该应用将不断重试执行 *check_network*(·), 直到获得一个正值返回. 这两种情况有一个共同点: 都包含由函数返回值决定的执行分支. 在图 5 所展示的两个例子中, 决定程序分支的临时变量值均来自寄存器 $r0$ 中的值. 根据 32 位 ARM 的函数调用规范, 该值正是某个函数调用的返回值.

基于这一观察, 本文提出了一种定位故障根源的思路: 寻找最后一个影响分支跳转且返回值为错误值的函数调用. 为了定位这样的函数调用, 本文进一步提出了一种反向污点分析算法. 首先, 从执行轨迹的最后一个基本块开始, 寻找一个以分支跳转为结束的基本块; 然后, 将用于决定该分支跳转目标的临时变量值(如图

© 中国科学院软件研究所 http://www.jos.org.cn

5(a)中的 $t14$ 和图 5(b)中的 $t17$)作为污点传播源. 对于每一个基本块, 从包含该临时变量值的最后一个指令开始, 将污点传播至所有参与计算现有污点元素的每个临时变量、寄存器及内存地址. 由于决定程序运行状态的值是来自于程序外部, 因此在传播过程中, 若一个污点元素的值完全由常量计算得来, 我们将清除其污点状态以减少假阳性污点. 该污点传播的过程将持续至执行轨迹的起点, 或者是当污点传播到某个函数的返回值时. 此时, 若该函数的返回值为一个代表错误的值, 则将该函数调用作为一个候选的错误根源返回给用户. 为判断函数返回值是否代表错误值, FirmDep 维护了 270 个常见 POSIX 函数的返回值范围. 对于未维护的函数, FirmDep 将小于 0 的返回值视作是代表错误的返回值. 在获取一个候选的错误根源后, FirmDep 将去除用于传参的寄存器的污点状态, 并继续进行污点分析直至执行轨迹的开始.

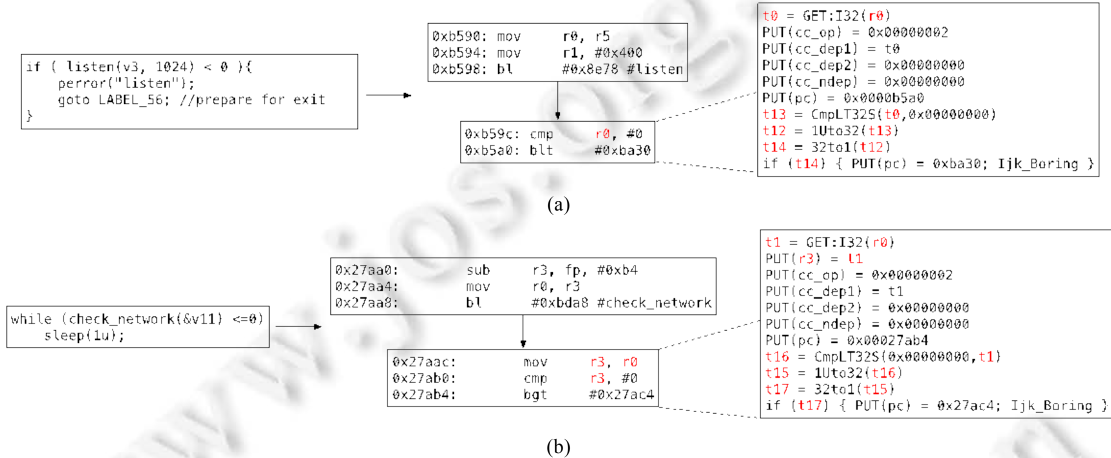


图 5　常见的程序错误处理机制及反向污点分析示例

算法 2 阐述了基本块内部的污点传播过程. 在 VEX IR 中, 每条指令都被分解为多条声明(stmt), 而每条声明都具备一个写入目标(stmt.target)以及即将被写入该目标的表达式(stmt.expr). 反向污点传播的过程可以被简单概括为: 若该声明中的写入目标已被污点标记, 则将污点传播至该声明的表达式中. 已经被污点标记的寄存器及内存地址的污点状态将延续至下一个被分析的基本块.

算法 2. 基本块内部的污点传播.

Input: *bb*——基本块中包含的 VEX IR 语句序列;

tainted_addrs——被污点标记的寄存器序号及内存地址;

source_tmp——污点传播源;

Output: *tainted_addrs*——被污点标记的寄存器序号及内存地址.

1. $tainted_tmps \leftarrow \varnothing$, $uncovered \leftarrow [0,1,\ldots,|bb|-1]$;
2. $tainted_tmps.add(source_tmp)$;
3. **while** $|\underline{uncovered}| \geq 0$ **do**
4. 　$last \leftarrow uncovered.pop_last(\cdot)$;
5. 　$stmt \leftarrow bb[last]$;
6. 　**if** $\underline{stmt.target \in tainted_tmps}$ **then**
7. 　　**if** $\underline{stmt.expr \cap tainted_\{tmps,addrs\} \neq \varnothing}$ **then**
8. 　　　**for** $\underline{t \in stmt.expr.tmps}$ **do**
9. 　　　　$tainted_tmps.add(t)$;
10. 　　　**end for**
11. 　　　**for** $\underline{a \in stmt.expr.addrs}$ **do**

© 中国科学院软件研究所 http://www.jos.org.cn

```
12.             tainted_addrs.add(a);
13.         end for
14.     else
15.         tainted_{tmps,addrs}.discard(stmt.target);
16.     end if
17.   else if stmt.target∈tainted_addrs then
18.     if stmt.expr 包含至少一个临时变量 then
19.         for t∈stmt.expr.tmps do
20.             tainted_tmps.add(t)
21.         end for
22.     else
23.         tainted_addrs.discard(stmt.target);
24.     end if
25.   end if
26. end while
```

对于通过此分析策略寻找到的托管失败问题, FirmDep 根据具体情况, 制定两种类型的仲裁方案.

(1) 若找到的 API 函数调用为已知的函数调用, 且能根据其参数、返回值和 errno 等获知导致问题的原因, 则提出调整仿真环境的对应仲裁方案;

(2) 若找到的函数调用为未知的函数调用, FirmDep 将尝试通过使用 angr 的约束求解器, 求解出一个可使程序选择不同分支的函数返回值, 并让仿真执行环境在下一次尝试进行托管时, 使用强制执行对该问题进行修复.

图 6 展示了使用此算法寻找应用托管失败原因的例子. FirmDep 通过污点分析算法找到一个影响了程序分支的 *bind*(·)函数调用, 并给出了通过调整仿真环境网络配置的仲裁方案.

```
In [8]: p.process(bbs)
[+] processing bb 61, tainted_regs = set()
[+] processing bb 60, tainted_regs = set()
[+] processing bb 59, tainted_regs = set()
[+] breakpoint here, just a notice!
[+] processing bb 58, tainted_regs = set()
[+] processing bb 57, tainted_regs = set()
[+] processing bb 56, tainted_regs = set()
[+] breakpoint here, just a notice!
[+] processing bb 55, tainted_regs = set()
[+] processing bb 54, tainted_regs = set()
[+] processing bb 53, tainted_regs = set()
[+] processing bb 52, tainted_regs = set()
[+] breakpoint here, just a notice!
[+] processing bb 51, tainted_regs = set()
[+] processing bb 50, tainted_regs = set()
[+] processing bb 49, tainted_regs = set()
[+] processing bb 48, tainted_regs = set()
[+] possible root cause found, tainted_regs = {8}
[+] bind((3, '<new socket>'),('AF_INET', '192.168.2.1', 8080),16) = 0xffffffff
[+] we think this is the root cause...
[+] suggestion for bind(): alter the network configuration
[+] ('vmarg', '-net', 'user,net=192.168.2.0/24,dhcpstart=192.168.2.1,hostfwd=tcp::10080-:8080', None)
```

图 6 通过反向污点分析算法寻找应用托管失败原因的示例

- 针对段错误的分析策略

当嵌入式应用在托管过程中对无效的指针进行操作时, 其会触发段异常并被系统终止运行. 该无效的指针可能是一个在仿真执行环境中不存在的物理地址, 或者是由于用于与硬件交互的 API 函数未能读取到正确的值而返回的空指针. FirmDep 通过一种对该指针进行溯源的策略来解决段错误的问题. 具体而言, 此分析策略检查最后触发段错误的函数调用是否存在空指针或无效指针参数: 若存在, 则以该参数作为污点传播源对

© 中国科学院软件研究所 http://www.jos.org.cn

执行轨迹进行反向污点分析, 并寻找产生该无效指针的函数调用并呈现给用户. 若该函数调用为某一用于查询 NVRAM 或数据库的函数, 则通过为对应的查询项增设预设值来仲裁对应的问题.

- 基于系统调用记录的分析策略

此分析策略是一种备选策略, 通过检查关键系统调用, 快速排除一些由套接字初始化失败引起的问题. 一些应用的开发者选择通过一个预定义的设置来初始化网络链接, 若模拟执行环境的网络配置与该预设不匹配, 会导致网络服务的初始化失败. 为提供网络服务, 应用需要创建一个套接字, 为套接字绑定一个地址和端口, 若必要的话为套接字设置属性, 最后监听用户请求. 这一系列操作中的每一步都与一个特定的系统调用相关. 此算法检查的相关系统调用包括 *sys_socket*(·), *sys_bind*(·), *sys_setsockopt*(·)和 *sys_listen*(·), 若有失败的套接字相关系统调用记录, FirmDep 将尝试根据系统调用的返回值, 对仿真执行环境的网络配置进行调整.

5 系统实现

本文为 32 位 ARM 架构实现了 FirmDep 的实验原型系统, 其源码可在文献[21]获取. 选择此架构的原因之一是该架构被广泛应用于 I 型设备中; 另一原因则是在开发原型系统时, PANDA 尚未为 x86 和 ARM 以外的硬件架构提供重放/回放功能的支持. 若 PANDA 或其他的提供重放/回放功能的仿真器支持了新的硬件架构, FirmDep 即可该硬件架构提供支持, 只要按照该架构的函数调用规范调整分析算法的实现即可.

5.1 仿真执行环境

仿真执行环境默认模拟一个带有 1G 内存、机器类型为 versatilepb 的设备. 选择此机器类型的原因是因为在进行 FirmDep 原型开发时, 只有该机器类型的 ARM 设备的执行轨迹录制/回放功能可以正常工作. 考虑到 versatilepb 默认仿真的 ARMv6 架构指令集的处理器可能无法运行一些用于较新 ARM 设备的可执行文件, FirmDep 为仿真设备指定处理器为 Cortex-A15 架构, 并加载一个经过修改的可支持 ARMv7 指令集的用于 versatilepb 设备的 Linux 3.14 版本内核. 在进行嵌入式应用托管时, 仿真执行环境加载一个基于 buildroot[22] 搭建的初始阶段系统并绑定一个默认的 IP 地址(192.168.1.1). 若仲裁措施包括网络配置的调整, 则按需进行调整. 此后, 初始阶段系统使用 chroot 工具将固件中包含的文件系统指定为根文件系统, 并运行固件文件系统中的 *shell*. 仿真执行环境通过设置环境变量, 使运行于其中的应用加载时会加载 API 劫持库. 最后, 使用固件预处理过程中识别出的启动指令来初始化嵌入式应用.

当仿真器开始启动, Tracer 首先通过 PANDA syscalls2 插件监听 *chroot* 函数调用, 等待环境初始化完毕. 此后, Tracer 通过 PANDA *osi_linux* 插件在每次 ASID 发生变化时(意味着发生了任务调度)获取当前线程的信息, 若任务名与目标应用相同, 则开始为程序录制执行回放. 若用户注册了强制执行的任务, Tracer 在这过程还将监视寄存器状态, 并通过 QEMU API 对系统状态进行修改.

获得执行回放记录后, Tracer 重放执行回放记录并生成执行轨迹. 具体而言, Tracer 插件.

1) 通过监听 *do_ mmap*2(·)系统调用获取最新的内存映射更新, 同时也记录一些网络、文件相关系统调用的行为;
2) 在每次 ASID 变化时获取任务名、pid 等信息以确认当前是否在运行目标应用;
3) 为 PANDA 的 PANDA_CB_BEFORE_BLOCK_EXEC 和 PANDA_CB_AFTER_BLOCK_EXEC 钩子注册回调函数, 使回调函数在基本块被执行前后分别被调用.

若当前正在运行目标应用且满足录制条件, 调用 QEMU 的 API 获取当前的寄存器状态信息、正在执行的基本块起始地址和长度和当前回放指令计数. 在应用托管过程中, 预加载动态链接库通过设置寄存器状态为特定状态来传递信息, 这些信息将在此步骤中被 Tracer 插件获取.

FirmDep 实验原型的仿真执行环境以 2020 年 11 月 30 日前的最新 PANDA 源码版本为基础搭建. 早期版本的 Tracer 插件基于 C 语言实现, 随着 PANDA 版本更新提供了 Python 绑定, Tracer 插件改为主要由 Python 实现, 包含约 500 行代码. 托管过程中加载的预加载动态链接库通过扩展源自 Firmadyne 项目用于劫持 NVRAM 相关硬件操作的动态链接库实现, 添加约 400 行 C 代码.

© 中国科学院软件研究所 http://www.jos.org.cn

5.2 执行轨迹分析器

FirmDep 实验原型的执行轨迹分析器以 2020 年 11 月 30 日前的最新 angr 版本为基础搭建. 此部分包含约 4 500 行 Python 代码, 其中, 执行轨迹预处理、模拟执行及各类分析算法的实现包含约 1 100 行代码, 执行轨迹查看器包含约 300 行代码, 常见 API 函数参数及返回值的类型识别、错误识别包含约 2 000 行代码, 针对已识别通用问题提出修复方案及强制执行修复相关的部分包含约 800 行代码.

6 实验及结果分析

6.1 嵌入式应用托管测试

为验证本文提出的固件托管方案的有效性, 本文建立了一个由真实设备固件组成的实验数据集, 并使用该实验集对 FirmDep 进行评估. FirmDep 实验平台在以下环境实现.

- 处理器: Intel(R) Core i7-8700 4.0 GHz;
- 内存: 32 GB DDR4;
- FirmDep 的实现及其依赖均被打包于以 Ubuntu 18.04LTS 为基础的 Docker 容器中.

- 研究问题.

本文尝试回答如下研究问题.

- RQ1. 本文提出的固件托管方案的兼容性如何, 与现有方案相比能否有效提高托管成功率?
- RQ2. 本文提出的固件托管方案能否有效识别出托管失败的嵌入式应用的环境依赖问题, 并给出仲裁方案?
- RQ3. 本文提出的固件托管方案是否适用于安全分析应用?

- 实验集及固件预处理.

本文实验部分的测试对象为一系列运行于 I 型嵌入式设备的 Web 应用. 选择此类服务的原因包括:

1) 此类服务主要用于为用户提供一个对设备进行控制的界面, 在修改配置的过程中较有可能需要与外设交互, 也因此在托管过程中较为容易遇到环境依赖问题;
2) 此类服务是嵌入式设备漏洞的重灾区. 根据我们的统计, 60%以上的嵌入式设备 CVE 与 Web 应用有关, 这使得它们成为了安全研究的重点研究对象.

此外, Web 应用托管成功率为先前的固件托管方案(包括 Costin 方案、Firmadyne 和 FirmAE)的唯一或主要的托管成功率统计指标, 选择 Web 应用作为托管目标可便于与先前的方案作对比.

本文使用由 Firmadyne 项目提供的网络爬虫从多个设备供应商的网站、FTP 服务器及备档获取了一系列用于路由器、网络摄像机、网关和其他嵌入式设备的固件镜像. 对于每个固件镜像, 本文尝试使用 binwalk[23] 提取出其中的文件系统, 并通过检查其中的可执行文件, 判断固件的处理器架构. 然后, 通过检查固件中是否包含使用*httpd, boa, goahead, nginx 等常见 Web 服务应用名称的文件, 筛选出其中包含 Web 服务的设备固件. 为了方便与先前的方案作对比, 本文参考 Firmadyne 和 FirmAE 这两个方案在实验阶段中的品牌选择, 选取了来自 ASUS, Netgear, D-Link, TP-Link, Linksys, Belkin, TrendNet, Zyxel 这 8 个嵌入式设备固件厂商的固件, 并使设备型号尽可能覆盖先前 Firmadyne 和 FirmAE 方案提供的数据集. 考虑到一个设备会有多个版本的固件, 我们为每个设备型号仅保留一个版本的固件.

在确定需要分析的固件后, 本文使用一个分析脚本提取出其中嵌入式 Web 服务的启动参数和依赖的配置文件等. 分析脚本首先在固件文件系统包含的所有有效路径及字符串中寻找包含常见 Web 服务应用子字符串的字符串, 包括*httpd, boa, goahead 和 nginx. 若在/etc/init.d 或/etc/rc.d 中找到包含这些字符串的脚本文件, 则将这些文件当作是 Web 服务的初始化脚本, 直接通过执行它们来初始化服务. 若字符串是来自某个二进制文件, 则将该其当作是 Web 服务的启动指令, 并对其进行进一步分析. 如果候选启动指令中包含任何类似路径的子串, 例如/etc/lighttpd.conf, 脚本将检查其是否存在, 并在必要的时候提醒用户去获取对应的文件. 对于一

些配置文件在文件系统中不存在的情况, 将尝试使用 FirmAE 托管该固件, 并在等待固件初始化基本完成后从其文件系统中提取文件. 最终, 本文筛选出 217 个符合条件的嵌入式固件, 并对其中的 Web 服务进行了测试, 数据集信息见文献[21].

- RQ1. FirmDep 的托管兼容性及与已有方案的对比.

本文使用 Firmadyne, FirmAE 及 FirmDep 分别对实验数据集的嵌入式 Web 应用进行了测试, 此类 Web 应用通常为对应设备的管理界面. 本文对成功托管的定义为固件中至少有一个 Web 应用能够监听并响应用户的请求. 考虑设备固件中可能存在文件缺失的情况, 被托管的应用可能并不能呈现正常的网页, 而是返回 HTTP 404 等错误信息. 本文认为, 在这样的情况下, 该 Web 应用依然是正常响应请求, 且具备被分析的价值,因此, 这样的情况本文也定义为成功托管. 各固件托管方案对实验集的托管测试结果见表 1.

表 1 FirmDep 及现有固件托管方案对实验集的托管测试结果

品牌	固件数	成功托管 Web 服务的固件数量			
		Firmadyne	FirmAE	FirmDep(仲裁前)	FirmDep(仲裁后)
ASUS	45	0 (0%)	24 (53.3%)	1 (2.2%)	29 (64.4%)
Belkin	6	0 (0%)	0 (0%)	1 (16.7%)	6 (100%)
D-Link	10	0 (0%)	6 (60%)	9 (90%)	9 (90%)
Linksys	13	0 (0%)	9 (69.2%)	0 (0%)	9 (69.2%)
Netgear	65	0 (0%)	32 (49.2%)	16 (24.6%)	43 (66.2%)
TP-Link	43	1 (2.3%)	26 (60.5%)	23 (53.5%)	23 (53.5%)
TrendNet	15	0 (0%)	15 (100%)	1 (6.7%)	13 (86.7%)
Zyxel	20	0 (0%)	1 (5%)	15 (75%)	15 (75%)
总和	217	1 (0.4%)	113 (52.1%)	66 (30.4%)	147 (67.7%)

在 217 个嵌入式 Web 应用中, FirmDep 在未经执行轨迹分析的情况下能够直接托管 30.4%的 Web 应用, 而经过执行轨迹分析及应用仲裁方案后能够托管 67.7%的应用, 成功率提高了一倍有余. 与现有的方案相比, Firmadyne 仅成功托管了 1 个 Web 应用, 而 FirmAE 的托管成功率整体而言也并不如 FirmDep 经过修复后的方案, 仅对于 TP-Link 的 Web 应用取得了比 FirmDep 更好的托管成功率. CostinFA 方案由于未放出源代码, 无法用于对比实验, 但是由于其采用了与 FirmDep 相同的基于 chroot 的托管方案, 因此可认为其固件托管成功率应接近或劣于未进行执行轨迹分析和仲裁的 FirmDep. 需要说明的是, FirmDep 并未包含 FirmAE 在 Firmadyne 基础上添加的仲裁措施, 这体现了 FirmDep 的固件托管方案配合半自动执行轨迹分析, 其效果可媲美或胜于当前完全基于人工经验分析的固件托管方案.

- RQ2. FirmDep 对嵌入式应用的环境依赖识别和仲裁能力.

通过使用本文提出的启发式算法, FirmDep 识别出了 95 个未能直接成功托管的 Web 应用的环境依赖问题, 并通过应用仲裁方案修复了 81 个问题. 表 2 展现了 FirmDep 在尝试托管实验数据集固件中的 Web 应用时识别出的部分环境依赖问题及对应的仲裁方案.

总体而言, FirmDep 识别出的环境依赖问题包含以下几类.

(1) 与文件相关的问题. 在 FirmDep 识别出的环境依赖问题中, 此类问题是最常见的. 具体表现为尝试使用 *open*(·), *fopen*(·)及 *open*64(·)等文件操作 API 以写入模式打开一个.pid 或者是.log 文件. 经过对固件文件系统进行分析, 我们发现这通常是因为这些文件所位于的路径是无效的符号链接, 这些符号链接往往是指向/tmp 或/tmp 路径下的目标, 这意味着这些符号链接只有在完成了系统初始化过程后才会变得可用. 对于此类问题, 本文为 FirmDep 制定的仲裁方案为在进行应用托管前为上述的符号链接创建有效的指向目标;

(2) 与套接字相关的问题. FirmDep 发现了一些失败的套接字初始化操作. 通过 errno 信息, FirmDep 识别出对应的嵌入式 Web 服务尝试为一个套接字绑定一个无效的 IP. FirmDep 通过对 *bind*(·)传递的 sockaddr 类型指针进行解析, FirmDep 识别出了这些函数调用所绑定的目标地址及端口. 对于此类的问题, 制定的仲裁方案为调整仿真环境的 IP 地址及端口映射设置, 以满足嵌入式 Web 服务的需求;

(3) 无效的函数返回值. 在尝试托管部分来自 ASUS 固件的嵌入式 Web 服务的过程中, 一些 Web 服务由

于无效的 *strlen*(·)及 *memset*(·)函数调用而触发了段异常错误. 造成段异常错误的原因是, 这些函数调用尝试操作空指针. 通过使用反向污点分析追踪空指针的来源, FirmDep 定位到一个用于从 NVRAM 硬件中读取数据的 *nvram_get*(·)函数调用. 然而, 由于这些 Web 服务未能加载 NVRAM 模拟库, 未能为这些服务提供有效的仲裁措施;

(4) 非 POSIX 函数导致的问题. 通过使用反向污点分析算法, FirmDep 识别出了一些导致程序进入异常执行路径的函数调用. 其中: *mssl_init*(·)函数的异常返回值影响了几乎所有的来自 ASUS 的 Web 应用的运行, *agApi_fwGetNextTriggerConf*(·)及 *isLanSubnet*(·)的返回值影响了一定数量的来自 Netgear 的 Web 应用的运行. FirmDep 通过使用约束求解为这些函数生成了可改变执行路径的返回值, 并通过使用强制执行对这些问题进行了修复.

表 2　FirmDep 识别出的部分环境依赖问题及对应的仲裁方案

厂商	应用路径	环境依赖问题	仲裁方案
ASUS	*/usr/sbin/httpds*	*mssl_init*(...)=0	强制执行,修改返回值为 1
		open("*/var/run/httpd.pid*","*w*")=–1	修复路径*/var/run*
		fopen("*/var/run/httpd.pid*","*w*")=0	修复路径*/var/run*
		nvram_get("*x_Setting*")=0	无(未能修复)
Belkin	*/usr/sbin/minhttpd*	*bind*(...,〈"*AF_INET*","192.168.2.1","8080"〉,...)=–1	调整 IP 地址及端口映射
	/bin/httpd	*bind*(...,〈"*AF_INET*","192.168.0.1","80"〉,...)=–1	调整 IP 地址
		fopen("*/var/run/httpd.pid*","*w*")=0	修复路径*/var/run*
Linksys	*/usr/local/bin/thttpd*	*fopen*("*/var/run/thttpd.pid*","*w*")=0	修复路径*/var/run*
	/usr/sbin/lighttpd	*fopen*("*/var/run/lighttpd.pid*","*w*")=0	修复路径*/var/run*
Netgear	*/usr/sbin/lighttpd*	*fopen*("*/var/run/httpd.pid*","*w*")=0	修复路径*/var/run*
		*open*64("*/var/run/httpd.pid*","*w*")=0	修复路径*/var/run*
		*open*64("*/var/log/lighttpd/error.log*","*w*")=0	修复路径*/var/log/lighttpd*
		open("*/etc/lighttpd/certs/server.pem*")=–1	执行命令生成证书文件
	/usr/sbin/httpd	*fopen*("*/var/run/httpd.pid*","*w*")=0	修复路径*/var/run*
		agApi_fwGetNextTriggerConf(...)=0	强制执行,修改返回值为 1
		bind(...,〈"*AF_INET*","192.168.0.1","80"〉,...)=–1	调整 IP 地址
		isLanSubnet(...)=0	强制执行,修改返回值为 1

- RQ3. 对嵌入式 Web 应用的漏洞扫描测试.

为验证 FirmDep 是否适用于安全分析场景, 本文对成功托管的嵌入式 Web 应用进行了漏洞扫描测试, 扫描结果见表 3.

表 3　对嵌入式 Web 应用的漏洞扫描结果

w3af 扫描结果			
危险等级	说明	影响的 Web 应用数目	
High	HTTP Basic authentication	4	共 4 个应用受影响
	Guessable credentials	4	
Medium	Click-Jacking	66	共 66 个应用受影响
	Secure content over insecure channel	12	
	Unhandled error in Web application	4	
Low	Cookie without HttpOnly	4	共 25 个应用受影响
	Private IP disclosure	5	
	Virtual host identified	11	
	Path disclosure	6	
Info	Potential buffer overflow	3	共 5 个应用受影响
	Potential virtual host misconfiguration	2	
总数		69	

RouterSploit 扫描结果	
漏洞	影响的 Web 应用数目
eseries_themoon_rce	9
uc_httpd_path_traversal	2
总数	9

© 中国科学院软件研究所 http://www.jos.org.cn

首先, 本文使用了加载 full_audit 配置文件的 w3af[24]对被托管的 Web 服务进行扫描, 以检查其是否包含网页漏洞. 根据 w3af 的扫描结果, 被托管的 147 个嵌入式 Web 应用中有 69 个应用被检测为可能受到漏洞影响, 其中, 分别有 4 个应用受到高等级漏洞影响、66 个应用受中等级漏洞影响及 25 个应用受低等级漏洞影响. 此外, 在上述的检测到可能存在漏洞的应用中,有 2 个应用可能包含缓冲区溢出漏洞, 3 个应用可能未适当配置虚拟主机相关的设置. 此外, 本文还使用 RouterSploit[25]对被托管的 Web 应用进行测试, 以检测这些服务是否受到已知漏洞的影响. 最终, 发现共 9 个 Web 应用受到已知漏洞的影响. 此实验结果表明了 FirmDep 应用于安全分析应用的可行性.

6.2 未托管案例分析

在第 6.1 节的固件托管过程中, 实验集中的一些嵌入式 Web 应用未能被直接托管, 且 FirmDep 未能为它们提供可用的仲裁方案. 未托管的案例可分为以下情况.

(1) 未生成执行轨迹. 在实验集的 217 个嵌入式 Web 应用中, FirmDep 未能为 28 个应用它们生成执行轨迹, 原因是在整个托管过程中, FirmDep 都未能识别到这些程序的执行. 应用未执行的情况主要归类为下面两者之一: 1) exec error 或 illegal instruction, 造成之类问题的原因主要是仿真器对嵌入式应用二进制文件中的指令未提供完整支持或由于未正确解压导致二进制文件完整性受到破坏; 2) 缺少动态链接库, 造成此类问题的原因主要是未正确解压或固件未包含完整文件系统导致文件系统不完整;

(2) 未能完成执行轨迹的预处理. 一些未成功托管的 Web 应用, 尽管它们的执行轨迹被生成了, 但 FirmDep 未能完成执行轨迹的预处理. 造成这种情况的原因包括: 1) Web 应用的可执行文件未能被 angr 加载; 2) 在判定基本块的有效性时出现了假阴性, 导致预处理后的执行轨迹残缺;

(3) 未从执行轨迹中识别出环境依赖问题. 首先, 本文所提出的几种基于错误处理分析的启发式算法并不能覆盖到所有真实世界的错误处理情况. 例如: 一些函数在调用时会传递一个用于存放结果的指针, 而不通过函数的返回值来传递结果. 此外, 一些应用可能对错误有较高的容忍度, 使得它们在遭遇错误后仍然能回到正常的执行路径. 例如: 通过对执行轨迹的人工分析, 我们发现一个来自未成功托管的 D-Link 的 Web 应用实际上已调用了 *select*(·)函数并在后台运行. 在此之前, 应用尝试将对应的套接字绑定到名为 *br*0 的网络接口上, 但由于该网络接口不存在于仿真环境中, 因此这些操作都失败了. 通过调整仿真环境的网络配置, 该 Web 服务可被成功托管. 由于该环境依赖问题并非通过本文提出的启发式算法分析得到, 因此此案例不计入成功托管的案例中. 再例如: 本文在分析一个来自 TP-Link 的 Web 应用时, 发现了一个未影响到任何执行分支的负值函数调用返回值, 不排除是因为开发人员遗漏了对该函数调用的错误处理. 除由于执行轨迹分析算法不足导致的分析失败以外, 若干个 Web 应用由于其二进制文件缺乏符号表, 难以从其执行轨迹推断出程序的高级语义行为, 故无法提出仲裁方案;

(4) 未能为已发现的环境依赖问题提供有效的仲裁方案. 除第 6.1 节中说明的部分未能修复的 *nvram_get*(·)函数调用外, 在实验中, 部分 Web 应用会尝试访问不存在的设备节点或文件. 例如: 通过反向污点分析, FirmDep 指出, 一个来自 Netgear 的 Web 应用可能是因为通过 *popen*(·)打开/tmp/jffs2ready 这一管道文件失败而导致托管失败. 由于缺乏对该设备的了解, 未能为该问题制定有效的仲裁方案.

7 讨　论

- 对其他处理器架构的支持

FirmDep 的实现并未包含太多架构相关的内容, 可很容易添加对其他架构处理器的支持. 事实上, 在进行 FirmDep 原型系统开发的时候, PANDA 已可提供 MIPS 架构及 ARM64 架构客户机的较好动态插桩及信息采集支持, 仅缺失对这些架构的录制/回放功能支持. FirmDep 使用录制/回放机制的动机并非主要是为分析效率,

而是为了托管过程与信息提取过程应用执行的一致性. 若可保证托管过程和分析过程的一致性, 亦可使用其他可获取运行轨迹的仿真器, 如 Valgrind 或用户态 QEMU 等替代 PANDA, 以实现其他处理器架构的支持. 此外, 尽管 FirmDep 原型系统仅为 32 位 ARM 函数调用规范(AAPCS)实现了反向污点分析算法, 由于算法的大部分实现是基于 VEX IR 的分析, 对于其他的处理器架构, 仅需对算法进行少量调整即可.

- FirmDep 方案的可能改进和局限性

经过第 6.1 节的固件托管实验及第 6.2 节对未成功托管的固件的分析, 我们将本文提出的固件托管方案的不足分为两类: (1) 通过扩展本文实现的原型可解决的问题; (2) 本文提出的方案难以解决的问题. 首先,通过扩展 binwalk 和 PANDA/Qemu, 可支持对更多设备固件的支持、提高运行二进制应用的兼容性; 通过为 FirmDep 添加更多的执行轨迹分析算法和仲裁措施, 可以识别修复更多本来无法托管的应用; 本文所实现的 FirmDep 原型系统并未充分利用 PANDA 所提供的对客户机系统的仲裁机能. 例如: PANDA 可通过劫持系统调用来伪造文件系统中的文件, 可利用这一技能来模拟一些硬件设备节点. 再比如, FirmDep 原型只使用了修改函数返回值这一强制执行手段, 通过修改函数调用参数或者跳过某些函数调用, 可支持修复更多的问题.

至于问题(2), 在实验过程中, 我们意识到一些嵌入式应用的环境依赖问题是难以使用本文提出的固件托管方案来识别和缓解的. 首先, 我们观察到, 一些应用的行为与配置文件内容高度相关(例如部分基于 lighttpd 二次开发的嵌入式 Web 应用). 本文提出的执行轨迹分析方法是基于基本块粒度的, 因此为此类应用记录的执行轨迹往往非常庞大, 其中很大一部分是对配置文件的文本解析. 这带来了两个挑战: (1) 本文提出的方案缺乏对字符串处理这一行为的感知, 难以将应用的行为与配置文件中的特定内容建立关联; (2) 此类应用的单次运行可产生包含几十万乃至几百万个基本块的执行轨迹. 本文提出的分析方案的模拟执行部分由于需要为每个基本块保留内存模型, 模拟执行的速度会逐渐减慢且占用大量内存, 在这种情况下, 单次的模拟执行将耗费大量时间, 使得本文提出的方案失去实用价值. FirmDep 在应对一些依赖外部字符串(例如启动参数)的嵌入式应用也面临相同的问题. 在分析本文第 6 节实验中属于这种情况的嵌入式应用时, 我们均尽可能裁剪了执行轨迹中包含配置文件解析的部分. 而对于缺乏符号信息、执行轨迹非常庞大无法裁剪的此类嵌入式应用, 我们均标记为托管失败. 除此之外, 本文提出的方案难以应对嵌入式应用的环境依赖信息不包含在设备固件中的情况. 此类情况包括应用依赖特殊的设备节点或文件中的内容、依赖特定较复杂的文本 NVRAM 值等. 此类情况对于其他的基于全系统仿真的固件托管方案而言是共同的难题, 通常需要获取原始的硬件设备并进行逆向提取解决.

- FirmDep 的定位

本文所提出的 FirmDep 固件托管方案与现有的全自动化固件托管方案是互补的关系, 而非相互替代的关系. 当不存在未解决的环境依赖问题时, FirmAE 等全自动化固件托管方案在使用上比 FirmDep 更简易. 另外, 在本文的实验过程中, 由于 FirmAE 框架包含了一些已知环境依赖问题的针对性修复, FirmAE 可支持托管一些未被 FirmDep 识别出环境依赖问题的服务. 用户可通过为 FirmDep 集成 FirmAE 等基于现有经验的固件托管框架所包含的仲裁措施来提高初始固件托管成功率, 亦可通过为 FirmAE 等框架添加部分由 FirmDep 生成的仲裁措施来实现更高的固件托管成功率. 此外, 本文认为, FirmDep 的用途并不局限于解决嵌入式应用托管环境依赖问题, 通过新增执行轨迹分析算法, FirmDep 也可用于其他的应用场景, 例如恶意软件行为分析等.

8 结 语

本文针对现有嵌入式设备固件托管方案无法识别和解决被托管应用未知环境依赖问题的缺陷, 提出了一种 I 型嵌入式设备应用托管的固件托管方案 FirmDep. FirmDep 通过使用按需记录及 PANDA 的录制/重放功能, 实现对应用托管过程的高效记录. 当嵌入式应用无法被成功托管时, FirmDep 可对执行轨迹进行信息提取及系统状态补全, 并使用多种执行轨迹分析算法识别和仲裁目标应用的环境依赖问题. 实验结果表明: 本文提出的固件托管方案可实现高于现有全自动固件托管方案的托管成功率, 能有效识别和解决嵌入式设备应用在运行过程中的环境依赖问题, 且可用于嵌入式设备固件的漏洞挖掘.

© 中国科学院软件研究所 http://www.jos.org.cn

References:

[1] Yu R, Nin F, Zhang Y, *et al*. Building embedded systems like it's 1996. In: Proc. of the 2022 Network and Distributed System Security Symp. San Diego: Internet Society, 2022. 1–18. [doi: 10.14722/ndss. 2022.24031]

[2] Yu YC, Chen ZN, Gan ST, *et al*. Research on the technologies of security analysis technologies on the embedded device firmware. Chinese Journal of Computers, 2021, 44(5): 859–881 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2021.00859]

[3] Chen DD, Egele M, Woo M, *et al*. Towards automated dynamic analysis for linux-based embedded firmware. In: Proc. of the 2016 Network and Distributed System Security Symp. San Diego: Internet Society, 2016. 1–16. [doi: 10.14722/ndss.2016.23415]

[4] Kim M, Kim D, Kim E, *et al*. FirmAE: Towards large-scale emulation of iot firmware for dynamic analysis. In: Proc. of the 36th Annual Computer Security Applications Conf. Austin: ACM, 2020. 733–745. [doi: 10.1145/3427228. 3427294]

[5] Zaddach J, Bruno L, Francillon A, *et al*. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: Proc. of the 2014 Network and Distributed System Security Symp. San Diego: Internet Society, 2014. 1–16. [doi: 10.14722/ndss.2014.23229]

[6] Gui ZJ, Shu H, Kang F, *et al*. FIRMCORN: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution. IEEE Access, 2020, 8: 29826–29841. [doi: 10.1109/ACCESS.2020.2973043]

[7] Muench M, Stijohann J, Kargl F, *et al*. What you corrupt is not what you crash: challenges in fuzzing embedded devices. In: Proc. of the 2018 Network and Distributed System Security Symp. San Diego: Internet Society, 2018. 1–15. [doi: 10.14722/ndss.2018. 23166]

[8] Costin A, Zarras A, Francillon A. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In: Proc. of the 11th ACM on Asia Conf. on Computer and Comm. Security. Xi'an: ACM, 2016. 437–448. [doi: 10.1145/2897845. 2897900]

[9] Zheng Y, Davanian A, Yin H, *et al*. FIRM-AFL: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In: Proc. of the 28th USENIX Conf. on Security Symp. USENIX Association, 2019. 1099–1114.

[10] Srivastava P, Peng H, Li J, *et al*. FirmFuzz: Automated iot firmware introspection and analysis. In: Proc. of the 2nd Int'l ACM Workshop on Security and Privacy for the Internet-of-things. London: ACM, 2019. 15–21. [doi: 10.1145/3338507.3358616]

[11] Clements AA, Gustafson E, Scharnowski T, *et al*. HALucinator: Firmware re-hosting through abstraction layer emulation. In: Proc. of the 29th USENIX Conf. on Security Symp. USENIX Association, 2020. 1201–1218.

[12] Feng B, Mera A, Lu L. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: Proc. of the 29th USENIX Conf. on Security Symp. USENIX Association, 2020. 1237–1254.

[13] Liu Q, Zhang C, Ma L, *et al*. FirmGuide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution. In: Proc. of the 36th IEEE/ACM Intl. Conf. on Automated Software Engineering. Melbourne: IEEE, 2021. 792–804. [doi: 10.1109/ASE51524.2021.9678653]

[14] Jiang M, Ma L, Zhou Y, *et al*. ECMO: Peripheral transplantation to rehost embedded linux kernels. In: Proc. of the 2021 ACM SIGSAC Conf. on Computer and Comm. Security. ACM, 2021. 734–748. [doi: 10.1145/3460120.3484753]

[15] Zhou W, Guan L, Liu P, *et al*. Automatic firmware emulation through invalidity-guided knowledge inference. In: Proc. of the 30th USENIX Conf. on Security Symp. USENIX Association, 2021. 2007–2024.

[16] Dolan-Gavitt B, Hodosh J, Hulin P, *et al*. Repeatable reverse engineering with panda. In: Proc. of the 5th Program Protection and Reverse Engineering Workshop. New York: Association for Comp. Machinery. 2015. 1–11. [doi: 10.1145/2843859. 2843867]

[17] Bellard F. QEMU, a fast and portable dynamic translator. In: Proc. of the 2005 USENIX Annual Technical Conf. USENIX Association, 2005. 41–46.

[18] O'Callahan R, Jones C, Froyd N, *et al*. Engineering record and replay for deployability. In: Proc. of the 2017 USENIX Annual Technical Conf. USENIX Association, 2017. 377–389.

[19] Shoshitaishvili Y, Wang R, Salls C, *et al*. SOK: (state of) the art of war: Offensive techniques in binary analysis. In: Proc. of the 2016 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2016. 138–157. [doi: 10.1109/SP.2016.17]

[20] Intermediate representation—Angr documentation. 2023. https://docs.angr.io/advanced-topics/ir

[21] FirmDep artifacts. 2023. https://gitlab.com/firmdep/firmdep_artifacts

[22] Buildroot—Making embedded linux easy. 2023. https://buildroot.org/

[23] Binwalk. GitHub. 2023. https://github.com/ReFirmLabs/binwalk/wiki/Usage

[24] W3af—Open source Web application security scanner. 2023. http://w3af.org/

[25] RouterSploit—Exploitation framework for embedded devices. 2023. https://github.com/threat9/routersploit

© 中国科学院软件研究所 http://www.jos.org.cn

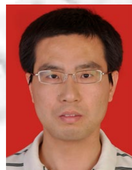
附中文参考文献:

[2] 于颖超, 陈左宁, 甘水滔, 等. 嵌入式设备固件安全分析技术研究. 计算机学报, 2021, 44(5): 859–881. [doi: 10.11897/SP.J.1016.2021.00859]

吴华茂(1995－), 男, 博士, CCF 学生会员, 主要研究领域为嵌入式设备软件, 系统安全.

周亚金(1982－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为区块链安全系统, 软件安全.

姜木慧(1994－), 男, 博士, 主要研究领域为网络安全, 系统安全.



李金库(1976－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为系统与网络安全, 移动安全, 云计算及其安全, 大数据应用及其安全.

www.jos.org.cn

www.jos.org.cn

© 中国科学院软件研究所 http://www.jos.org.cn