计算机科学
COMPUTER SCIENCE

一种面向嵌入式设备的动态插桩方法

司健鹏, 洪征, 周振吉, 陈乾, 李涛

引用本文

司健鹏, 洪征, 周振吉, 陈乾, 李涛. 一种面向嵌入式设备的动态插桩方法[J]. 计算机科学, 2024, 51(11): 347-355.

SI Jianpeng, HONG Zheng, ZHOU Zhenji, CHEN Qian, LI Tao. Dynamic Instrumentation Method for Embedded Physical Devices [J]. Computer Science, 2024, 51(11): 347-355.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

关键字敏感的嵌入式设备固件模糊测试方法

Keyword Sensitive Fuzzing Method for Embedded Device Firmware

计算机科学, 2024, 51(10): 196-207. https://doi.org/10.11896/jsjkx.230700068

基于函数调用指令特征分析的固件指令集架构识别方法

Function-call Instruction Characteristic Analysis Based Instruction Set Architecture Recognization Method for Firmwares

计算机科学, 2024, 51(6): 423-433. https://doi.org/10.11896/jsjkx.230500087

基于信息熵与闭合频繁序列的密码协议逆向方法

Cryptographic Protocol Reverse Method Based on Information Entropy and Closed Frequent Sequences

计算机科学, 2024, 51(3): 326-334. https://doi.org/10.11896/jsjkx.221200147

基于残差网络和循环神经网络混合模型的应用层协议识别方法

Application Layer Protocol Recognition Based on Residual Network and Recurrent Neural Network

计算机科学, 2022, 49(11): 293-301. https://doi.org/10.11896/jsjkx.210800252

基于多尺度的稀疏脑功能超网络构建及多特征融合分类研究

Construction and Multi-feature Fusion Classification Research Based on Multi-scale Sparse Brain Functional Hyper-network

计算机科学, 2022, 49(8): 257-266. https://doi.org/10.11896/jsjkx.210600094

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230700091

一种面向嵌入式设备的动态插桩方法

司健鹏　洪　征　周振吉　陈　乾　李　涛
陆军工程大学指挥控制工程学院　南京 210007
(1183373785@qq.com)

摘　要　现有动态插桩方法大多基于 x86/x64 指令集,对嵌入式设备常用的 RISC 兼容性较差,且在应用嵌入式设备时存在插桩效率低、资源消耗大等问题。文中提出了一种面向嵌入式设备的动态插桩方法 DIEB(Dynamic Instrumentation Method for Embedded Physical Devices)。DIEB 在嵌入式设备中使用以控制转移指令为探针的探测模式对目标进程进行动态二进制插桩。DIEB 提出了一种轻量化的解释执行指令方法,根据指令的运行环境设置指令解释执行区域,并在解释执行区域中解释执行指令获取执行结果。在目标进程动态运行过程中,DIEB 通过解释执行用作探针的控制转移指令,获取控制转移指令的目的地址,从而跟踪目标进程的执行流,在软硬件资源紧张的嵌入式设备上高效地进行动态插桩。ARM 指令集是一种典型的 RISC 指令集,测试实验以 ARM 指令集为验证对象,在 NetGear R7000 等设备上进行。实验结果表明,经过 DIEB 插桩的进程可以正常运行,插桩导致的时延远小于基于 ptrace 的插桩方式,解决了 PIN,Dynamorio 等现有动态插桩框架难以在嵌入式设备上运行的问题。此外,DIEB 具有在多线程环境下稳定运行的能力,可以准确记录并发线程的执行流轨迹。

关键词:动态二进制插桩;指令解释执行;嵌入式设备;灰盒测试;程序运行状态反馈
中图分类号　TP313

Dynamic Instrumentation Method for Embedded Physical Devices

SI Jianpeng,HONG Zheng,ZHOU Zhenji,CHEN Qian and LI Tao
College of Command and Control Engineering,Army Engineering University of PLA,Nanjing 210007,China

Abstract　Most existing dynamic instrumentation methods are based on the x86/x64 instruction set,which is poorly compatible with reduced instruction set(RISC) commonly used in embedded devices,and there are problems such as low instrumentation efficiency and large resource consumption when the dynamic instrumentation methods are applied to embedded devices. This paper proposes a dynamic instrumentation method for embedded physical devices(DIEB). DIEB uses control transfer instructions as probes in embedded devices to dynamically perform binary instrumentation on target processes. It proposes a lightweight method to interpret the execution of instructions,and sets the instruction execution area based on the operating environment. DIEB interprets the execution instructions in the simulation execution area to obtain the execution results. During the dynamic operation of the target process,DIEB interprets and executes control transfer instructions to obtain the destination address of the control transfer instructions,and tracks the execution flow of the target process so as to efficiently perform dynamic instrumentation on embedded devices with limited resources. Taking the ARM instruction set as the verification object,experiments are carried out on physical devices such as NetGear R7000. Experimental results show that the DIEB instrumentation process can run normally,and the time delay caused by instrumentation is much smaller than that of the ptrace-based instrumentation method. In addition,DIEB can run stably in a multi-threaded environment and accurately record the execution flow traces of concurrent threads.

Keywords　Dynamic binary instrumentation,Instruction interpretation execution,Embedded equipment,Grey box test,Program operation status feedback

1　引言

随着嵌入式网络设备性能的提升和制造成本的下降,嵌入式设备在工业控制、消费电子、通信设备等领域的应用比重将越来越大[1]。

嵌入式设备在蓬勃发展的同时也带来了大量的安全漏洞。由于相应的安全防护技术还不完善,更新修补机制较为繁琐,因此嵌入式设备面临日趋严峻的安全威胁。2020 年

到稿日期:2023-07-13　返修日期:2023-11-13
基金项目:智慧城市网络安全综合防控关键技术及系统(2019YFB2101704)
This work was supported by the Key Technologies and Systems for Comprehensive Prevention and Control of Cybersecurity in Smart Cities (2019YFB2101704).
通信作者:洪征(hz5215@163.com)

6 月,安全专家披露了名为“CallStranger”(漏洞编号 CVE-2020-12695)的 upnp 漏洞[2],该漏洞可影响数十亿设备。2022 年国家计算机病毒应急处理中心和 360 公司分别发布专题研究报告[3],指出美国国家安全局(NSA)的网络攻击武器“酸狐狸”漏洞攻击武器平台可利用嵌入式网络设备在局域网中的特殊地位,对目标主机发起中间人攻击。因此,研究嵌入式设备漏洞挖掘技术对解决嵌入式设备的安全问题有着重要的现实意义。

现有的漏洞挖掘方法往往依赖动态插桩技术提供的程序运行状态反馈。动态插桩技术是在不改写程序源文件的前提下,在进程运行过程中插入桩代码。

动态二进制插桩的粒度根据具体应用的需求和性能要求进行调整,主要分为指令粒度、函数粒度和基本块粒度。指令粒度的插桩指在每条指令前后进行插桩,可以满足需要细粒度的程序运作状态反馈的动态分析技术,但这种插桩粒度会带来较大的性能开销;基本块粒度的插桩指在每个基本块的入口和出口处进行插桩,一般用于统计目标进程运行时的代码覆盖率,可以满足以代码覆盖率为引导的动态分析技术的需求,如 AFL[4];函数粒度的插桩指在函数入口和出口处进行插桩,可以用于统计进程运行过程中函数调用序列。由于基本块粒度的插桩相比指令粒度的插桩具有更高的效率,因此本文采用基本块粒度的插桩方法对嵌入式设备进行插桩。

在通用 PC 平台上,主流的动态二进制插桩有 3 种执行模式:JIT 模式(Just-in-time Mode)、解释模式(Interpretation Mode)和探测模式(Probe Mode)。

JIT 模式通过运行时动态编译代码,可进行指令粒度的插桩,并且可以根据程序的实际执行情况,动态调整优化策略,能够获得较好的性能表现,因此 JIT 模式是较常用的动态二进制插桩模式。在 JIT 模式下,原始二进制文件或可执行文件实际上没有被修改或执行。PIN[5] 从第一条被劫持的指令开始,使用 JIT 编译器将即将执行的指令编译生成代码副本,在此过程中将用户自定义的功能代码植入代码副本中,替换原代码执行。Dynamorio[6] 在目标进程和操作系统间构建了一个中间解释层,通过将二进制程序的代码拷贝到代码缓存的方式模拟目标程序的执行。在动态模拟执行目标程序的过程中,可以根据分析需求,对程序运行状态进行监控和修改。但 PIN 和 Dynamorio 对嵌入式设备指令集支持不足,难以移植到嵌入式设备中运行。

在解释模式下,每条指令被更换成一条或多条具有相应功能的替代性指令。例如,Valgrind 通过构建中间语言 IR,将指令转化为中间语言进行插桩,可以兼容多种指令集,但转化翻译 IR 语言的过程会消耗较多的软硬件资源,对于资源有限的嵌入式设备,不能很好地移植。

在探测模式下,一般使用新指令覆盖原有指令,来达到修改二进制程序的目的。gdb 将待插桩的指令修改为“int 3”指令,在待插桩指令被执行时,触发软中断,此时,gdb 通过 ptrace 系统调用获取目标进程的调试权限。但这一过程会频繁要求进程间的通信和用户态与内核态的状态切换,运行开销较大,也不适合直接应用于嵌入式设备的自动化动态分析。

与动态二进制插桩技术不同的是,静态二进制插桩在二进制程序运行之前改写程序源文件。静态二进制插桩方法首先基于静态反汇编工具对目标文件进行扫描,根据基本块、程序流程图 CFG、程序依赖图 PDG 等信息,确定待插桩的指令地址,而后将插桩代码写入这些地址。静态二进制插桩方法的局限性在于过于依赖静态反汇编工具的分析能力,对于二进制程序中的隐式控制流和隐式数据流难以准确地分析。因此,静态二进制插桩方法普遍存在反汇编精度不高等问题。

动态二进制插桩的 3 种执行模式中,JIT 模式要求动态编译程序,解释模式需要转换二进制指令,二者都需要大量的计算资源,不适用于资源紧张的嵌入式设备。本文采用探测模式,以控制转移指令为探针,对设备的目标进程进行动态二进制插桩。针对传统探测模式存在的频繁在用户模式和内核模式之间切换而导致的运行开销高昂的问题,本文提出了一种动态插桩方法,在程序动态运行过程中通过解释执行控制转移指令获取其目的地址,实现对目标进程指令流的监控。这一过程无需切换至内核模式,可以在用户模式中通过解析控制转移指令的语义来完成,从而大幅提升了插桩效率。本文的主要贡献如下:

1)提出了一种指令解释执行方法,根据指令的运行上下文环境设置指令解释执行区域,并在解释执行区域中解释执行指令,获取执行结果。指令解释执行方法是根据控制转移指令的特征设计的,可以方便快捷地获取控制转移指令的目的转移地址,避免 JIT 模式和解释模式需要动态编译程序,或转换二进制指令而产生的巨大资源消耗,从而大幅提高插桩效率。

2)在跟踪目标进程指令流的过程中通过解释执行控制转移指令,获取其目的地址,并对该地址进行插桩。由于当前指令在被 CPU 执行时,语义具有唯一性,因此可以保证解释结果的准确性,从而实现以跟踪进程执行流的方式,对执行路径上的每个基本块进行插桩,解决了静态二进制插桩技术难以识别隐式控制流和数据流的问题。

3)提出了一种解决多线程环境下由插桩导致代码段被竞争访问的线程调度方法,通过在多线程环境下进行安全的指令劫持和指令恢复操作,确保插桩方法可在同一时空内并发跟踪多个由目标进程创建的线程,满足高并发性的动态分析需求。

2 相关工作

嵌入式设备使用的指令集大多为 RISC 指令集,且软硬件资源有限,主流动态二进制插桩技术难以兼容嵌入式设备。在缺乏插桩技术支持的条件下,针对嵌入式设备的动态分析方法难以基于程序运行状态的反馈,调整测试用例的生成策略,生成高质量的测试用例。

一些研究人员在动态测试之前,通过静态分析固件的逻辑特征,提高测试用例的生成质量。FirmFuzz[7] 针对嵌入式设备的 Web 接口,提出了一种基于遗传算法的迭代模糊

器,用于生成符合语法规范的输入。相较于FirmFuzz,SIoT-Fuzzer[8]在对Web接口进行静态分析的基础上增加了状态分析,以分析消息之间的依赖关系,确保测试用例更有效地绕过服务器状态检查。SRFuzzer[9]同样通过分析Web前端来生成测试用例,但SRFuzzer根据Web前后端的语义处理逻辑,提出了“Key-Value”和“Conf-Read”模型。“Key-Value”模型指在Web请求中存在很多键值对,例如在登录请求中,会有类似于“username=admin”的参数,这里“username”为键名,“admin”为键值。“Conf-Read”模型指在Web请求中存在类似于“content_length=100 content=submit”的键值对,其中“content_length”为配置项,规定了“content”的最大长度为100。基于上述模型设计Web爬虫策略,生成种子并进行变异。FIRM-COV[10]和Snipuzz[11]认为嵌入式设备固件是具有严格输入要求的软件程序。FIRM-COV通过对二进制程序进行静态分析,利用符号执行技术,对程序的每个路径上的符号变量进行约束求解,生成满足路径约束条件的输入数据,提高测试的代码覆盖率。Snipuzz根据固件接收消息后发送的执行响应优化变异方法,使得生成的测试用例更好地满足固件语法要求。Nilo等[12]和Chen等[13]提出通过分析与目标物联网设备通信的移动APP的输出逻辑,设计高质量的测试用例生成策略。

上述工作在动态测试之前生成高质量的测试用例,但在动态测试阶段,由于缺乏程序运行状态反馈,调整测试用例生成策略的能力较弱。

为了在嵌入式设备上获得程序运行状态反馈,Jang等[14]提出了一种函数粒度的插桩方法,通过修改目标程序依赖的动态库,劫持库函数,将程序监控代码注入进程,从而在进程崩溃时记录完整的崩溃信息,指导后续测试用例生成。Zheng等[15]提出了一种间接的插桩方法,通过分析Web后端二进制程序的关键字,与测试用例的响应报文进行比对,推测测试用例的代码执行路径。FIRM-AFL[16]使用虚拟仿真技术实现了指令粒度的插桩。为了获得更高的仿真精度,FIRM-AFL设计了Qemu的系统模式和用户模式动态转换的调度策略,在系统模式下启动目标文件,随后切换至用户模式获得更快的执行速度,并使用插桩技术来获取程序运行的反馈信息。这种插桩方法依赖于仿真环境兼容设备固件的能力,在硬件接口需求特殊的设备固件上的应用效果较差。

总体上看,现有研究主要采用间接的方法提供程序运行状态反馈,难以在实体设备上做到基本块粒度的运行状态反馈。针对嵌入式设备的动态分析方法主要依赖静态分析过程中所生成的测试用例的质量,难以根据程序运行状态反馈实时调整策略,测试效率较低。

3 系统实现

3.1 系统概述

本文参考CPU流水线模型和代码基本块模型,在程序动态运行过程中实时解释执行控制转移指令的目的分支,并对计算得到的目的分支进行插桩,从而跟踪进程执行流,监控程序的运行状态。

1)CPU流水线模型

在单线程的执行环境下,CPU的执行顺序在逻辑上是线性的,且执行的指令代码的语义是唯一的。因此,采用动态分析程序的方法可以避免静态反汇编存在的指令语义二义性的问题。在程序动态运行的过程中,即使控制转移指令发生了地址空间的跳转,但CPU依然保持线性执行。

2)代码基本块模型

代码基本块是不包含分支或跳转语句的简单代码片段。代码基本块只有一个入口和一个出口。每个基本块在程序运行时都作为一个整体被连续执行,不会被其他语句打断。代码基本块通常是对代码进行分析和优化的最小单位。只要代码基本块中第一条指令被执行了,那么其他的指令都会按照顺序执行一次。

本文设计并实现了一种动态插桩方法DIEB。在RISC指令集上,使用探测模式对目标进程进行动态二进制插桩。ARM指令集是一种典型的RISC指令集,也是嵌入式设备常用的指令集之一。本文选择ARM指令集作为分析对象。

在插桩过程中,DIEB以控制转移指令为探针,跟踪目标进程执行流,获取目标进程基本块粒度的运行状态。桩代码是DIEB通过进程注入的方式写入目标进程内存中的一段代码。桩代码必须在目标进程内部运行,通过劫持和恢复目标进程的探针指令来控制目标进程的执行流,完成跟踪目标进程执行流的所有工作,从而获取目标进程运行状态。而探针指令为控制转移指令,桩代码需要通过解释执行探针指令获取其跳转地址,实现对目标进程执行流的跟踪。被注入桩代码的目标进程会在执行每个控制转移指令时进入桩代码,在桩代码内部完成记录进程运行状态、劫持目标进程下一个探针指令后,再次跳转回原执行流,继续通过下个探针指令进入桩代码。目标进程在桩代码的控制下不断重复上述过程,使桩代码获取到目标进程基本块粒度的运行状态。

DIEB由六大子模块组成,分别是进程注入模块、上下文保存恢复模块、核心变量初始化模块、指令解释执行模块、指令识别扫描模块和分支指令劫持模块。其中,桩代码由上下文保存恢复模块、核心变量初始化模块、指令解释执行模块、指令识别扫描模块和分支指令劫持模块组成,通过进程注入模块注入并运行于目标进程内存中。上下文保存恢复模块的功能是在进程进入或退出桩代码时保存或恢复进程运行上下文环境;核心变量初始化模块的功能是在进入桩代码后初始化桩代码运行所依赖的全局变量;指令解释执行模块的功能是获取当前被劫持的控制转移指令的目的跳转地址;指令识别扫描模块的功能是从指令解释执行模块的结果中提取下一条控制转移指令;分支指令劫持模块的功能是对指令识别扫描模块所得到的指令进行插桩。DIEB的整体运行流程如图1所示。

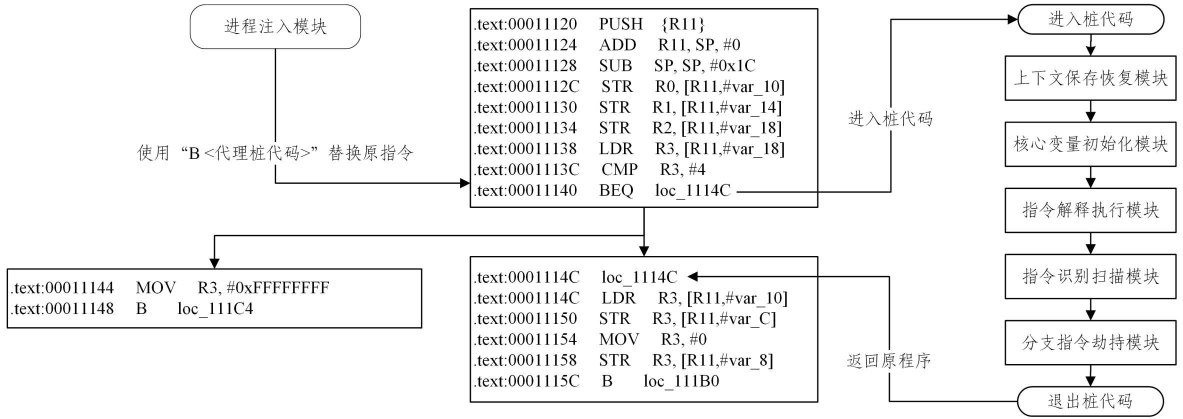

图 1 DIEB 运行流程示意图

Fig. 1 Schematic diagram of DIEB operation process

3.2 执行流跟踪方法

DIEB 的执行流跟踪方法通过桩代码的指令解释执行模块、指令识别扫描模块和分支指令劫持模块实现。

根据 CPU 流水线模型，如果 CPU 没有收到异常信号，则会将两条相邻的控制跳转指令之间的所有指令顺序执行一遍。根据代码基本块模型，只要基本块中第一条指令被执行，那么基本块内所有指令都会顺序执行一次。并且在内存空间中，由于控制转移指令的目的地址一般与控制转移指令自身的地址不相邻，因此控制转移指令常常被作为代码基本块的分界点。本文将控制转移指令作为探针来跟踪目标进程的执行流。

DIEB 的指令解释执行模块首先获得当前探针指令的目的地址，由指令识别扫描模块对该目的地址进行扫描，获取下一条探针指令，而后分支指令劫持模块对下一条探针指令进行劫持。当目标进程运行到被劫持的探针指令时，从原有的执行流中跳转至桩代码，并依次执行指令解释执行模块、指令识别扫描模块和分支指令劫持模块。重复上述过程，DIEB 可以实现对目标进程执行流的跟踪。

由于外部库函数一般是开源的第三方组件，并且也不是动态分析的主要对象，因此为提升插桩效率，本文不对链接的外部库函数进行插桩。但这会出现一个问题，当桩代码发现目标进程调用外部库函数时，一旦放弃对库函数的插桩，在库函数调用结束后也就丢失了对目标进程执行流的跟踪。

根据控制转移指令在执行的同时是否会将返回地址存入 R14 寄存器，本文将控制转移指令划分为带链接的控制转移指令和不带链接的控制转移指令。通常情况下，外部库函数的调用都是通过带链接的控制转移指令完成的。为了让函数在调用完毕后可以返回到调用之前的位置，带链接的控制转移指令在执行的同时会将返回地址保存在 R14 寄存器中，不带链接的控制转移指令则不会将返回地址保存在 R14 寄存器。

DIEB 的指令识别扫描模块在扫描过程中若发现带链接的控制转移指令，则会将其替换为指令“BL 〈代理桩代码〉”，并继续扫描，直到发现不带链接的控制转移指令才结束，同时使用“B 〈代理桩代码〉”替换不带链接的控制转移指令。如果发生了外部库函数调用，虽然在调用过程中桩代码失去对目标进程的监控权，但由于已经劫持该外部库函数调用指令之后的不带链接的控制转移指令，在外部库函数调用结束后，桩代码可以继续跟踪目标进程。执行流跟踪的算法如算法 1 所示。

算法 1 执行流跟踪算法

输入：当前指令内存地址 cur_addr

输出：目的分支内存地址 dst_addr

```
1. code_hex，code_type←get_code_info(cur_addr)
   /* 获取当前指令的二进制代码和操作码类型 */
2. dst_addr←emulated_code(code_hex，code_type)
   /* 解释执行当前指令，获得目的分支内存地址 */
3. tmp_addr←dst_addr
4. while(1)/* 循环扫描目的分支内存地址 */
5.     code_hex，code_type←get_code_info(cur_addr)
   /* 获取被扫描指令的二进制代码和操作码类型 */
6.     if code_type !=控制转移指令：
7.         tmp_addr+=4
           /* 在 ARM 指令集中，单条非 THUMB 指令长度为 4 字节 */
8.         continue/* 若该指令不是控制转移指令，则继续判断下一条指令 */
9.     if code_type==不带链接的控制转移指令：
10.         ins_this_code(tmp_addr，code_type)
            /* 使用“B 〈代理桩代码〉”指令劫持该指令 */
11.         push_addr_to_stack(tmp_addr)
            /* 将该指令内存地址保存到栈结构中，当击中该指令时，可通过 pop_addr_from_stack 函数从栈结构中读取指令内存地址 */
12.         break/* 结束扫描，退出循环 */
13.     if code_type==带链接的控制转移指令：
14.         ins_this_code(tmp_addr，code_type)
            /* 使用“BL 〈代理桩代码〉”指令劫持该指令，当击中该指令时，可通过读取 R14 寄存器获取指令内存地址 */
15.         tmp_addr+=4
16.         continue/* 继续循环扫描下一条指令 */
17. return dst_addr/* 返回目的分支地址，算法结束 */
```

通用 PC 平台上对代码基本块的识别，主要是通过递归下降技术来解析汇编语言指令流，或者在目标进程动态运行的过程中，通过 JIT 模式或者解释模式实时翻译原指令，从而

识别基本块的边界。但若对目标进程的所有指令进行翻译执行,将增加设备软硬件资源的占用率。DIEB的执行流跟踪方法不需要对目标进程所有指令的具体执行过程进行监控,只需监控与执行流相关的控制转移指令,从而在资源紧张的嵌入式设备上获取目标进程的基本块粒度的运行状态,提高插桩效率。

3.3 指令解释执行方法

由于作为探针的控制转移指令的目的地址一般与控制转移指令自身地址不相邻,为了跟踪目标进程执行流,必须获取待插桩的原始控制转移指令的目的地址。

在此过程中,首先根据ARM指令集的条件码,判定原始控制转移指令是否应被解释执行。在原始指令被判定为应该执行的前提下,通过解释执行原始指令获取原始指令的目的地址。若原始指令被判定为不应该执行,则其目的地址就是与自身地址相邻的下一条指令。在ARM指令集中,一条指令定长为4字节,即32 bit,对于被判定为不应该执行的原始指令,原始指令的目的地址就是自身地址加4。

3.3.1 判断原指令条件码

在ARM指令集中设置了条件码,CPU根据每条指令的条件码和当前标志寄存器判断当前指令是否应被执行。如图2所示,地址0x122BC处的指令为“BEQ loc_12428”,“B”为指令操作码,表示这是一条跳转指令;“EQ”是指令条件码。根据上一条指令“CMP R3,#1”的标志寄存器结果确定地址0x122BC处的指令是否应被执行。上述两条指令的意义是,如果R3寄存器等于1,就跳转至地址0x12428处,否则执行与0x122BC相邻的地址——0x122C0处的指令。

```
.text:000122B4    LDR   R3, [R11,#var_10]
.text:000122B8    CMP   R3, #1
.text:000122BC    BEQ   loc_12428
.text:000122BC
.text:000122C0    LDR   R3, [R11,#var_10]
.text:000122C4    CMP   R3, #2
.text:000122C8    BNE   loc_122D8
```

提取条件码 “EQ”

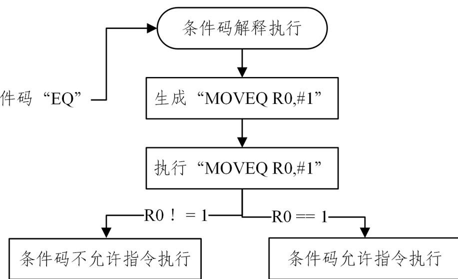

图2 条件码解释执行示意图

Fig. 2 Schematic diagram of condition code interpretation execution

条件码由指令的第28-31位这4 bit表示。本文采用对条件码进行解释执行的方法判断指令是否应被执行。具体地,提取原始指令的条件码作为测试指令的条件码,将测试指令置于被桩代码保存的上下文环境中执行,通过观察测试指令的执行结果来判断测试指令是否应被执行。由于桩代码保存的是目标进程执行到原始指令时的上下文环境,若测试指令可以正常执行,则说明原始指令的条件码允许原始指令执行,即原始指令应被解释执行。如图2所示,本文使用指令“MOV R0,#1”作为测试指令,提取原始指令“BEQ loc_12428”第28-31位的条件码,作为测试指令“MOV R0,#1”的条件码,生成新指令“MOVEQ R0,#1”。当桩代码保存的上下文环境满足条件码所要求的条件时,该指令的执行结果是R0寄存器被赋值为1,说明原始指令应被解释执行。条件码解释执行的流程如图2所示。

ARM指令集的条件码有16种,根据标志寄存器的值,采用逐一比对的方式解析条件码的语义效率较低。本文提出的解释执行方法的处理只需3步:首先,提取原始指令条件码并生成测试指令;其次,执行测试指令;最后,判断测试指令结果。在需要频繁插桩的情况下,该方法可以有效提高插桩效率。

3.3.2 解释执行原指令

如果原指令被判定为不执行,则直接将原始指令的下一条指令地址作为目的地址。对于判定执行的原指令,本文提出指令解释执行方法用于获取原指令的目的地址。在指令解释执行区域为待解释执行的指令设置与实际执行相同的上下文环境,并在指令解释执行区域中解释执行原始指令,获取其目的地址。

与x86指令集架构不同,ARM指令集存储当前执行指令地址的寄存器是R15寄存器,也被称作指令寄存器。ARM指令集中可修改指令寄存器值,或者说可以控制CPU跳转执行的指令均可作为控制转移指令,包括数据处理指令、数据加载指令和分支指令。为方便指令的解释执行,进一步将上述指令细分为四大类18种指令操作码,如表1所列。

表1 ARM控制转移指令分类

Table 1 ARM control transfer instruction classification

控制转移指令分类	指令操作码助记符
数据处理指令	ADC
	ADD
	BIC
	MVN
	SUB
	SBC
	RSB
	RSC
	MOV
	EOR
	MUL
	AND
以寄存器索引的跳转指令	BLX
	BX
以立即数索引的跳转指令	BL
	B
数据加载指令	LDM
	LDR

根据上述指令操作码的划分,本文所使用的指令解释方法分为以下4种。

1)对数据处理指令的解释执行

本文设计的指令解释执行策略是将原始指令指定的Rn源寄存器、Rd目的寄存器、Rm移位寄存器分别设置为通用

寄存器R3,R1,R2。在ARM指令集中,通用寄存器一般用于函数传参和算术运算,用通用寄存器替代原始指令中的Rn,Rd,Rm寄存器,被修改的指令与原始指令在算术功能上是相同的,这种替代不会改变指令的计算结果。此外,最终的执行结果保存在R1寄存器中,方便桩代码读取。完成对原始指令的修改后,从桩代码保存的进程上下文中恢复原始指令指定的Rn源寄存器和Rm移位寄存器值,并分别赋值给R3和R2两个寄存器。最后,执行被修改后的原始指令,R1寄存器保存的就是原始指令的目的地址。解释执行数据处理指令的策略如图3所示。

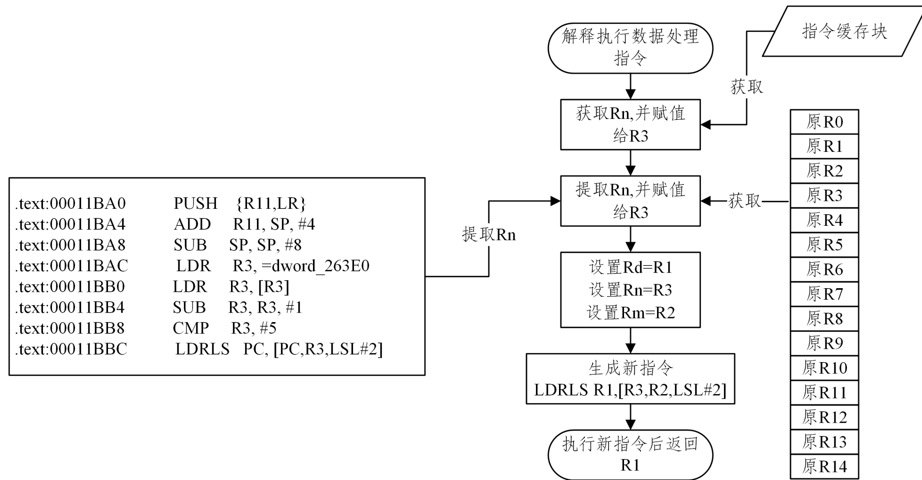

图3 解释执行数据处理指令示意图

Fig. 3 Schematic diagram of interpretation execution of data processing instructions

2)对以寄存器为索引的跳转指令的解释执行

该类指令主要包括BX和BLX指令。根据原指令第0—3位的值,即索引寄存器的名字,在保存的进程上下文中获取对应寄存器的初始值,即为目的跳转地址。SP寄存器(栈顶指针寄存器)和R15寄存器这两个特殊的寄存器在进程进入桩代码时没有被桩代码作为进程上下文环境保存。由于正在进行指令解释执行过程中的FP寄存器是进入桩代码前的SP寄存器减去一个固定数值得到的,因此进入桩代码前的SP寄存器可通过正在进行指令解释执行过程中的FP寄存器加上这个固定数值恢复。进入桩代码前的R15寄存器的值,即被解释执行的原始指令地址,可以通过指令缓存块中存储的原始指令地址恢复。通过上述策略,在指令解释执行区域恢复了BX,BLX指令的进程上下文环境,并通过解释执行获取了原始指令的目的地址。

3)对以立即数为索引的跳转指令的解释执行

这类指令主要包括B和BL指令。这类指令的解释执行策略较简单,与指令执行结果相关的上下文环境只有原始指令的地址,该地址可以通过指令缓存块获得,而后获取原指令第0—23位的值,符号扩展后乘4,再加上原始指令地址,即可得到目的跳转地址。

4)对数据加载指令的解释执行

数据加载指令主要是LDM指令。该指令将Rn所指地址的数据批量加载到目的寄存器列表中。其中,原始指令的第16—19位表示为Rn寄存器;第0—15位表示为目的寄存器列表,其中的每一个bit位代表一个目的寄存器;原指令的第24位表示加载数据到寄存器时的索引策略,当第24位为1时表示为前索引,即先索引数据再赋值到寄存器,第24位为0时表示为后索引,即先赋值到寄存器再索引数据;原指令的第23位表示前向加载还是后向加载。对LDM指令的解释执行策略是根据LDM指令的语义,在完成上下文环境的恢复后,对LDM指令进行解析并获取指令的目的跳转地址。

3.4 线程调度方法

本文使用以控制转移指令为探针的探测模式对目标进程进行动态插桩。在多线程环境下,由于代码段是所有线程的共享内存区域,因此探测模式对探针指令的修改或替换可能破坏线程的内存读写等原子性操作。不仅如此,如果某个线程错误地访问了被探针指令替换的指令,进而干扰了原有的执行流,则会导致进程崩溃。为了能够在多线程环境下跟踪目标进程,本文设计并实现了线程调度方法,在多线程环境下安全地进行指令劫持和指令恢复操作,解决多线程环境下的代码段竞争访问的问题。

指令劫持的目的是使桩代码获得目标进程的控制权,进而进行后续的动态插桩。本文采用探测模式对目标进程插桩,并使用"B〈代理桩代码〉"或"BL〈代理桩代码〉"替换作为探针的指令。当目标进程执行到探针指令的位置时,会执行被替换的指令,进入桩代码。

指令恢复的目的是在目标进程进入桩代码后,将被替换的探针恢复为原始指令,防止其他线程意外进入桩代码而引发错误。

3.4.1 *指令劫持方法*

在探测模式下,进程在进入桩代码前,需要完成运行状态上下文的保存、设置栈帧和跳转至桩代码等工作,一般需要多条指令才能完成。如果是单线程环境,那么通过多条指令来完成上述工作是没有问题的。但是对于多线程环境,这种处理并不可行。

举例来看,在多线程环境下,用于劫持的第一条指令为A指令,第二条指令为B指令。当被DIEB跟踪的线程执行到A指令,该线程会继续执行B指令,然后进入桩代码。插桩操作对代码段的修改会对所有线程造成影响,用于插桩的A指令和B指令也可能被其他线程执行。如果用于劫持的

A 指令和 B 指令不在同一个基本块内，则可能有线程会在跳过 A 指令的情况下直接执行 B 指令，从而破坏插桩动作的原子性。该线程不能按预设逻辑进入桩代码，而是会跳转至未知地址，线程崩溃，进而导致整个进程结束。

ARM 指令集中，不存在类似于 x86 指令集中“CALL”的指令，指令不会在发生跳转的同时将返回地址保存在内存中。ARM 指令集只能使用 BL，BLX 等指令在发生跳转的同时将返回地址保存在 R14 寄存器中。如果用 BL，BLX 等指令替换探针指令，则当进程执行到探针指令的位置时，因为 BL，BLX 等指令会改写 R14 寄存器的值，则 DIEB 的桩代码无法获取并保存完整的进程运行上下文环境。

为解决上述问题，DIEB 设置了存储指令地址的指令缓存块，采用先进后出的栈结构进行管理。根据 CPU 流水线模型，在获取控制转移指令的目标跳转地址后，预先将该地址之后的第一条不带链接的控制转移指令的地址压入指令缓存块的栈结构，并使用指令“B〈代理桩代码地址〉”对这条指令进行劫持。当进程从下一条不带链接的控制转移指令的地址处进入桩代码后，可从指令缓存块的栈结构中弹出一个地址数据，这个地址数据即为进入桩代码前的原指令地址。对于带链接的控制转移指令，可直接使用指令“BL〈代理桩代码地址〉”进行替换，当命中此类桩点时，根据 R14 寄存器的值直接获取原始指令地址。由于原始指令是带链接的控制转移指令，执行原始指令会在跳转的同时，将其相邻的下一条指令地址赋给 R14 寄存器。通过上述操作，无论是执行原始指令还是执行被替换后的指令，R14 寄存器总会被赋值为原始指令的下一条指令地址。

如图 4 和图 5 所示，当扫描到 0x110B0 时，将对 0x110D8 和 0x110E8 插桩，并将 0x110E8 压入栈中。当扫描到 0x109CC 时，将对 0x109D4 插桩并将 0x109D4 压入栈中。在执行到 0x109D4 时会将栈中的 0x109D4 弹出，此时栈中指针指向 0x110E8。当 recv 函数返回时，命中 0x110E8 地址的桩点，会将 0x110E8 从栈中弹出，桩代码可以正常解析。

```
.text:000110B0 00 48 2D E9      PUSH    {R11,LR}
.text:000110B4 04 B0 8D E2      ADD     R11, SP, #4
.text:000110B8 10 D0 4D E2      SUB     SP, SP, #0x10
.text:000110BC 08 00 0B E5      STR     R0, [R11,#fd]
.text:000110C0 0C 10 0B E5      STR     R1, [R11,#buf]
.text:000110C4 10 20 0B E5      STR     R2, [R11,#n]
.text:000110C8 10 20 1B E5      LDR     R2, [R11,#n]        ; n
.text:000110CC 00 30 A0 E3      MOV     R3, #0              ; flags
.text:000110D0 0C 10 1B E5      LDR     R1, [R11,#buf]      ; buf
.text:000110D4 08 00 1B E5      LDR     R0, [R11,#fd]       ; fd
.text:000110D8 3B FE FF EB      BL      recv
.text:000110D8
.text:000110DC 00 30 A0 E1      MOV     R3, R0
.text:000110E0 03 00 A0 E1      MOV     R0, R3
.text:000110E4 04 D0 4B E2      SUB     SP, R11, #4
.text:000110E8 00 88 BD E8      POP     {R11,PC}
```

图 4　通过 plt 表调用 recv 函数的示意图

Fig. 4　Schematic diagram of calling recv function through plt table

```
.plt:000109CC                   ; ssize_t recv(int fd, void *buf, size_t n, int flags)
.plt:000109CC                   recv                          ; CODE XREF: sub_110B0+28↓p
.plt:000109CC 00 C6 8F E2       ADR     R12, 0x109D4
.plt:000109D0 15 CA 8C E2       ADD     R12, R12, #0x15000
.plt:000109D4 B0 F6 BC E5       LDR     PC, [R12,#(recv_ptr - 0x259D4)]! ; __imp_recv
```

图 5　plt 段中的 recv 函数示意图

Fig. 5　Schematic diagram of recv function call in plt segment

DIEB 使用栈结构保存原始指令的地址，实现了用单一指令替换不带链接的原始控制转移指令，同时解决了多线程条件下使用多条指令替换原始指令可能破坏插桩动作原子性的问题。此外，先进后出的栈结构符合 C/C＋＋函数调用约定，进行内存分配、数据增删查改等操作非常简便，使用 DIEB 进行高频次的插桩操作时可以显著提升插桩效率。

3.4.2　指令恢复方法

在单线程环境下，被劫持的指令在执行后可以不恢复原指令。但在多线程环境下，不恢复原指令的策略会存在风险。

如图 6 所示，被监控线程 A 劫持了地址 0x110E8 处的指令“POP {R11，PC}”，线程 B 没有被 DIEB 监控，因此桩代码的核心变量初始化模块没有对线程 B 进行初始化。如果线程 B 执行到了 0x110E8，会发生崩溃。针对此问题，DIEB 采用的方法是当线程由替换的指令进入桩代码后，立即把替换的指令恢复为原始指令。如图 6 所示，线程 A 从地址 0x110E8 处进入桩代码，0x110E8 处的原指令将在第一时间被恢复。

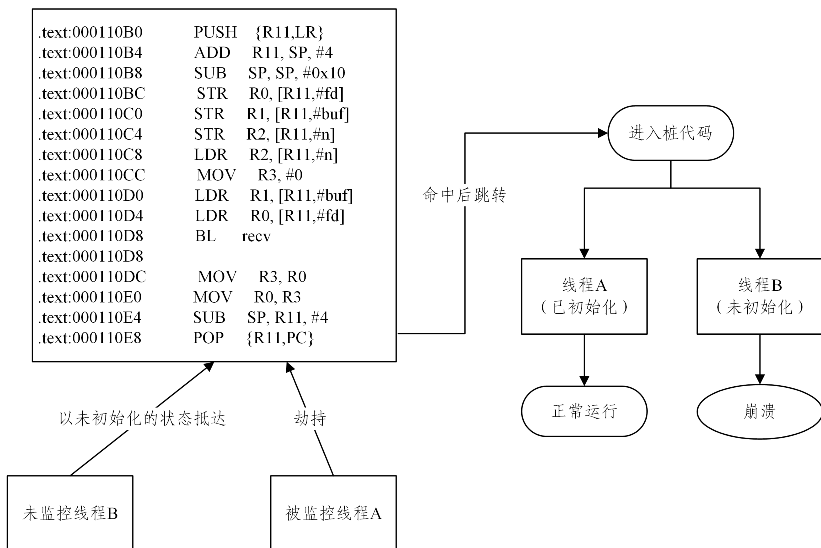

图 6　不恢复原指令导致线程竞争的示意图

Fig. 6　Schematic diagram of thread contention caused by not restoring the original instructions

在多线程环境下进行插桩操作需要格外注意代码段的竞争访问问题。DIEB 的指令劫持方法使用单一指令替换原始指令，保证了在多线程环境下进入桩代码的原子性，同时通过指令恢复方法，在进入桩代码后及时恢复

原始指令，保证了桩代码内部环境的稳定。

4 实验

本章对所提方法进行了测试实验，对 DIEB 的稳定性和有效性进行了验证。

4.1 方案设计

测试实验考虑的主要性能指标是时间膨胀率和多线程环境运行的稳定性，分别在小型家用路由器（Small Office，Home Office）NetGear R7000 实体设备上和基于 QEMU 的 ARM 仿真环境下进行。

时间膨胀率指由插桩导致的进程运行时间增量与原始运行时间的比值。对时间膨胀率的实验方法是比较被 DIEB 插桩前后的 NetGear R7000 设备对 HTTP 请求的响应时间差。通过测算时间膨胀率，可以准确评估 DIEB 的插桩效率。

为了提高测试效率，动态分析方法通常采用多线程并发测试，被测试的目标进程大多在多线程的环境下运行，因此需要测试 DIEB 在多线程环境下运行的稳定性。对多线程环境运行稳定性测试的实验方法是对被 DIEB 插桩后的网络代理转发程序进行高频次、大流量的代理转发请求，记录目标进程的运行状态。

4.1.1 测试对时间膨胀率的影响

实验中通过记录每次请求 NetGear R7000 的 Web 接口的响应时间差来计算 DIEB 的时间膨胀率。

首先通过逆向分析 NetGear R7000 固件的 httpd 程序，确定表 2 所列的 12 个有效 URL 的处理函数地址。实验选择在嵌入式设备上进行，并与基于 ptrace 的插桩方法进行比对。

表 2 NetGear R7000 设备固件 httpd 文件插桩前后运行时间

Table 2 Runtime of device firmware NetGear R7000 before and after httpd file instrumentation

目标 URL	插桩次数	原运行时间/s	本平台运行时间/s	ptrace 插桩运行时间/s
http://192. 168. 1. 1/pforward. cgi	67877	0.016	1.962	9.839
http://192. 168. 1. 1/fwSchedule. cgi	73373	0.122	2.124	9.972
http://192.168.1.1/det-wan. cgi	9096	0.017	0.278	1.696
http://192.168.1.1/wiz_fix2. cgi	14737	0.018	0.437	2.714
http://192. 168. 1. 1/ddns. cgi	265965	0.144	7.563	34.746
http://192. 168. 1. 1/se-curity. cgi	28937	0.310	1.033	6.599
http://192. 168. 1. 1/pptp. cgi	14459	0.014	0.362	4.753
http://192. 168. 1. 1/l2tp. cgi	11883	0.015	0.342	3.924
http://192. 168. 1. 1/routinfo. cgi	17588	0.016	0.494	4.486
http://192.168.1.1/key-word. cgi	71051	0.020	1.999	11.299
http://192.168.1.1/fw_serv. cgi	40671	0.017	1.141	7.245
http://192.168.1.1/fw_serv_edit. cgi	58007	0.018	1.710	7.867
Average	56137	0.060	1.621	8.761

ptrace 系统调用是在 Linux 和其他类 Unix 操作系统中使用的系统调用。它允许一个进程监测和控制另一个进程的执行，通常被用于调试（Debug）或跟踪（Trace）程序的执行流程。调试进程通过 ptrace 系统调用可以调试进程并观察它的内存和寄存器状态，同时调试进程还可以通过 ptrace 系统调用修改被调试进程的内存数据和注入代码。

基于进程调试的 ptrace 插桩方法可以在嵌入式设备实现有限的插桩，但目前除了基于 ptrace 的插桩方法，尚无其他动态插桩方法可以在嵌入式设备上运行，因此实验的对比对象是插桩前的原始响应时间、使用本方法插桩的响应时间和基于 ptrace 插桩的响应时间。为了更好地对比分析 DIEB 对时间膨胀率的影响，实验使用 ptrace 对 httpd 文件的所有跳转指令进行插桩，而非使用单步执行的方式对所有指令进行插桩，从而可以在插桩次数一致的情况下分析 DIEB 和 ptrace 的性能指标。

4.1.2 测试多线程运行环境下的稳定性

使用 earthworm 对 DIEB 进行测试，评估其在多线程环境下稳定运行的能力。Earthworm 是多线程代理转发程序，可以支持高并发的网络代理链接，在代理远程桌面服务、SMB 协议等时有着快速稳定的表现。本次实验所使用的 earthworm 程序使用无阻塞式的 select 网络编程方法，在通信信道没有数据的情况可返回 0，而阻塞式的 select 网络编程方法在通信信道没有数据的情况下进程会进入等待状态。在无阻塞式的 select 网络编程方法下，进程不会进入等待状态，而在高并发的代理请求的情况下会使用多线程快速重复执行相同的执行流，不同线程同时执行同一代码基本块的概率极大，极易引起代码段竞争访问问题。

此次实验在基于 QEMU 的 ARM 仿真环境中完成。首先测试在不插桩的情况下，对于大小为 0x100，0x200，x400，0x80，0x1000，0x2000，0x4000，0x8000，0x10000，0x20000，0x40000，0x80000，0x100000，0x200000，0x400000，0x800000，0x1000000 的数据包的收发时延。每种大小的数据包测算 10 次，取平均值，将结果记录至文件。然后在经过 DIEB 系统的插桩后，使用同样方法进行测试。最后对上述结果进行比较。

4.2 结果分析

表 2 列出了在 NetGear 实体设备上进行动态插桩的响应时间结果。通过分析发现，经过 DIEB 插桩后的响应时间大致是原始时间的 27 倍；而经过基于 ptrace 的插桩后的响应时间是原始时间的 146 倍，约是 DIEB 的 5 倍。从表 2 中的实验结果可以看出，虽然经过 DIEB 插桩后的时间膨胀率较大，但由于实体设备本身的性能和网络时延的影响，插桩次数在万次数量级的条件下，DIEB 插桩后的响应时间远小于基于 ptrace 的插桩方法。这是由于基于 ptrace 的动态插桩方法需要频繁地通过系统调用陷入内核，而每进行一次系统调用需要保存 ring3 层的运行状态，等待内核调度资源，而后再恢复 ring3 层的运行状态，这一过程极大地增加了时间消耗。不仅如此，基于 ptrace 的动态插桩方法需要通过进程间通信获取程序运行状态和控制目标进程运行，而进程间通信又依赖于操作系统内核分配信道资源和进程调度，同样也增加了时间消耗。与基于 ptrace 的动态插桩方法不同，DIEB 在对目标进行动态插桩的过程中，始终运行在 ring3 层，不需要系统调用

陷入内核；同时，DIEB 的桩代码可以无需进程间通信，独立完成进程监控、运行状态获取等操作，减少了时间损耗。从表 2 的实验结果可以看出，DIEB 插桩前后处理时间的平均差值在 1 s 左右，性能损耗在可接受的范围内。

图 7 给出了在基于 QEMU 的 ARM 仿真环境上，对多线程代理转发程序 earthworm 插桩的时间膨胀率。通过分析发现，经过 DIEB 插桩的目标进程在高并发的状态下处理不同长度的数据包时性能稳定，且瞬时的最大时间膨胀率不超过 100%。随着数据包长度的增加，时间膨胀率逐渐稳定在 40%左右，验证了 DIEB 在多线程、高并发条件下的稳定性。

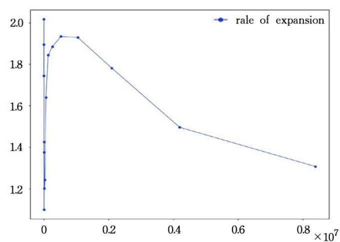

图 7　多线程环境下网络数据包的转发时间

Fig. 7　Forwarding time of network packets in a multi-thread environment

在 QEMU 仿真环境测得的时间膨胀率之所以远远小于在实体设备测得的结果，是因为 QEMU 本身的处理能力弱于实体设备，QEMU 自身性能不佳导致的时延误差对整体处理时延有较大的影响，甚至可能大于插桩带来的影响。

结束语　本文提出了一种面向嵌入式设备的动态插桩方法，实现在以精简指令集为主的嵌入式设备上对目标进程执行流的追踪。相比静态插桩，本方法可在目标进程动态运行的过程中更好地识别隐式控制流和隐式数据流，从而实现更精准的插桩。相比动态插桩，本文所使用的方法对运行时库和软硬件资源的依赖小，解决了其他动态插桩框架难以在嵌入式设备上进行插桩的问题。实验结果验证了 DIEB 的有效性和在多线程环境下运行的稳定性。但是，本文方法依然存在改进空间。首先，需要丰富用户调用的 API 接口，使 DIEB 更好地与动态测试框架结合。其次，DIEB 目前主要针对 ARM 指令集，需要进一步研究，使之适用于 mips，x86，x64 等指令集。

参 考 文 献

[1] KNUD L，MOHAMMAD H，SINHA S，et al. IOT ANALYTICS：State of IoT－Spring 2022[EB/OL].（2022-05-18）[2023-08-11]. https://iot-analytics. com/product/state-of-iot-spring-2022/.

[2] CHINA Communications Standards Association：Internet of Things Operating System Security White Paper（2022）[EB/OL].（2022-09-08）[2023-08-11] http://blog. nsfocus. net/wp-content/uploads/2022/09/iot-whitepaper. pdf.

[3] National Computer Virus Emergency Treatment Center：An Investigation Report on the Network Attack Incidents of Northwestern Polytechnic University by NSA of the United States [EB/OL].（2022-09-05）[2023-08-11]. https://www. cverc. org. cn/head/zhaiyao/news20220905-NPU. htm.

[4] Zalewski M：American fuzzy lop [EB/OL].（2017-11-04）[2023-08-11]. https://lcamtuf. coredump. cx/afl/.

[5] LUK C，COHN R，MUTH R，et al. Pin：Building customized program analysis tools with dynamic instrumentation [J]. Association for Computing Machinery，2005，40(6)：190-200.

[6] BRUENING D，GARNETT T，AMARASINGHE S，et al. An infrastructure for adaptive dynamic optimization[C]// International Symposium on Code Generation and Optimization. 2003：265-275.

[7] SRIVASTAVA P，PENG H，LI J，et al. FirmFuzz：Automated IoT firmware introspection and analysis[C]//Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things. 2019：15-21.

[8] ZHANG H，KAI L，XU Z，et al. SIoTFuzzer：Fuzzing Web Interface in IoT Firmware via Stateful Message Generation [J]. Applied Sciences，2021，11(7)：3120.

[9] ZHANG Y，HUO W，K P，et al. SRFuzzer：An automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities [C]// Proc. 35th Annu. Computer Security Applications Conf. 2019：544-556.

[10] KIM J，YU J，KIM H，et al. FIRM-COV：High-Coverage Grey-box Fuzzing for IoT Firmware via Optimized Process Emulation [J]. IEEE Access，2021，9：101627-101642.

[11] FENG X，SUN R，ZHU X，et al. Snipuzz：Black-box fuzzing of iot firmware via message snippet inference[C]// Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. 2021：337-350.

[12] NILO R，ANDREA C，DIPANJAN D，et al. DIANE：Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices[C]// 2021 IEEE Symposium on Security and Privacy. 2021：484-500.

[13] CHEN J，DIAO W，ZHAO Q，et al. IoTFuzzer：Discovering memory corruptions in IOT through APP-based fuzzing[C]// Proceedings of the 2018 Network and Distributed System Security Symposium. 2018.

[14] JANG D，KIM T，KIM D，et al. Dynamic Analysis Tool for IoT Device[C]// 2020 International Conference on Information and Communication Technology Convergence. IEEE，2020：1864-1867.

[15] ZHENG Y，SONG Z，SUN Y，et al. An efficient greybox fuzzing scheme for linux-based iot programs through binary static analysis[C]// 2019 IEEE 38th International Performance Computing and Communications Conference. IEEE，2019：1-8.

[16] ZHENG Y，DAVANIAN A，YIN H，et al. FIRM-AFL：High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation[C]// USENIX Security Symposium. 2019：1099-1114.

SI Jianpeng，born in 1996，postgraduate. His main research interests include cyber securiy and program analysis.

HONG Zheng，born in 1979，Ph.D，associate professor. His main research interests include cyber securiy and software reverse engineering.

（责任编辑：何杨）