

# 北京理工大学

## 本科生毕业设计(论文)

### 基于模糊测试的协议脆弱性研究

Research on Network Protocol Vulnerability via Fuzz Testing

学 院:	网络空间安全学院
专 业:	网络空间安全
班 级:	12112002
学生姓名:	常振轩
学 号:	1120202439
指导教师:	谭毓安

2024 年 5 月 18 日

## 原创性声明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导老师的指导下独立进行研究所取得的成果。除文中已经注明引用的内容外，本文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。

特此申明。

本人签名: \_\_\_\_\_ 日 期: \_\_\_\_\_ 年 月 日

## 关于使用授权的声明

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的,复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名: \_\_\_\_\_ 日 期: \_\_\_\_\_ 年    月    日

指导老师签名: \_\_\_\_\_ 日期: \_\_\_\_\_ 年 \_\_\_\_\_ 月 \_\_\_\_\_ 日

## 基于模糊测试的协议脆弱性研究

### 摘 要

固件是各种数字设备的核心控制软件，面临着日益严峻的安全问题。模糊测试已经被证明是一项十分有用的自动化测试和漏洞挖掘技术。由于固件程序往往需要特殊的执行环境，传统的模糊测试技术难以直接应用于固件程序。目前尚未出现有效的针对固件内的网络协议应用程序进行灰盒模糊测试的方法和工具。

本文从模糊测试的角度对协议脆弱性进行研究，提出一种基于QEMU插件进行系统级灰盒模糊测试的方法。该方法分为两部分，一是利用QEMU插件监测客户机系统内的系统调用，通过页目录地址识别目标测试进程，进而获取在灰盒模糊测试中用于指导变异的目标测试进程代码执行信息。二是利用网络请求对目标测试进程的状态进行探测和控制。该方法为实现以固件内网络应用为目标的灰盒模糊测试提供了基础。

基于上述方法，本文结合QEMU的AFL-SPY插件，在AFLNet的基础上开发出针对固件网络应用程序的系统级灰盒模糊测试框架AFLNetSpy。该原型系统通过获取客户机系统内目标程序代码执行信息，并利用网络请求探测和控制目标程序状态，能够对固件系统内的网络应用程序进行高效的灰盒模糊测试，成功将AFLNet的灰盒模糊测试能力扩展到固件网络协议脆弱性测试领域。本文通过实验从有效性、稳定性和性能三个方面对AFLNetSpy原型系统进行了分析验证。

**关键词：**灰盒模糊测试；固件脆弱性；网络协议

## Research on Network Protocol Vulnerability via Fuzz Testing

### Abstract

Firmware, the core control software of various digital devices, faces increasingly severe security challenges. Fuzzing has been proven to be a highly useful automated testing and vulnerability discovery technique. Due to the special execution environments required by firmware programs, it is difficult to apply traditional fuzzing techniques to firmware programs directly. Currently, there lacks effective methods and tools for conducting gray-box fuzzing of network protocol applications embedded within firmware.

In this thesis, we investigate protocol vulnerabilities from the perspective of fuzzing and propose a system-level gray-box fuzzing method based on QEMU plugins. The method consists of two parts. First, it utilizes a QEMU plugin to monitor system calls within the guest system and identify the target testing process by recognizing the page directory address, thereby obtaining execution information of the target testing process, which is used to guide mutation. Second, it leverages network requests to detect and control the state of the target testing process. This method provides the foundation for implementing gray-box fuzzing targeting network applications within firmware.

Based on the aforementioned method, we develop AFLNetSpy, a system-level gray-box fuzzing framework targeting network application programs within firmware, by integrating the AFL-SPY plugin of QEMU into AFLNet. By obtaining the code execution information of the target program within the guest system and utilizing network requests to detect and control the state of the target process, AFLNetSpy enables efficient gray-box fuzzing for network applications within firmware, which successfully extends the gray-box fuzzing capabilities of AFLNet to the field of firmware network protocol vulnerability testing and plays a significant role in prompting fuzzing research in the firmware field. We perform experimental analysis and validation of AFLNetSpy from three perspectives: effectiveness, stability and performance.

**Key Words: Gray-box Fuzzing; Firmware Vulnerability; Network Protocol**

## 目 录

摘 要 .....	I
Abstract.....	II
第 1 章 绪论 .....	1
1.1 研究背景 .....	1
1.2 研究意义 .....	4
1.3 国内外研究现状 .....	5
1.4 本论文组织结构 .....	9
第 2 章 相关工作 .....	10
2.1 AFL .....	10
2.2 AFLNet .....	12
2.3 TriforceAFL .....	13
2.4 DECAF-QEMU .....	16
2.4 ISPRAS-QEMU .....	16
2.5 本章小结 .....	17
第 3 章 系统级灰盒模糊测试 .....	18
3.1 系统级灰盒模糊测试方法 .....	18
3.1.1 代码执行信息的获取方法 .....	18
3.1.2 目标进程的状态探测和控制方法 .....	21
3.2 系统级灰盒模糊测试系统 .....	23
3.2.1 AFLNetSpy 架构 .....	24
3.2.2 QEMU-SPY 设计 .....	25
3.2.3 系统调用回调实现 .....	31
3.5 本章小结 .....	35
第 4 章 实验结果与分析 .....	36
4.1 环境构建 .....	36
4.2 有效性分析 .....	37
4.3 稳定性分析 .....	40
4.4 性能分析 .....	42

结 论 .....	46
参考文献 .....	47
致 谢 .....	49

## 第1章 绪论

### 1.1 研究背景

在当今信息技术高速发展的背景下，软件安全得到了广泛的关注。随着软件规模和复杂度的不断提高，依靠人工审查和手动测试来确保软件的正确性和安全性变得不再可行。因此，自动化软件测试技术尤其是模糊测试技术应运而生，并在近年得到越来越多的关注。通过不断改进和利用模糊测试技术，人们在桌面应用如浏览器、网络服务、办公软件等程序中找到了大量有意义的错误和漏洞，证明了模糊测试技术的实用性。

模糊测试(fuzz testing)是一种自动化测试技术<sup>[1]</sup>，它通过向程序输入随机或半随机的数据，观察程序的响应并检测异常行为。模糊测试的目标是发现软件中的潜在漏洞和错误，以改进软件的质量和安全性。模糊测试的一个重要优势是它可以发现那些由于输入组合庞大而难以穷尽的漏洞，从而使得测试更加全面和高效。模糊测试在各种应用程序中检测到了数千个错误和漏洞，已经被证明为是一种高效的漏洞发现挖掘技术。

关于模糊测试的分类有两种指标<sup>[2]</sup>，一是生成测试用例的方法，二是对待测程序信息的需求程度。

根据生成测试用例的方法，可将模糊测试划分为两类：①基于生成。基于生成的模糊测试是指模糊测试引擎根据用户规定的格式从头开始生成全新的测试用例。生成测试用例的过程可以基于随机算法、语法模型或其他规则。这种方法能够产生具有多样性和高覆盖性的测试用例，以探索待测程序的各个边界和异常情况。基于生成的模糊测试通常需要一定的领域知识和规范定义，以确保生成的测试用例符合预期的输入格式和语义；②基于变异。基于变异的模糊测试是指通过对已有测试用例进行变异来生成新的测试用例。用户首先提供一个或多个初始测试用例作为种子，然后通过对这些种子用例进行变异操作，如位翻转、删除、插入、替换等，生成新的测试用例。这种方法可以在已有测试用例的基础上进行迭代改进，更加高效地探索待测程序的执行路径和边界情况。基于变异的模糊测试通常需要一些变异操作的策略和机制来确保生成的测试用例具有多样性。

根据对待测程序信息的需求程度，可将模糊测试分为三类：①白盒模糊测试。白盒模糊测试需要使用待测程序的源码信息进行分析。测试人员或工具需要对待测程序的内部结构和逻辑有一定的了解，以便生成针对性的测试用例。白盒模糊测试能够更深入地探索程序中隐藏的路径和边界条件，发现潜在的漏洞和异常情况。为了获取源码结构信息，通常需要进行静态分析、符号执行或动态插桩等技术；②灰盒模糊测试信息。灰盒模糊测试不需要对待测程序的源码进行分析，但需要获取和使用其他能够反映测试程序执行情况的信息。例如，可以利用程序的运行日志、覆盖率信息、异常处理机制等来指导测试用例的生成和选择。灰盒模糊测试可以在不了解程序内部细节的情况下，通过观察程序的行为和响应来推测潜在的漏洞和异常情况；③黑盒模糊测试。黑盒模糊测试不需要待测程序的任何信息。测试人员只需要了解待测程序的输入输出规范和预期行为，而无需关注程序的内部实现细节。黑盒模糊测试可以模拟真实用户的输入，发现程序在不确定或异常输入下的响应情况。黑盒模糊测试是最常见和通用的模糊测试方法，适用于各种类型的软件和系统。

与传统的手动测试和静态分析相比，模糊测试具有许多优势。首先，它可以自动化执行，减少了人工测试的工作量。其次，由于输入是随机生成的，它可以探索更广泛的测试空间，涵盖更多的边界情况和异常情况，还可以帮助发现未知的漏洞。

嵌入式系统和物联网设备的兴起与智能化的发展密切相关。嵌入式系统是指内置在其他设备或系统中的计算机系统，它们通常运行在嵌入式处理器上，用于控制和管理设备的各种功能。物联网设备是指通过互联网连接的各种智能设备，如智能家居、智能城市基础设施、智能工业设备等。这些设备在现代社会中起着越来越重要的作用，涵盖了各个领域，包括交通、医疗、能源、环境监测等。嵌入式系统和物联网设备中的固件程序是实现设备功能的核心软件组件。固件程序通常负责设备的控制、通信、数据处理等任务。由于嵌入式系统和物联网设备的特殊特点，固件程序的安全性至关重要。一个存在漏洞的固件程序可能会导致设备失效、数据泄露、远程攻击等严重后果。因此，对嵌入式系统和物联网设备中的固件程序进行充分的测试和验证是至关重要的。

然而，传统的模糊测试技术主要针对桌面应用程序进行设计 and 应用，对于嵌入式系统和物联网设备中的固件程序来说，存在一些特殊的挑战<sup>[3]</sup>，这给模糊测试技术的应用带来了一定的困难：



(1) 嵌入式系统和物联网设备通常具有资源受限的特点。这些设备的处理能力和内存容量有限，无法承受大规模的模糊测试。传统的模糊测试技术通常需要大量的输入数据和计算资源来生成和执行测试用例，这在资源受限的环境下是不可行的。

(2) 固件程序通常在特定的硬件平台上运行，与桌面应用程序的执行环境有所不同。这意味着传统的模糊测试技术需要适应不同的硬件平台和操作系统，并针对特定的固件程序进行测试。这对测试工具和测试方法提出了更高的要求，需要充分考虑到嵌入式系统和物联网设备的特殊性。

(3) 嵌入式系统和物联网设备中的固件程序通常与外部环境和其他设备进行交互。这种交互可能涉及各种通信协议和接口，如无线通信、以太网、蓝牙等。传统的模糊测试技术需要能够模拟和生成各种通信协议和接口的测试数据，以覆盖不同的交互场景。这对测试工具和测试方法的设计提出了更高的要求。

(4) 固件系统中的程序崩溃不易检测。传统的测试方法依赖于程序的可观察到的崩溃，而常见的程序崩溃方式如内存故障等在嵌入式设备和固件系统中表现出的行为往往和桌面系统程序存在差异，难以进行监测。

针对以上挑战，研究人员和工程师们已经开始探索 and 开发适用于嵌入式系统和物联网设备的固件程序的模糊测试技术。一种常见的方法是通过测试用例生成和执行过程进行优化，以减少资源消耗。例如，可以利用针对嵌入式系统和物联网设备的特定特征和约束条件，设计高效的测试用例生成算法。同时，可以采用轻量级的测试工具和测试框架，以减少计算和存储资源的使用。另一种方法是针对特定硬件平台和操作系统进行测试工具和测试方法的设计和开发。这包括对硬件平台的深入了解，以便更好地理解固件程序的执行环境和特征。同时，还可以开发针对特定硬件平台和操作系统的模糊测试工具和测试框架，以提高测试效率和准确性。

此外，针对嵌入式系统和物联网设备中的通信协议和接口，也需要专门的测试技术和工具。这包括对通信协议和接口的解析和分析，以及针对特定协议和接口的测试用例生成和执行。可以利用模糊测试技术生成各种异常和边界情况的测试数据，以测试通信协议和接口的健壮性和安全性。另外，还可以考虑结合静态分析和模糊测试技术，以提高测试的效率和准确性。静态分析可以用于分析固件程序的代码和结构，发现潜在的漏洞和错误。结合模糊测试技术，可以针对性地生成测试用例，以验证静态分析的结果，并发现更多的漏洞和错误。

## 1.2 研究意义

随着物联网技术的快速发展和广泛应用，固件系统的安全性成为了一个日益重要的问题。固件是嵌入在各种物联网设备中的软件，负责控制设备的各种功能和操作。然而，由于固件系统通常存在漏洞和安全风险，黑客可以利用这些漏洞入侵设备，窃取用户信息，甚至控制设备进行恶意操作。因此，保护固件系统的安全对于确保物联网设备的可靠性和用户数据的隐私至关重要。

传统上，针对桌面系统程序的测试方法主要包括静态分析、动态分析和模糊测试。模糊测试是一种常用的自动化测试技术，通过向目标系统输入具有随机或半随机数据的输入，以尝试触发潜在的漏洞和错误。然而，由于固件系统的特殊性，传统的模糊测试技术和工具往往无法直接应用于固件系统。因此，将传统的模糊测试技术和工具扩展到固件系统，对于固件安全领域的发展具有重要的意义。

为了应对固件系统的特殊性，研究人员和安全专家们已经开始探索将传统的模糊测试技术和工具扩展到固件安全领域。一些针对固件系统的特定模糊测试工具和框架已经被开发出来，以支持固件系统的安全测试。这些工具和框架通常考虑了资源消耗、硬件架构和访问限制等因素，并提供了特定于固件系统的测试方法和技术。

本文提出了利用QEMU的全系统仿真模式和插件功能，收集代码执行信息和控制测试程序状态的方法。QEMU是一个开源的模拟器，可以模拟多种计算机体系结构的硬件环境。通过在QEMU中实现插件功能，可以监控和收集固件系统运行过程中的代码执行信息，从而为后续的模糊测试提供基础数据。此外，该方法还可以控制测试程序的状态，以便更好地触发潜在的漏洞和错误。基于这一方法，本文开发了原型系统AFLNetSpy，该系统支持对固件网络应用进行灰盒模糊测试。

AFLNetSpy是在已有的灰盒模糊测试工具AFLNet的基础上进行扩展而来的。AFLNet是一个专门用于网络应用和网络协议的灰盒模糊测试工具，通过模拟网络请求和响应，在初始测试用例的接触上通过随机变异生成具有新的测试用例，并能够监控应用程序的行为。通过将AFLNet扩展到固件网络应用领域，本文为固件系统的安全性提供了一种全新的测试方法。AFLNetSpy的开发不仅充分利用了QEMU的全系统仿真模式和插件功能，还针对固件系统的特殊性进行了相应的优化和改进。

这一研究成果具有重要的意义和广泛的应用前景。首先，AFLNetSpy为固件安全领域提供了一种新的测试工具和方法。传统的模糊测试技术和工具往往无法直接应

用于固件系统，而AFLNetSpy通过扩展现有的工具，充分考虑了固件系统的特殊性，为固件安全领域的研究和实践提供了一种创新的解决方案。其次，AFLNetSpy的应用可以提高固件系统的安全性和可靠性。通过对固件网络应用进行灰盒模糊测试，可以发现潜在的漏洞和安全风险，并及时采取相应的措施进行修复和改进，从而提高固件系统的抗攻击能力和稳定性。

此外，AFLNetSpy的研究成果还为后续固件安全领域的研究提供了可供借鉴的思路和方法。随着物联网技术的不断发展，固件安全的重要性也日益凸显。研究人员和安全专家需要不断探索和创新，以应对新的威胁和挑战。AFLNetSpy提出的基于QEMU的全系统仿真模式和插件功能的方法，为固件安全领域的研究提供了一种新的思路和技术手段。其他研究人员可以在此基础上进行进一步的研究和改进，推动固件安全技术的进一步发展。

最后，AFLNetSpy的研究成果对于推动物联网技术和固件安全技术的进一步发展具有积极的意义。随着物联网设备的普及和应用场景的不断扩大，固件安全的重要性将变得越来越突出。通过提供一种有效的测试工具和方法，如AFLNetSpy，可以帮助设备制造商和开发者提高固件系统的安全性和可靠性，从而促进物联网技术的健康发展。同时，通过推动固件安全技术的研究和创新，可以有效应对不断增长的网络安全威胁，保护用户的隐私和数据安全，推动物联网技术的广泛应用和进一步发展。

总的来说，本文提出的基于QEMU的全系统仿真模式和插件功能的方法以及开发的原型系统AFLNetSpy在固件安全领域具有重要的研究意义和实际应用价值。它不仅为固件安全领域提供了新的测试工具和方法，有助于提高固件系统的安全性和可靠性，而且为后续该领域的研究提供了可供借鉴的思路，对于推动物联网技术和固件安全技术的进一步发展具有积极意义。

### 1.3 国内外研究现状

1988年，Miller首次提出模糊测试的基本概念，即通过反复向程序提供随机输入数据来寻找崩溃<sup>[4]</sup>。之后的三十多年里，模糊测试技术不断发展，历经多个发展阶段，现已成为成为一种广泛使用的自动化测试和漏洞挖掘技术。

模糊测试发展早期，相关研究主要集中在黑盒测试<sup>[5][6]</sup>。在2013年AFL<sup>[7]</sup>诞生之后，基于覆盖率引导的灰盒模糊测试研究成为主流。AFL是由Google团队开发的一种基于变异的灰盒模糊测试工具，它通过插桩获取测试目标的代码执行信息，并在此基础上

实现覆盖率驱动测试用例选择和变异算法。本文将在2.1节详细介绍AFL的工作流程。除了AFL，Google还开源了支持对Linux等操作系统的系统调用进行模糊测试的工具Syzkaller<sup>[8]</sup>，并且建立了OSS-Fuzz<sup>[9]</sup>和ClusterFuzz<sup>[10]</sup>平台，促进了模糊测试在工业界的应用。

基于AFL，国内外研究者进行了大量的改进研究。其中，Marcel Böhme在AFLFast<sup>[11]</sup>中将基于覆盖率的灰盒模糊测试建模为对马尔科夫链状态空间的系统探索，并提出了几种加强对低频路径重视程度的搜索和调度算法，提高了灰盒模糊测试的效率。Chenyang Lyu等人在MOpt<sup>[12]</sup>中利用定制的粒子群优化(PSO)算法，找到算子在模糊测试有效性方面的最优选项的概率分布，并提供“起搏器”模糊测试模式，以加速PSO的收敛速度，实现模糊测试效率的提高。LAF-INTEL<sup>[13]</sup>提出将多字节数据比较转化为逐字节数据比较的方法，以解决模糊测试中的大数和字符串比较问题。RedQueen<sup>[14]</sup>基于KAFL<sup>[15]</sup>提出了绕过模糊测试中硬比较与校验和检查的方法。

AFL存在的一个主要限制是对输入数据的结构缺乏感知能力，只能实现比特级的变异操作。Peach<sup>[16]</sup>是一个能够感知测试数据结构的黑盒模糊测试工具，它能够通过用户规定的输入模型拆解有效的测试数据，并将得到的片段进行删除、修改和重新组合以得到新的测试数据。Pham等人在AFLSmart<sup>[17]</sup>中集成了Peach的输入结构组件和AFL的覆盖反馈组件，可实现对测试用例高度结构化的变异，并保证新测试用例的有效性，适用于遵循树结构的所有复杂数据格式。

AFL已于2021年停止维护，大量基于AFL的学术研究和改进工作的成果处于分散状态。为解决这个问题，Andrea Fioraldi等人提出AFL++<sup>[18]</sup>，它是一个社区驱动的开源工具，整合了大量基于AFL和最新的模糊测试研究成果。同时通过提供可自定义的变异API和插件机制，AFL++为未来的模糊测试研究提供了良好的基础设施，使得研究者能够快速开发原型系统以及尝试组合各种改进策略，并进行统一公平的评估，对促进模糊测试领域的健康发展具有重要意义。

尽管AFL++提供的自定义API和插件机制为后续的模糊测试研究提供了基础设施，但由于受制于AFL的基础代码框架，难以在软件工程水平上保证AFL++的扩展性。随着越来越多的改进算法加入到其中，AFL++项目的代码量逐渐增大，变得越来越难以维护，且延续了多个新分支间相互隔离的历史问题。

为此，AFL++的开发团队Andrea Fioraldi等人在AFL++之外，以可扩展性和可复用性为第一原则，使用RUST从头实现了LibAFL<sup>[19]</sup>项目。LibAFL由一组库组成，用

用户可以通过组合基于可扩展实体的库组件构建自定义模糊器。它实现这一目标要归功于几个因素：①易于扩展，LibAFL的设计目标之一是提供一个易于扩展的模糊测试框架。为了实现这一点，LibAFL利用了Rust编程语言的特性，如所有权系统、模式匹配和模块化设计。这些特性使得开发人员能够轻松添加新的功能、算法和组件，并能够快速构建和测试新的组件。此外，Rust的强类型系统和内存安全性保证也有助于减少错误和漏洞的可能性，提高代码的可靠性和稳定性；②基于对现代模糊器组件的分类，LibAFL提供了一组库组件，包括输入生成器、覆盖率分析器、变异器等。用户可以根据其需求选择和组合这些组件，构建出适用于特定应用场景的定制化模糊器。这种组件化的设计使得LibAFL更加灵活和可定制，可以根据用户的具体需求进行定制化的模糊测试；③充分利用RUST语言特性，LibAFL利用了Rust在编译时简单快速地序列化对象和组件插槽的功能，这种序列化机制使得LibAFL能够高效地处理和传递复杂的数据结构，提高了模糊测试的性能和效率。同时，这种序列化的方式也有助于减少运行时的开销，进一步提高了模糊测试的速度和响应能力；④开源驱动：作为一个开源项目，LibAFL积极关注研究和创新，并不断更新和改进其算法和功能。它与模糊测试研究社区保持紧密联系，吸纳最新的模糊测试技术和最佳实践。通过提供最新的算法和功能，LibAFL能够帮助用户在模糊测试过程中获得更好的测试覆盖和漏洞发现能力，对模糊测试领域的现代化发展以及打通学术界和工业界的壁垒具有重要意义。

除了提高模糊测试流程各个环节的效率，将模糊测试扩展到更多的领域也是主要研究方向之一。

网络应用和网络协议的安全性一直是互联网领域的关注焦点。由于网络应用和协议的复杂性和多样性，传统的测试方法往往无法覆盖所有潜在的漏洞和安全隐患。因此，灰盒模糊测试成为一种被广泛采用的方法，它通过模拟输入数据的多样性和异常性，尝试触发目标应用和协议中的漏洞和错误行为。在网络应用和网络协议领域，Thuan Pham等人率先提出AFLNet<sup>[20]</sup>，将AFL的灰盒模糊测试能力扩展到了网络应用和网络协议的测试中，本文将在2.2节详细介绍AFLNet的工作流程。Anastasios Andronidis等人在AFLNet的基础上提出改进策略，并实现SnapFuzz<sup>[21]</sup>原型系统，加快了网络应用模糊测试的效率。

在固件模糊测试领域，Marius Muench等人介绍了固件模糊测试的难点并给出一个监测固件应用内存错误的方法<sup>[22]</sup>，Jiongyi Chen等人在IoTFuzzer<sup>[23]</sup>工作中，通过收

集并分析固件厂商提供的APP和固件之间的通信数据,实现在无法访问固件镜像的情况下挖掘固件网络应用的内存问题。Hangwei Zhang等人提出SloTFuzzer<sup>[24]</sup>,对IoTFuzzer进行了状态扩展。NccGroup提出TriforceAFL<sup>[25]</sup>,率先将AFL的灰盒模糊测试能力成功应用于固件系统的系统调用测试中。TriforceAFL的关键创新在于结合了QEMU的全系统仿真能力。QEMU<sup>[26]</sup>是一种开源的虚拟化软件,可以模拟多种硬件平台和操作系统环境。通过与QEMU的结合,TriforceAFL可以在模糊测试过程中对固件系统进行全面的仿真和监控,以便更好地模拟真实设备环境和检测系统调用的执行情况。本文将在2.3节详细介绍TriforceAFL的工作流程。Yaowen Zheng等人在FirmAFL<sup>[27]</sup>和EQUAFL<sup>[28]</sup>中分别提出利用混合仿真模式和增强的QEMU用户仿真模式两种方法,以提高固件应用程序模糊测试的效率,然而这些方法对目标程序有一定的限制,存在不够普适的问题。

此外,还有一些基于机器学习和人工智能的模糊测试方法正在被研究和开发。这些方法利用机器学习算法实现测试用例进行生成、变异和选择,以提高模糊测试的效率和覆盖率。例如,使用遗传算法、强化学习等技术来指导测试用例的生成和变异。这些方法在某些情况下可以产生更好的测试用例,但也面临着挑战,如测试用例的可解释性、训练数据的获取等问题。

Ruijie Meng等人在ChatAFL<sup>[29]</sup>工作中,意识到大语言模型在训练过程中很可能已经提取了数百万页的人类可读的协议规范,可以在协议模糊测试期间利用大语言模型关于网络协议的知识,于是他们开发了一个由大模型引导的协议实现引擎供模糊测试使用,ChatAFL能够为目标协议中的没用中消息类型构建相应的语法,然后通过与大语言模型的交互来改变消息或预测消息序列中所需的下一条消息。

Yinlin Deng等人在TitanFuzz<sup>[30]</sup>工作中,提出直接利用大语言模型生成合法的代码片段,对深度学习库进行模糊测试的方法。他们认为现代大语言模型在其训练语料库中包含大量调用常见深度学习库的代码片段,因此可以隐式学习编程语言的语法和语义以及复杂的接口约束,从而生成合法的调用目标深度学习库的代码程序。随后,Yinlin Deng等人在FuzzGPT<sup>[31]</sup>工作中,通过利用大语言模型的微调和上下文学习能力,结合曾经触发错误的历史代码,进一步增强了利用大语言模型对常见深度学习库机型模糊测试的能力。Fuzz4all<sup>[32]</sup>是第一个通用的基于大语言模型的模糊测试器,它利用了一种新颖的自动提示技术,能够支持多种输入语言。Cen Zhang等人创新性地提出了利用大语言模型驱动测试任务构建的方法<sup>[33]</sup>。

总的来说，目前模糊测试领域，尤其是针对固件应用的模糊测试领域，发展十分迅速，但仍存在一些问题供后续研究者探索解决。

## 1.4 本论文组织结构

本论文的正文部分按照以下五个部分组织划分：

(1) 第一章：绪论。重点介绍本论文的研究背景和研究意义，并阐明了模糊测试和固件模糊测试领域的国内外研究现状。

(2) 第二章：相关工作。本章重点介绍模糊测试领域的知名工具AFL，以及基于AFL进行扩展得到的AFLNet和TriforceAFL。同时介绍DECAF和ISPRAS-QEMU两个利用QEMU插件系统实现虚拟机内省的相关工作。

(3) 第三章：系统级灰盒模糊测试。本章对本文提出的系统级灰盒模糊测试方法进行介绍，并对基于该方法实现的AFLNetSpy原型系统的架构和设计进行讲解。

(4) 第四章：实验结果与分析。本章首先介绍实验环境的准备工作，然后从有效性、稳定性和性能三个方面设计实验对AFLNetSpy原型系统进行分析验证。

(5) 结论。本章根据前文提出的方法和实验结果，对研究成果进行总结，并指出不足之处，同时对未来进行展望。

## 第2章 相关工作

本章选择了几项和本文研究内容最相关的工作，包括模糊测试领域的里程碑AFL、网络应用模糊测试领域的先驱性工作AFLNet、固件模糊测试领域的先驱性工作TriforceAFL，以及基于QEMU插件系统实现虚拟机内省的两项工作DECAF-QEMU和ISPRAS-QEMU，进行详细介绍。

### 2.1 AFL

AFL是Google团队开发的一个知名的灰盒模糊测试工具，通过收集测试进程的代码执行信息，并利用遗传算法，进行覆盖率引导的灰盒模糊测试。AFL支持两种使用模式，静态插桩模式和动态插桩模式。

静态插桩模式适用于有待测程序源码的情况。AFL对gcc等编译工具进行包装，得到afl-gcc等工具。使用AFL包装后的编译工具对待测程序进行编译时，AFL会在生成的程序中添加一些代码片段，用于在后续测试过程中收集代码执行信息，以及探测和控制程序运行状态，并在模糊测试过程中和afl-fuzz保持交互。静态插桩模式下，AFL的工作流程如下：

(1) 首先使用afl-fuzz命令启动模糊测试后，afl-fuzz首先会创建一个子进程并执行待测程序，待测程序启动后作为Forkserver和afl-fuzz保持通信。

(2) 之后的每一次测试前，afl-fuzz会通过管道请求Forkserver创建一个测试进程，并等待测试进程的进程号。

(3) Forkserver接收到afl-fuzz的请求后，会创建测试进程，并将其进程号通过管道返回给afl-fuzz，实现同步。

(4) afl-fuzz接收到测试进程号后，从测试队列选择或变异生成一个测试用例，并通过系统初始化阶段设置好的文件描述符将测试用例发送给测试进程。然后等到Forkserver返回测试进程的退出状态码，若定时器超时，则主动kill掉测试进程。测试进程执行过程中会不断更新trace\_bits数组，以记录代码执行情况。

(5) Forkserver会等待其子进程即测试进程退出，并将退出状态码返回给afl-fuzz进行分析，实现同步。

(6) afl-fuzz在接收到测试进程的退出状态码后，会检查通过共享内存和测试进程共享的trace\_bits数组的状态，并根据trace\_bits的状态和测试进程退出状态码对本次测



试用例的价值进行评判，如果触发了新的执行路径，则将该测试用例保存至测试队列。然后回到第(2)步，重复(2)-(6)的过程，直到用户手动停止afl-fuzz。如图2-1所示。

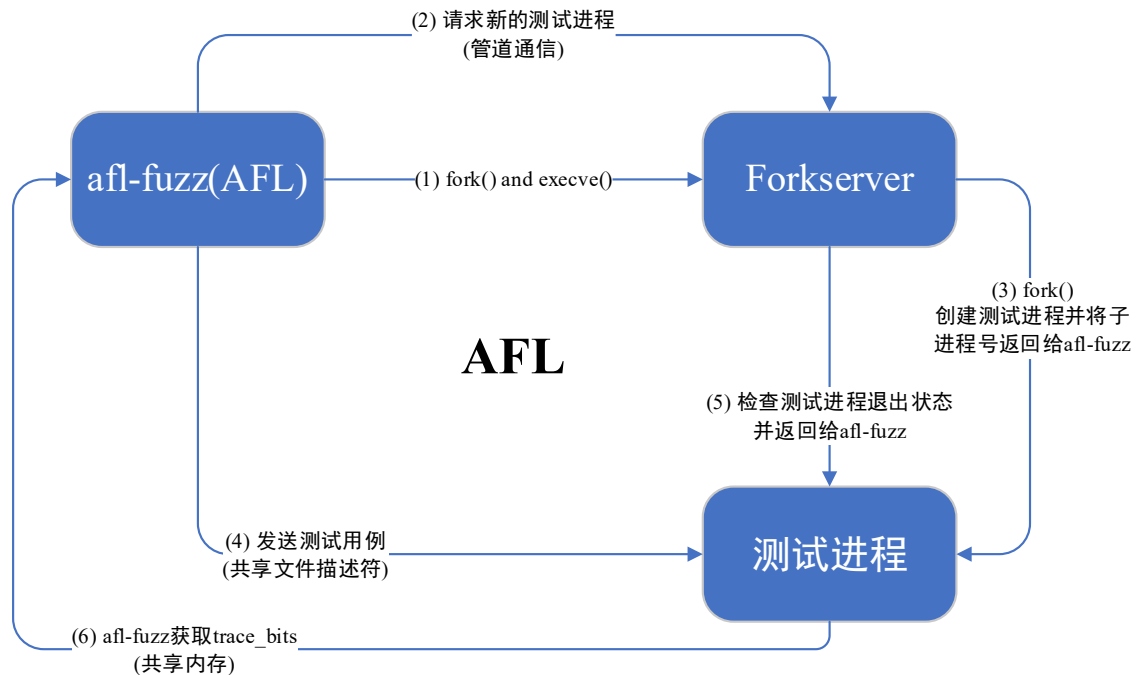


图 2-1 AFL 工作流程图

动态插桩模式适用于无法访问待测程序源码，只有二进制程序的情况。AFL在动态插桩模式下的工作流程和静态插桩模式基本一致，但是这种情况下由于无法通过在待测程序中插入二进制片段，没办法探测和控制测试进程的状态，也无法测试进程的代码执行信息。

为了解决这个问题，AFL引入并修改了用户模式的QEMU。修改后的QEMU可以发挥和静态插桩模式下Forkserver一样的作用，能够和afl-fuzz通过管道保持通信和状态同步。同时由于QEMU的翻译执行机制中翻译块的存在，代码执行信息的收集也变得可行。

另外需要注意的是，动态插桩模式的效率要比静态插桩模式要低，且动态插桩模式无法像静态插桩模式那样，根据跳转指令切分程序代码块，所得到的代码执行信息的意义不如静态插桩模式直观。但QEMU的使用使得测试不同架构下的二进制程序变得可行，扩大了AFL的适用范围。

## 2.2 AFLNet

AFLNet是Thuan Pham等人针对网络应用测试开发出的AFL扩展版。AFL只支持测试以文件或命令行的方式进行输入的程序，如果想要使用AFL测试某个网络应用，首先必须修改该应用的输入方式，使其能够从文件中读取网络请求。当没有源代码时，AFL就对该网络应用束手无策了。而AFLNet在AFL的基础上添加了网络通信支持，能够通过网络套接字直接向待测网络应用发送测试请求。同时，考虑到网络应用的错误往往需要一个序列的请求才能够触发，因此AFLNet不像AFL一次测试只发送一个请求，它支持在一个测试用例中存放多个请求报文，并在一次测试中按顺序发送它们。

AFLNet的工作流程的如图2-2所示，和AFL的工作流程基本一致。区别主要有两点：

(1) 一是第(4)步中AFLNet采用网络套接字即socket通信的方式向测试进程发送测试请求，然后接受响应，并提取响应中的状态码，状态码不仅可以用于协议状态机的分析，还能够和代码执行信息一起用来评估测试用例的价值和指导新测试用例的生成。

(2) 二是由于网络应用通常处于无限循环状态，处理完一次测试用例后并不会自动退出，为此AFLNet支持两种策略：①一个测试进程只处理一个测试用例，在一次测试结束后，由afl-fuzz主动向测试进程发送SIGTERM信号以关闭测试进程；②允许一个测试进程处理多个测试用例，直到出现超时或者崩溃，再启动一个新的测试进程。本文开发的原型系统AFLNetSpy就采用了第二种策略。

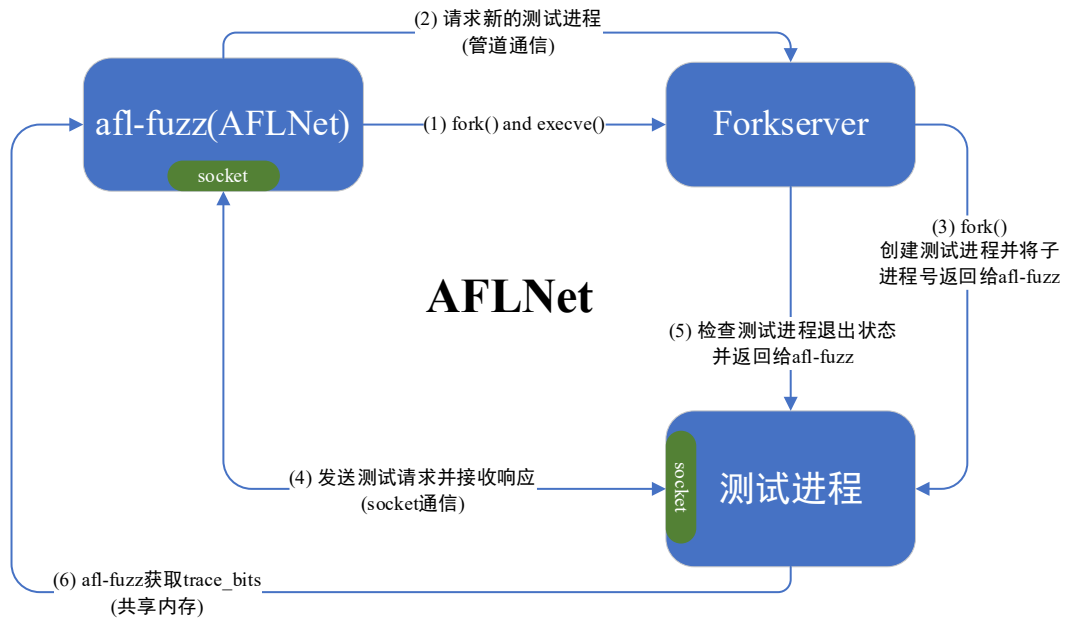


图 2-2 AFLNet 工作流程图

同时，AFLNet提出一个新的协议推断算法，通过观察AFLNet和待测网络应用间交换的请求响应数据，能够一定程度上还原出通信采用的协议格式。因此，作为一个协议测试工具，它并不需要用户手动指定协议规范和报文格式，可以直接将用户收集到的客户端和待测服务器的通信流量中的请求作为种子，在重放请求的过程中，自动学习协议知识。不过，当初始测试用例较少时，自动获取的报文格式难免会存在误差，导致变异生成的大部分测试用例仍然是无效的。

另外需要注意的是，AFLNet在发送一次测试用例后，会尝试接收响应。但是，以HTTP服务器为例，当测试请求无效时，服务器会直接丢弃该请求报文，并不会返回响应。因此，相比于AFL，AFLNet需要用户额外手动设置一个合适的等待响应的最大时间。

## 2.3 TriforceAFL

TriforceAFL是NccGroup针对固件系统调用测试开发出的AFL扩展版。它创新性地成功将QEMU的系统模式和AFL结合起来，使得TriforceAFL能够对QEMU模拟启动的固件系统中的系统调用进行高效的灰盒模糊测试。

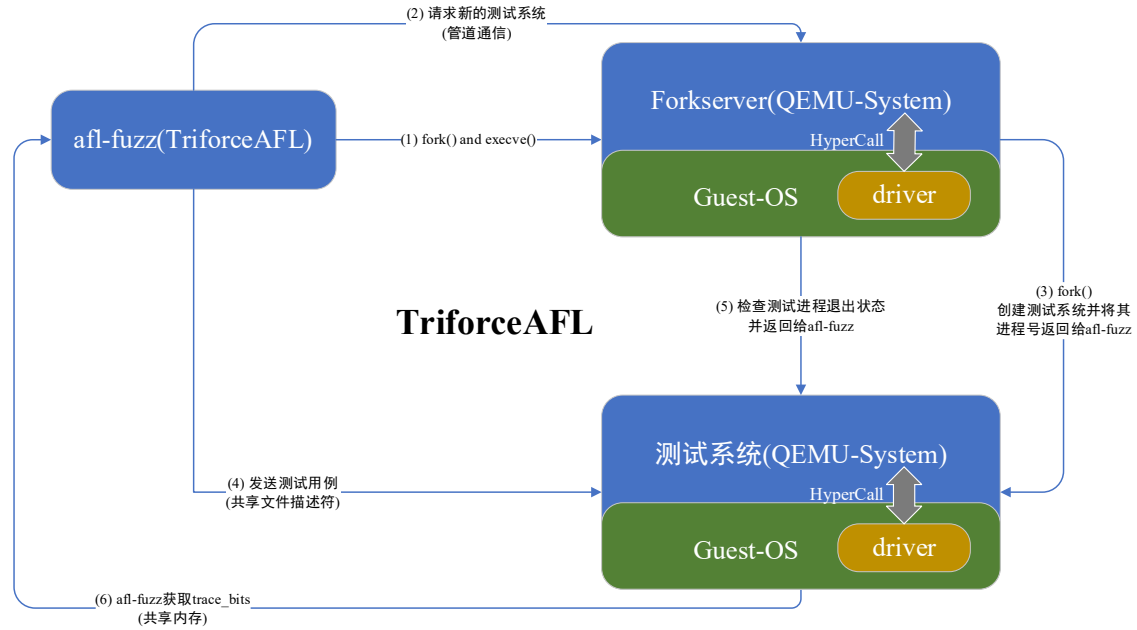


图 2-3 TriforceAFL 工作流程图

TriforceAFL的整体工作流程如图2-3所示，和AFL工作流程的主要区别，也即TriforceAFL重点解决的问题有三个：

(1) 一是测试用例的输入。在使用AFL对桌面应用程序进行的模糊测试中，能够通过dup()函数在父子进程之间，主要是afl-fuzz进程和测试进程之间，共享所需的文件描述符，并通过该文件描述符直接传递测试用例。然而运行在主机系统上的afl-fuzz，无法与运行在QEMU内部的客户机系统及其中的应用程序共享文件描述符，也就无法通过这种方式来传递测试用例。出于进行固件系统调用测试的目的，TriforceAFL采用的策略是，在固件系统中添加一个用户编写的driver程序，driver程序能够通过执行特殊的HyperCall指令与QEMU进行协同，而QEMU进程作为afl-fuzz的子进程，能够通过管道和afl-fuzz进程保持通信并共享文件描述符。于是，以QEMU为桥梁，客户机系统中运行的driver程序便能够接收到afl-fuzz发送的测试用例。

(2) 二是获取测试用例执行结果并退出测试系统。driver在接收到测试用例后，能够启动一个子进程作为测试进程执行测试用例对应的系统调用，并获取其执行状态，然后通过特定的HyperCall指令将执行状态以QEMU作为媒介，返回给afl-fuzz进程。同时，driver还会通过另一条特定的HyperCall指令，通知当前用于测试的QEMU进程退出，QEMU进程退出后，其内部用于本次测试的客户机系统自然也就实现了退出。

(3) 三是测试系统的状态恢复。由于系统调用测试的特殊性，异常的系统调用很可能导致客户机操作系统的崩溃。为保证模糊测试的持续运行，每一次测试结束后，必须重启客户机系统或设法恢复客户机系统状态至执行测试用例前。重启客户机系统耗时过长，显然不是一个可行的方案。TriforceAFL通过实现QEMU-System级别的Forkserver，实现了客户机系统状态的快照和恢复。在客户机系统启动完成并运行driver进程后，driver会通过一个特定HyperCall指令告知QEMU-System客户机系统已经就绪，然后QEMU-System就会保存当前的虚拟CPU状态，后续创建的每一个测试系统都将以此刻为运行起点。

需要注意的是，虽然NccGroup宣称依靠TriforceAFL可以实现“Fuzz Everything”，TriforceAFL也的确提供了创新性的思路，但本文经过测试，发现事实并非如此。目前依靠TriforceAFL仅仅只能够实现固件系统调用级别的灰盒模糊测试，要想实现固件内应用程序级别的灰盒模糊测试，还需要额外的工作。原因如下：

(1) 代码执行信息的收集：QEMU系统模式下，启动的客户机系统中往往同时运行着大量的应用程序，而QEMU只会根据当前虚拟CPU的状态来获取二进制代码块，并进行翻译和执行，无法直接判断出当前代码块属于客户机系统中的哪个进程。为此，TriforceAFL采用的策略是，通过修改固件的文件系统镜像和初始化脚本，保证客户机系统启动后其内部只有driver进程和其子进程即测试进程处在运行状态，然后通过统计高地址空间即操作系统使用的地址空间中的指令执行情况，获取测试用例对应的代码执行信息。显然，这种方式并不适用于固件内的普通应用程序。

(2) QEMU级别的Forkserver：TriforceAFL通过QEMU级别的Forkserver来保证每一次测试结束后，下一次测试执行前，能够恢复客户机系统的状态。然而，经过测试，这种方式并不能够在恢复客户机系统状态的同时恢复客户机内各个应用程序的执行状态。本文以HTTP服务器为例进行了测试，当客户机内的HTTP服务器接收到网络请求，即在QEMU层监测到该服务器进程调用accept系统调用时，仿照TriforceAFL的实现方式，退出vcpu线程，在主线程中执行fork()命令，得到的子进程无法继续处理刚才接收到的网络请求，且有可能触发各类段错误。因此，对于固件内应用程序级别的模糊测试，采用QEMU级别的Forkserver并不是可行的方案。本文将提供一种针对固件内网络应用的替代方案。

## 2.4 DECAF-QEMU

DECAF<sup>[34]</sup>是Andrew Henderson等人开发的一个动态可执行代码分析框架。DECAF为QEMU提供了一套易于使用的、事件驱动的插件机制，并基于该机制实现了实时的虚拟机内省即VMI功能。

动态二进制分析是程序分析研究中广泛采用且不可或缺的技术。在DECAF之前，尽管已经提出了几种动态二进制分析工具和框架，但它们都存在一些问题，包括性能下降严重、仅支持用户模式、缺乏API等。DECAF通过结合一种新颖的TCG指令级别的位粒度污染分析技术，实现对TCG指令的精细控制，并在此基础上，提供了三个与平台无关的插件，分别用于监测特定指令、监测API执行情况和监测键盘输入情况。尽管DECAF的主要目的是实现比特粒度、高准确性的TCG指令污点分析，但具备低开销VMI功能的DECAF-QEMU也可作为独立组件用于其它目的。

另外需要注意的是，DECAF-QEMU所使用的QEMU版本较低，难以启动较新的固件镜像如OpenBMC等镜像，且对操作系统的内核结构存在依赖，当特定数据结构出现变动时需要重新进行修改适配。但其实现方法仍然具有很大的参考价值，本文发出的原型系统AFLNetSpy所需的系统调用监测功能就借鉴了其实现思路。

## 2.4 ISPRAS-QEMU

ISPRAS-QEMU<sup>[35]</sup>是Ivan Vasiliev等人受ISPRAS组织支持开发出的一个基于QEMU的非入侵式虚拟机插桩和内省框架。它是一个用于对虚拟机内二进制程序进行动态分析的轻量级框架，具备跨系统和非入侵即不修改目标二进制程序及其执行流程两个主要特点。

ISPRAS-QEMU通过建立分层的事件驱动的插件机制，使得它实现的整个VMI系统结构清晰且易于扩展。然而和DECAF-QEMU一样，每个插件都需要在程序启动后手动开启，而不能直接在命令行参数中进行设置，在某些场景下使用起来不够方便。

另外，和DECAF-QEMU相比，ISPRAS-QEMU不需要依赖操作系统内核结构的信息，而是通过ABI规范创建了一些可以恢复内核级别信息的工具，从而实现各个事件的监测和定位。这种策略使得ISPRAS-QEMU具备了不依赖系统板的特性，但同时也损失了一些功能和便利性，如只能通过页目录地址标识进程，以至于无法通过指定进程名称和进程号的方式来筛选属于特定进程的各类事件。

## 2.5 本章小结

本章介绍了模糊测试领域和利用QEMU构建VMI系统的相关工作。

其中，模糊测试方面，介绍了AFL、AFLNet、TriforceAFL三者的工作流程、各自特点及注意事项。如表2-1所示，AFLNet在AFL的基础上添加了网络通信支持，TriforceAFL将AFL的模糊测试能力扩展到了固件领域，但仅限于固件系统调用的测试，无法应用于固件应用程序。

利用QEMU构建VMI系统方面，介绍了DECAF-QEMU和ISPRAS-QEMU两项通过构建QEMU插件系统实现VMI功能的工作及其各自特点和注意事项。如表2-2所示，DECAF-QEMU和ISPRAS-QEMU所使用的QEMU版本都很低，导致较新的固件系统如OpenBmc无法成功启动。

同时，DECAF-QEMU依赖客户机操作系统内核的数据结构信息，来探测客户机内各个进程的信息，从而可以识别目标进程，但不同的操作系统版本，内核数据结构信息会发生变化，对OS版本的依赖导致DECAF-QEMU不够通用。而ISPRAS-QEMU只对检测到的进程的页目录地址进行记录，不依赖操作系统版本，但也不支持目标进程的识别。

另外，DECAF-QEMU和ISPRAS-QEMU诞生时，QEMU官方尚未提供插件机制，因此二者使用的插件系统都是项目内部自行实现的，通用性和版本兼容性都存在不足。

表 2-1 模糊测试工具对比

名称	网络通信	固件系统调用测试	固件网络应用测试	固件非网络应用测试
AFL	×	×	×	×
AFLNet	√	×	×	×
TriforceAFL	×	√	×	×
AFLNetSpy	√	√	√	×

表 2-2 QEMU-VMI 系统对比

名称	QEMU 版本	支持 OpenBmc	不依赖 OS 版本	可识别目标进程	基于官方插件系统
DECAF-QEMU	1.0	×	×	√	×
ISPRAS-QEMU	2.8.50	×	√	×	×
QEMU-SPY	8.2.91	√	√	√	√

## 第3章 面向固件网络应用程序的系统级灰盒模糊测试

本文提出的系统级灰盒模糊测试方法分为两个部分：①获取客户机内目标进程代码执行信息的方法；②利用网络请求对目标进程的状态进行探测和控制的方法。本章在3.1节对上述方法的两个部分分别进行介绍和示例说明，然后在3.2节介绍本文基于系统级灰盒模糊测试方法实现的AFLNetSpy原型系统，并对AFLNetSpy系统架构、QEMU-SPY子系统设计和系统调用回调实现三个部分分别进行阐述。

### 3.1 系统级灰盒模糊测试方法

#### 3.1.1 代码执行信息的获取方法

要想进行固件内应用程序的灰盒模糊测试，首先需要能够获取目标进程的代码执行信息。

Linux系统下，每启动一个进程，会为其分配一个页目录，页目录地址和进程是一一对应的对应关系，因此可以通过页目录地址标识进程，并区分不同进程的指令信息。QEMU系统中，页目录地址会被存放在VCPU的特定页目录地址寄存器，以Arm系统为例，页目录地址存放在`cp15.ttbr0_el[3]`寄存器中，能够通过对应的CPUArchState类型的结构体方便地访问该值。也就是说，我们使用页目录地址就能够标识不同的进程，省去了探索操作系统内核信息的麻烦和系统时间开销。

利用以上原理，通过QEMU-SPY的插件系统，本文首先实现两项基本的监测功能：

(1) 一是对客户机系统内的进程进行监测。如图3-1所示，通过实时监测`execve`系统调用的参数，能够获取新启动进程的名称，从而实现进程监测。需要注意，其中`ctx`为当前执行指令的进程的页目录地址，以第158行启动`hello-crow`进程为例，`0x8203c008`并非`hello-crow`进程的页目录地址，而是其父进程(这里是`systemd`)的页目录地址。



```
149  ctx: 8203c008  execve  /usr/bin/bmcweb
150  ctx: 81a1c008  execve  /proc/self/fd/9
151  ctx: 80ebc008  execve  /sbin/fw_setenv
152  ctx: 81a1c008  execve  /proc/self/fd/9
153  ctx: 81e14008  execve  /usr/bin/convert-pam-configs.sh
154  ctx: 81a1c008  execve  /proc/self/fd/9
155  ctx: 80fd0008  execve  /usr/bin/sh
156  ctx: 80fd0008  execve  /usr/bin/grep
157  ctx: 81a1c008  execve  /proc/self/fd/9
158  ctx: 8203c008  execve  /usr/bin/hello-crow
159  ctx: 80ebc008  execve  /usr/bin/hello-crows
160  ctx: 81a1c008  execve  /proc/self/fd/9
161  ctx: 81a1c008  execve  /proc/self/fd/9
162  ctx: 80fe4008  execve  /usr/bin/hello-server
```

图 3-1 进程启动监测

(2) 二是对客户机内执行的指令进行监测。如图3-2所示，通过注册翻译块执行回调函数，每一个QEMU翻译块执行时，都可以进行记录。其中的ctx即为当前执行指令的进程对应的页目录地址，pc的值为当前翻译块中的第一条客户机指令对应的客户机虚拟地址。

```
78357  ctx: 8232c008  tb_exec pc: 807a3bc8
78358  ctx: 8232c008  tb_exec pc: 807a383c
78359  ctx: 8232c008  tb_exec pc: 8010e9b8
78360  ctx: 8232c008  tb_exec pc: 8010e9e0
78361  ctx: 80004008  tb_exec pc: 8010e9e8
78362  ctx: 80004008  tb_exec pc: 8010e9f0
78363  ctx: 80004008  tb_exec pc: 8010eab8
78364  ctx: 80004008  tb_exec pc: 8010ead0
78365  ctx: 80004008  tb_exec pc: 8010ead8
78366  ctx: 80004008  tb_exec pc: 8010ea7c
78367  ctx: 80004008  tb_exec pc: 8010ed84
78368  ctx: 82290008  tb_exec pc: 8010ed9c
78369  ctx: 82290008  tb_exec pc: 8010eda0
78370  ctx: 82290008  tb_exec pc: 807a3848
78371  ctx: 82290008  tb_exec pc: 80101024
```

图 3-2 指令执行监测

从图3-2中还可以看出，QEMU客户机系统内多个进程同时运行，要想获取目标程序的代码执行信息，需要先获取目标进程对应的页目录地址，然后根据该地址从所有翻译块执行信息中筛选出属于目标进程的执行信息。且无法在目标进程启动时通过监测execve系统调用获取目标进程的页目录地址，那样只能获得其父进程的页目录地址。要想获取目标进程的页目录地址，需要另外的策略。

为此，本文充分考虑到目标进程的网络应用特性，使用请求探测和监控Accept系统调用的方式实现简单高效地获取目标进程页目录地址的功能。

首先，要设法确认客户机系统和目标进程启动完成，通过监测execve等用于创建新进程的系统调用及其参数，即可根据进程名称判断目标进程是否启动。同时，可以通过进程名称选定一个大概率最后启动的进程，当该进程启动时，认为客户机启动完成。具体实现中，如图3-3所示，当设置phosphor-host-state-manager进程为系统启动完成的标志时，一旦监测到execve系统调用创建了该进程，即认为系统启动完成，同时也认为目标进程也已启动完成。

```
386 ctx: 86014008 execve /usr/bin/netipmid
387 ctx: 82330008 execve /usr/bin/phosphor-host-state-manager
388 Attached to shared memory trace_bits: 0x7f4502c1e000, spy_signal: 0x7f4504c44000
389 System Started.
390 Wrote RDY! to STATE_WRITE_FD
```

图 3-3 确认客户机系统启动完成

在确认客户机系统和目标进程启动完成后，向目标进程发送网络请求，随后调用Accept系统调用接收网络请求的进程即可确定为目标进程，对应的页目录地址即为所需的目标进程页目录地址。具体实现中，如图3-4所示，我们首先会向SpyAgent进程发送网络请求，然后监测Accept系统调用获取SpyAgent进程的页目录地址，然后使用同样的方式即可获取目标测试进程的页目录地址。其中SpyAgent进程用于控制目标测试进程的状态，将在3.1.2节中进行介绍。

```
388 Attached to shared memory trace_bits: 0x7f4502c1e000, spy_signal: 0x7f4504c44000
389 System Started.
390 Wrote RDY! to STATE_WRITE_FD
391 ctx: 82134008 accept sockfd: 6, sock_addr: 00000000, addr_len: 0
392 Agent Context set to 82134008
393 Wrote agent_ctx to STATE_WRITE_FD
394 ctx: 80f7c008 accept sockfd: 6, sock_addr: 00000000, addr_len: 0
395 Target Context set to 80f7c008
396 Wrote target_ctx to STATE_WRITE_FD
```

图 3-4 获取目标进程页目录地址

### 3.1.2 目标进程的状态探测和控制方法

要想进行固件内应用程序的灰盒模糊测试，还要实现目标进程的状态探测和控制。为此，需要用户根据实际测试环境和测试目标编写三个网络请求脚本，示例如下：

(1) TEST\_AGENT请求。如图3-5所示，其中14817为SpyAgent进程端口号。SpyAgent进程是需要内置于客户机系统的“间谍代理”进程，它的作用是接收外部网络请求，并执行请求参数中携带的Shell命令。因此，我们可以通过向SpyAgent进程发送命令，实现对目标进程状态的控制。

```
spy-test-http > scripts > $ test_agent.sh
1 curl 127.0.0.1:14817 --max-time 2 > /dev/null 2>&1
2 # test_agent.sh
3 # (1) Inform qemu-aflspy to set the context of the SpyAgent process based on this request
4 # (2) The port number 4817 is taken from the homophone of spy-agent,
5 # and the presence of "1" at the beginning is due to port forwarding of QEMU
```

图 3-5 TEST\_AGENT 请求示例

(2) TEST\_ALIVE请求。如图3-6所示，其中18080为目标进程的网络端口。

```
spy-test-http > scripts > $ test_alive.sh
1 curl 127.0.0.1:18080 --max-time 2 > /dev/null 2>&1
2 # test_alive.sh
3 # (1) Inform qemu-aflspy to set the context of the target process based on this request
4 # (2) Check the alive status of the target process
```

图 3-6 TEST\_ALIVE 请求示例

(3) RESTART\_TARGET请求。如图3-7所示，CMD命令为需要在客户及系统内执行的Shell命令，SpyAgent进程将提取该命令，并使用system函数进行执行。该命令的作用是重启目标测试进程hello-crow。

```
spy-test-http > scripts > $ restart_target.sh
1 TARGET_PROC=hello-crow
2 TARGET_SERVICE=hello-crow.service
3 CMD="pgrep $TARGET_PROC && (pgrep $TARGET_PROC | xargs kill -9); \
4     systemctl reset-failed && \
5     systemctl restart $TARGET_SERVICE"
6
7 curl 127.0.0.1:14817/execute \
8     --max-time 2 \
9     -d '{"cmd": "$CMD"}' > /dev/null 2>&1
10
11 # restart_target.sh
12 # (1) Use pgrep to check if the process is running before killing it
13 # (2) Use systemctl reset-failed to reset the failed status of the service[IMPORTANT]
14 # (3) Use systemctl start or restart to restart the service, no obvious difference between them
15 # (4) Put the comments below in the file to avoid incomplete reading of the commands due to excessive comments
```

图 3-7 RESTART\_TARGET 请求示例

利用上述三个网络请求脚本，可以实现以下进行系统级灰盒模糊测试所必需的功能（具体实现中，一般将目标进程的退出和重新启动组合为一条shell命令，如图3-7所示）：

(1) 目标进程的状态探测。由于目标进程是网络服务器，除了检查它返回给afl-fuzz的响应中的状态码，还应在每次测试结束后检查目标进程的存活状态，以保证当目标进程崩溃时及时得到通知。本文采取的策略是通过发送由用户编写的针对目标进程的TEST\_ALIVE网络请求，根据返回情况判断目标进程是否仍处于运行状态。

(2) 目标进程的快速退出。经过测试，当外部请求触发客户机内目标进程的一个段错误时，会触发异常信号，目标进程会等待客户机系统处理该信号，并关闭目标进程。然而，由于QEMU的实现机制，异常信号不会立刻得到处理，目标进程也不会迅速被关闭，从触发段错误到目标进程成功退出，需要4分钟左右。这是难以接受的。因此，本文提出一种主动探测式的方法，通过TEST\_ALIVE请求探测目标进程状态，若超出规定的最大时间仍未成功返回，则认为目标进程出现了CRASH，并向SpyAgent程序发送强制杀掉目标进程的命令，从而实现目标进程的快速退出。

(3) 目标进程的重新启动，为保证模糊测试的持续运行，在触发目标进程的崩溃并退出目标进程后，需要重新启动新的目标进程来继续模糊测试。这可以通过向客户及系统内的代理程序Spy-Agent发送重启目标进程的shell命令来实现，如systemctl

restart \$TARGET\_SERVICE等。另外，还需要注意的是，现代操作系统及服务管理程序如systemd等，可能存在避免服务器频繁异常重启的保护机制，因此可能需要在强制退出目标进程后，重启目标进程前，重置该网络服务的错误状态信息，如使用systemd的systemctl reset-failed命令等。

### 3.2 系统级灰盒模糊测试系统

本文实现的系统级灰盒模糊测试系统AFLNetSpy，相较于现有模糊测试框架的改进点如表2-1所示，AFLNet和TriforceAFL等工具对于固件内应用程序的灰盒模糊测试束手无策，而AFLNetSpy在继承AFLNet网络通信功能的基础上，借鉴TriforceAFL使用QEMU系统模式进行固件模拟的思路，结合利用QEMU官方插件系统开发出的QEMU-SPY，能够实现固件内网络应用的灰盒模糊测试。

AFLNetSpy中的QEMU-SPY子系统相较于DECAF-QEMU和ISPRAS-QEMU实现的VMI内省系统的主要改进点如表2-2所示，包括：

(1) 基于最新稳定版的QEMU进行开发，保证系统能够支持较新的固件镜像如OpenBMC等镜像的启动。

(2) 通过巧妙地利用网络应用程序的系统调用序列的特征，能够实现目标网络进程的识别。由于不需要依赖操作系统版本，提高了通用性和兼容性，也不用探测内核结构，因此开销要远小于DECAF-QEMU的策略。不过需要注意，目前并不支持非网络应用进程的识别。

(3) 基于QEMU官方提供的插件机制进行开发。QEMU官方从4.2版本起开始提供一套插件机制，目前已经比较稳定，但功能较少，需要进行二次开发。使用QEMU官方的插件机制，而不是像DECAF-QEMU或ISPRAS-QEMU一样自行实现或使用第三方提供的插件系统，可以提高系统稳定性，并使系统保持对QEMU版本的向后兼容性。

下面首先在3.2.1节介绍AFLNetSpy系统的整体架构和工作流程，然后在3.2.2节讲解QEMU-SPY子系统的详细设计，最后在3.2.3节详细讲解系统调用回调的实现逻辑。

### 3.2.1 AFLNetSpy 架构

本文所实现的AFLNetSpy系统的整体架构如图3-8所示。其中trace\_enabled和next\_step与trace\_bits一样，是afl-fuzz和QEMU-SPY通过共享内存方式共享的两个变量，用于控制和同步trace\_bits的记录。QEMU-SPY为经过扩展的QEMU和AFL-SPY插件组成的具有VMI功能的系统，其具体实现可参见本文的代码仓库，且将在3.3节进行详细讲解。

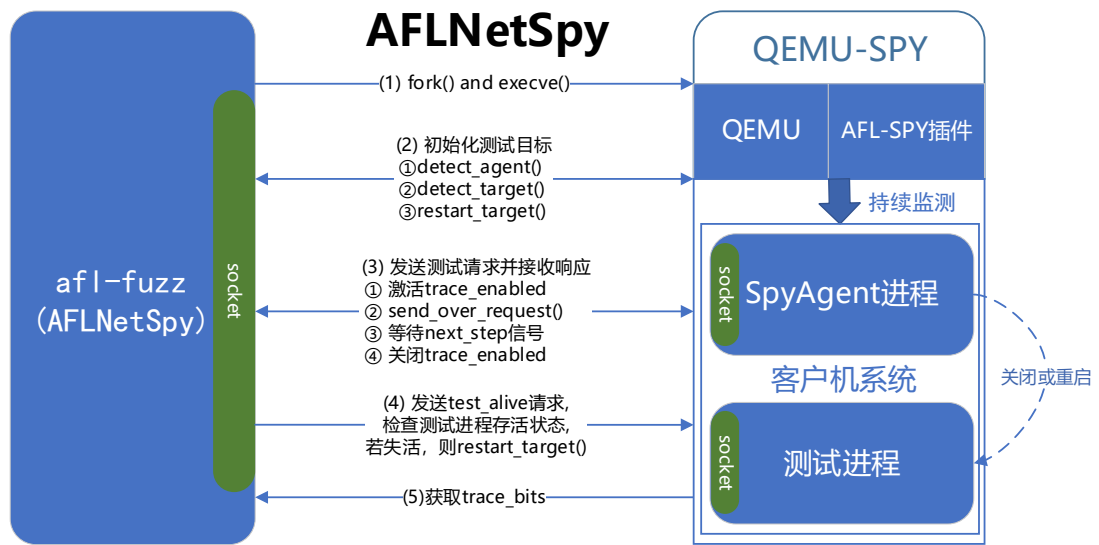


图 3-8 AFLNetSpy 系统架构图

如图3-8所示，AFLNetSpy的整体工作流程如下：

(1) 首先afl-fuzz进程创建一个子进程用来运行QEMU，QEMU启动客户机系统前会加载通过命令行参数指定的AFL-SPY插件libaflspy.so，并注册所需的各类回调函数。启动后的客户及系统内，包含待测试的目标网络进程，和提前内置或后续传输进客户机系统的SpyAgent进程，该进程能够接收外部的网络请求并执行请参数中的shell命令，可以实现快速关闭和重启目标进程。

(2) 初始化测试目标，包含三个步骤：①detect\_agent，向SpyAgent进程发送用户编写的TEST\_AGENT请求，AFL-SPY插件在监测到该请求时，会将SpyAgent进程的页目录地址设为AgentCtx，并将该值通过管道返回给afl-fuzz进程；②detect\_target，向目标测试进程发送用户编写的TEST\_ALIVE请求，AFL-SPY插件在监测到该请求时，会将目标进程的页目录地址设为TargetCtx，并将该值通过管道返回给afl-fuzz进程；



③restart\_target，由于重启后的目标进程的页目录地址会发生改变，需要重新确定。因此该函数分为两步，一是向SpyAgent进程发送用户编写的RESTART\_TARGET请求，SpyAgent接收到该请求后，会提取请求参数中的shell命令并执行，从而实现目标进程的重启，同时AFL-SPY插件在监测到该请求时会将TargetCtx的值设为0，表明原TargetCtx已失效，需要重新定位。二是执行②中提到的detect\_target函数，从此实现TargetCtx的更新。

(3) 发送测试请求并接收响应，包含四个步骤：①激活trace\_enabled，通知AFL-SPY插件，对接下来的目标进程的翻译块进行记录；②send\_over\_request是AFLNet提供的发送测试请求并接受响应的函数，本文对其进行了扩展，以支持HTTPS通信；③等待next\_step信号，AFL-SPY插件在监测到目标进程执行特定系统调用如sendto、sendmsg时，将next\_step的值设为1，表示此次测试请求已经处理完毕可以开始准备下一次测试了。afl-fuzz进程会轮询next\_step的值，直到该值为1或达到最大轮询次数，则进入下一步；④关闭trace\_enable，通知AFL-SPY停止记录翻译块的执行信息，保证得到的trace\_bits的数据足够干净。

(4) 发送TEST\_ALIVE请求，检查处理上次测试请求后目标测试进程的存活状态，若未正常返回，则认为上次的测试请求触发了测试进程的CRASH，然后使用restart\_target函数重启一个新的测试进程，保证测试能够继续进行。若正常返回，说明上次的测试请求没能触发测试进程的CRASH，出于性能考虑，可以不重启测试进程。

(5) afl-fuzz获取AFL-SPY插件更新过的trace\_bits，进行后续分析。然后生成新的测试请求，重复(3)-(5)步，直到用户手动停止测试。

### 3.2.2 QEMU-SPY 设计

本节介绍QEMU-SPY子系统的详细设计。QEMU-SPY子系统由QEMU和AFL-SPY插件组成。

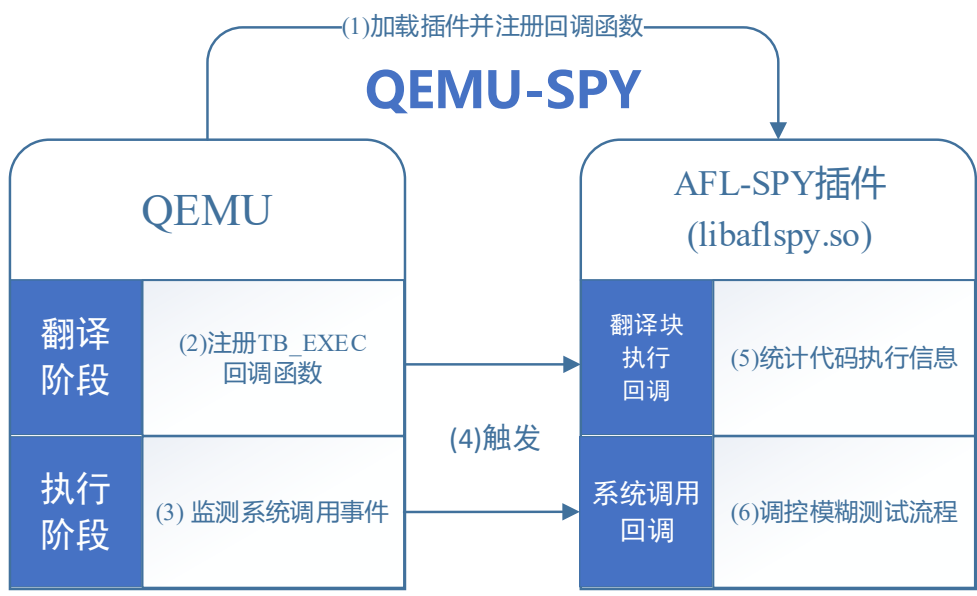


图 3-9 QEMU-SPY 子系统结构图

QEMU-SPY的架构的如图3-9所示，其工作机制如下：

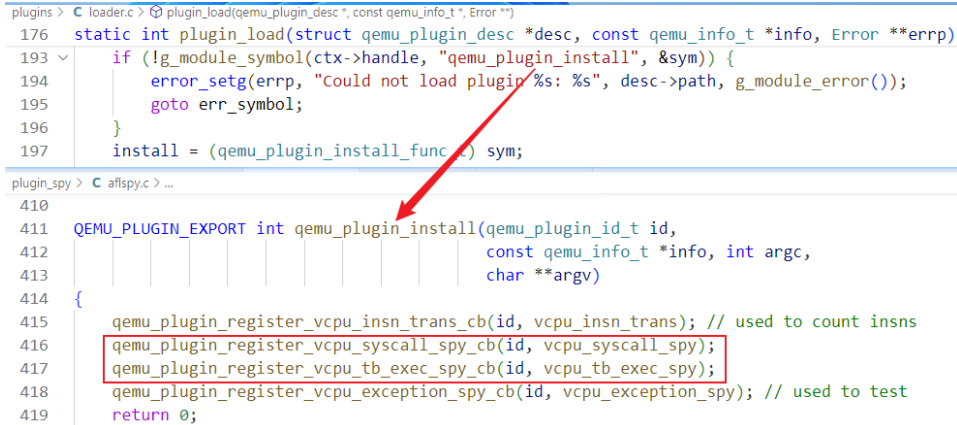
(1) 加载插件并注册回调函数。首先如图3-10所示，QEMU进程在启动后会根据命令行参数加载指定的AFL-SPY插件，即libaflspy.so。然后如图3-11所示，通过查询插件中的函数符号，获取qemu\_plugin\_install函数地址并执行，从而实现各类事件的回调函数注册。

```
KERN=/home/czx/openbmc-workspace/openbmc/build/romulus/tmp/deploy/images/romulus
QEMUSPY=/home/czx/qemu-workspace/qemu-spy
AFLSPY=$QEMUSPY/plugin_spy

$QEMUSPY/build/qemu-system-arm \
  -m 256 -machine romulus-bmc \
  -drive file=$KERN/obmc-phosphor-image-romulus.static.mtd,if=mtd,format=raw\
  -net nic \
  -net user,hostfwd=:127.0.0.1:18080-:8080,hostfwd=:127.0.0.1:14817-:4817,hostname=qemu \
  -d plugin,nochain \
  -plugin $AFLSPY/build/libaflspy.so \
  -D qemu_log.txt \
  -nographic \
```

图 3-10 QEMU 系统模式使用插件时的命令行参数





```


plugins > C loader.c > plugin_load(qemu_plugin_desc *, const qemu_info_t *, Error **)
176 static int plugin_load(struct qemu_plugin_desc *desc, const qemu_info_t *info, Error **errp)
193 {
194     if (!g_module_symbol(ctx->handle, "qemu_plugin_install", &sym)) {
195         error_setg(errp, "Could not load plugin '%s': %s", desc->path, g_module_error());
196         goto err_symbol;
197     }
198     install = (qemu_plugin_install_func_t) sym;
199 }

plugin_spy > C aflspy.c > ...
410
411 QEMU_PLUGIN_EXPORT int qemu_plugin_install(qemu_plugin_id_t id,
412                                           const qemu_info_t *info, int argc,
413                                           char **argv)
414 {
415     qemu_plugin_register_vcpu_insn_trans_cb(id, vcpu_insn_trans); // used to count insns
416     qemu_plugin_register_vcpu_syscall_spy_cb(id, vcpu_syscall_spy);
417     qemu_plugin_register_vcpu_tb_exec_spy_cb(id, vcpu_tb_exec_spy);
418     qemu_plugin_register_vcpu_exception_spy_cb(id, vcpu_exception_spy); // used to test
419     return 0;

```

图 3-11 插件加载与回调函数注册

(2) 注册TB\_EXEC回调函数。当客户机架构和宿主机架构不同时，如客户机为arm架构，宿主机为x64架构，QEMU需要将客户机二进制代码翻译为宿主机二进制代码才能够执行。翻译阶段包含两个步骤：①将客户机二进制指令块翻译为TCG中间指令块；②将TCG中间指令块编译为宿主机二进制指令块。通过在每个TCG中间指令块中插入特定的代码，能够实现类似回调函数的功能，且QEMU提供的HELPER机制使得这项操作变得易于实现。如图3-12所示，本文通过在每个翻译块，具体来说是TCG中间指令块中，使用QEMU提供的HELPER机制插入HELPER(tb\_exec\_spy)函数，使得每个翻译块执行前都会调用AFL-SPY插件中编写的翻译块执行回调函数，从而实现代码执行信息的统计。



```

accel > tcg > C plugin-gen.c > plugin_gen_insn_trans_spy(CPUState *, const DisasContextBase *)
966
967 void plugin_gen_tb_trans_spy(CPUState *cpu, const DisasContextBase *db)
968 {
969     gen_helper_tb_exec_spy(tcg_env);
970 }
971
972 void HELPER(tb_exec_spy)(CPUArchState *env)
973 {
974     CPUState *cpu = env_cpu(env);
975     g_autofree TBInfo *data = g_new0(TBInfo, 1);
976     data->ctx = env->cp15.ttbr0_el[3];
977     data->pc = env->regs[15];
978
979     qemu_plugin_tb_exec_spy_cb(cpu, env, data);
980 }

plugin_spy > C aflspy.c > vcpu_tb_exec_spy(qemu_plugin_id_t, CPUState *, CPUArchState *, void *)
25 void vcpu_tb_exec_spy(qemu_plugin_id_t id, CPUState *cpu, CPUArchState *env)
32 {
33     if (is_trace_enabled()) {
34         if (info->ctx == target_ctx && info->pc < 0x01000000) {
35             count++;
36             afl_maybe_log(info->pc); // used to record code execution info

```

图 3-12 TB\_EXEC 回调函数实现

(3) 监测系统调用事件。本文还对 QEMU 进行了另一项扩展，使其支持系统模式下客户机调用的监测。实现思路如图 3-13 所示，首先对每一条客户机二进制指令进行检查，当监测到系统调用指令如 arm 的 svc 指令(0xef000000)时，根据当前 VCPU 的寄存器参数判断系统调用类型，若属于需要监测的系统调用类型，则收集本次系统调用的参数，并触发 AFL-SPY 插件中编写的系统调用回调函数。具体代码实现如图 3-14 所示，其中 SyscallInfo 结构体的设计如图 3-15 所示，通过设计良好的数据结构，我们的系统调用回调函数极为良好的可扩展性，可以快速地添加对其他系统调用的监测和对应的回调操作。

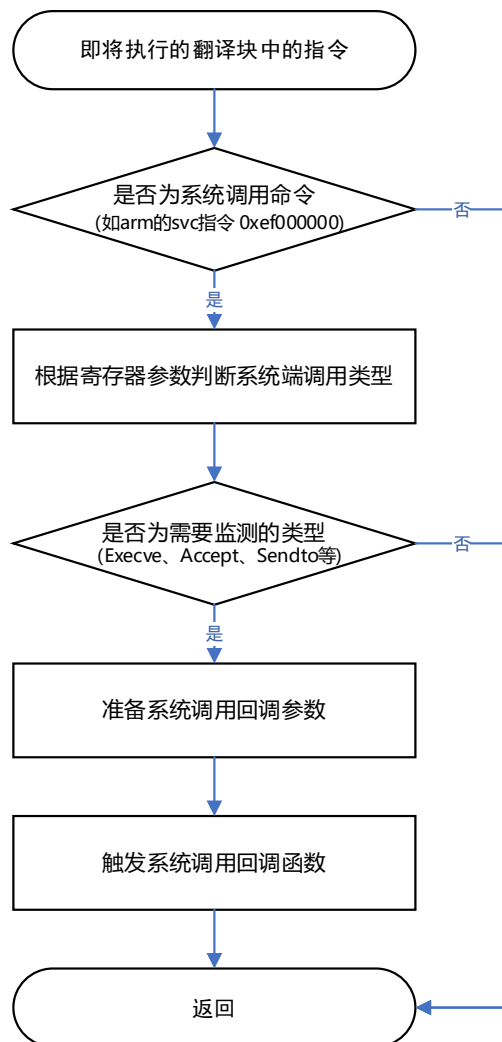


图 3-13 系统调用监测流程图

```

accel > tcg > C plugin-gen.c > plugin_gen_tlb_set_spy(CPUState *, vaddr, hwaddr, int, int)
982 void plugin_gen_insn_trans_spy(CPUState *cpu, const DisasContextBase *db)
983 {
984     CPUArchState *env = cpu_env(cpu);
985     uint32_t insn = cpu_ldl_code(env, db->pc_next);
986     qemu_plugin_insn_trans_cb(cpu, env, insn);
987     if (insn == 0xef000000) {
988         gen_helper_syscall_spy(tcg_env);
989     }
990 }
991
992 QEMU_DISABLE_CFI
993 void HELPER(syscall_spy)(CPUArchState *env)
994 {
995     CPUState *cpu = env_cpu(env);
996     g_autofree SyscallInfo *data = g_new0(SyscallInfo, 1);
997     data->num = env->regs[7];
998     data->ctx = env->cp15.ttbr0_el[3];
999 > switch (env->regs[7]) { ...
1107     qemu_plugin_syscall_spy_cb(cpu, env, data);
1108 }

plugin_spy > C aflspy.c > vcpu_syscall_spy(qemu_plugin_id_t, CPUState *, CPUArchState *, void *)
87 void vcpu_syscall_spy(qemu_plugin_id_t id, CPUState *cpu, CPUArchState *en
110
111 > switch (info->num) { ...
404 }

```

图 3-14 SYSCALL\_SPY 回调函数实现

```

145 typedef struct {
146     uint32_t num;
147     uint32_t ctx;
148     union {
149         ExitParams      *exit_params;
150         ReadParams      *read_params;
151         WriteParams     *write_params;
152         OpenParams      *open_params;
153         CloseParams     *close_params;
154         ExecveParams     *execve_params;
155         SendParams      *send_params;
156         SendtoParams    *sendto_params;
157         SendmsgParams    *sendmsg_params;
158         RecvParams      *recv_params;
159         RecvfromParams   *recvfrom_params;
160         SocketParams     *socket_params;
161         ListenParams     *listen_params;
162         BindParams       *bind_params;
163         AcceptParams     *accept_params;
164     } params;
165 } SyscallInfo;

```

图 3-15 SyscallInfo 结构体设计

(4) 当监测到特定事件时，触发AFL-SPY中的回调函数。这一功能，由QEMU官方提供的插件机制来保障实现。如图3-16所示，本文保持该机制的框架不变，额外添加了所需的TB\_EXEC\_SPY和SYSCALL\_SPY等事件及配套的中间函数。

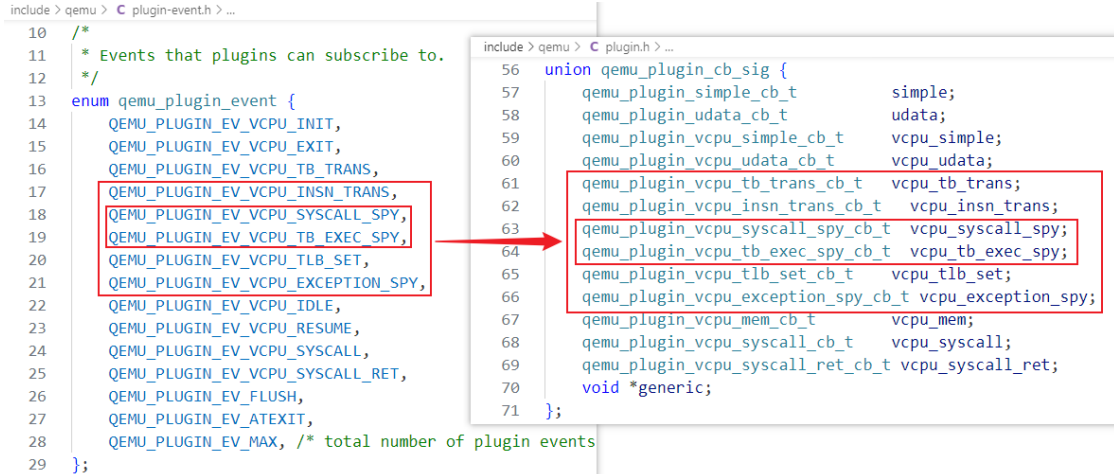


图 3-16 QEMU 官方插件机制扩展

(5) 翻译块执行回调。AFL-SPY插件中编写的翻译块执行回调函数(tb\_exec\_spy)，用于实现代码执行信息的统计。实现思路如图3-17所示，其中TBInfo为作为参数传给回调函数的结构体，包含当前页目录地址ctx，和当前翻译块中对应的客户机二进制代码块中第一条指令的地址pc。首先判断trace\_enabled是否处于激活状态，若激活且当前进程是目标进程，则进一步检查pc的值是否处于监测范围，若处于则进行记录，并更新trace\_bits。具体代码实现可参见图3-12。

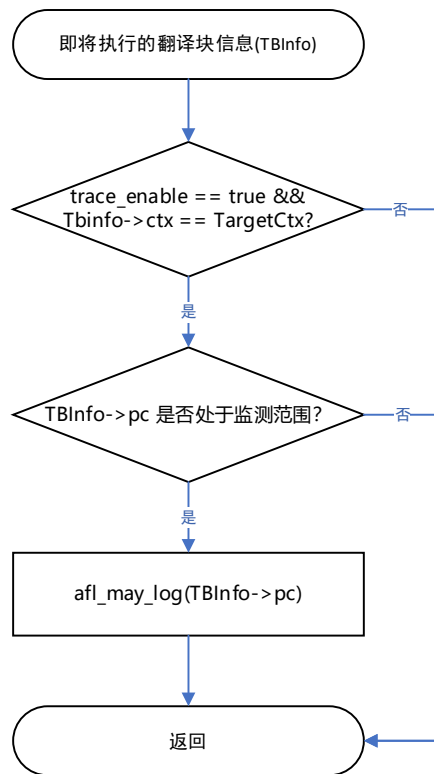


图3-17 “翻译块执行事件”回调函数流程图

(6) 系统调用回调。AFL-SPY插件中编写的翻译块执行回调函数(syscall\_spy)，用于调控模糊测试流程并和afl-fuzz进程保持同步。该函数的代码框架可参见图3-14，通过switch判断系统调用号，执行对应的操作。各个系统调用对应的具体回调操作逻辑较为复杂，在下一节中进行详细讲解。

### 3.2.3 系统调用回调实现

AFL-SPY插件中实现的系统调用回调函数部分是调控模糊测试流程，并和afl-fuzz进程保持同步的关键逻辑所在。本文实现的AFLNetSpy原型系统需要监测三类系统调用：①用于创建进程的Execve等系统调用；②用于接收网络请求的Accept系统调用；③用于发送网络请求的Send/Sendto/Sendmsg等系统调用。下面分别进行介绍：

(1) 系统调用回调函数对Execve系统调用的处理逻辑如图3-18所示，当监测到Execve系统调用时，首先会提取需要创建的进程的进程名，如果进程名是用户指定的能够标识系统启动完成的进程名，则将SystemStarted设为True，表示系统启动完成，同时认为目标进程和SpyAgent进程也已经启动完成。具体代码实现如图3-19所示。

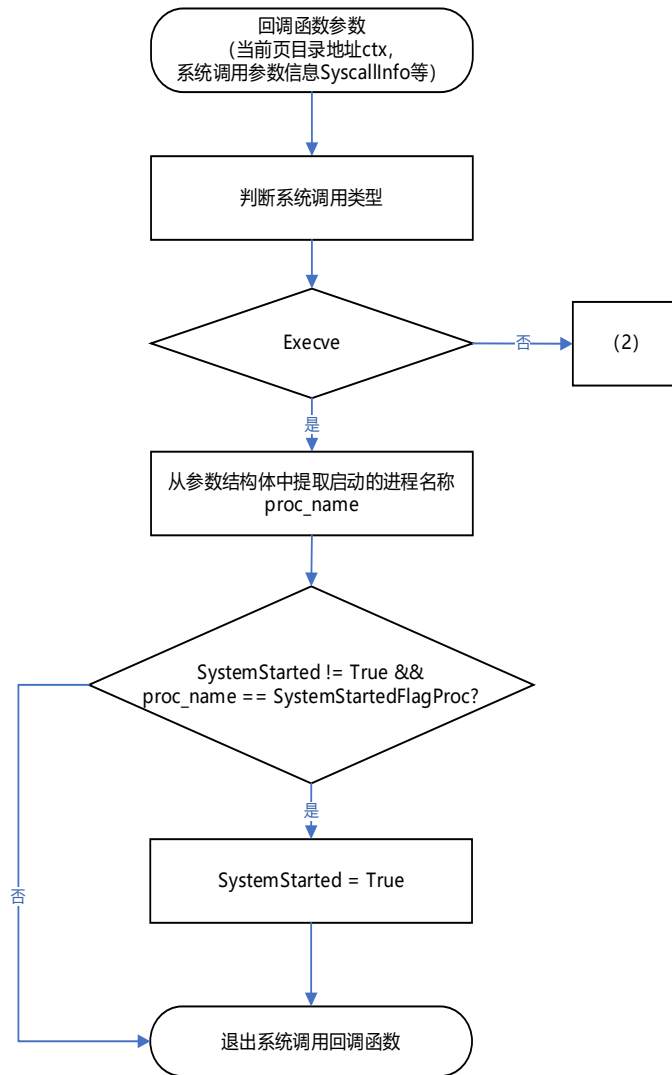


图 3-18 “Execve 系统调用”回调函数流程图

```

plugin_spy > C aflspy.c > vcpu_syscall_spy(qemu_plugin_id_t CPUState *, CPUArchState *, void *)
87 void vcpu_syscall_spy(qemu_plugin_id_t id, CPUState *cpu, CPUArchState *env, void *data)
178     case EXECVE: {
179         // Use strstr instead of g_strncmp0 to wildcard match
180         if (!system_started &&
181             strstr(info->params.execve_params->filename, SYSTEM_STARTED_INDICATOR_PROCESS) != NULL)
182         {
183             system_started = true;
184             afl_setup();
185             LOG_STATEMENT("System Started.\n");
186             if (write(STATE_WRITE_FD, "RDY!", 4) != 4) {
187                 LOG_ERROR("Failed to write RDY! to STATE_WRITE_FD\n");
188             } else {
189                 LOG_STATEMENT("Wrote RDY! to STATE_WRITE_FD\n");
190             }
191         }
  
```

图 3-19 “Execve 系统调用”回调函数代码

(2) 系统调用回调函数对Accept系统调用的处理逻辑如图3-20所示，当监测到Accept系统调用时，根据当前状态有四条执行路径，其中后三条执行路径以系统启动完成为前提：①系统启动完成前，忽略Accept调用，不进行任何操作；②设置AgentCtx：如果AgentCtx仍为初始值0，表明本次处理的请求是TEST\_AGENT请求，因此把AgentCtx的值设为当前执行进程及SpyAgent进程的页目录地址ctx，并通过管道将该值返回给afl-fuzz进程实现同步；③设置TargetCtx：如果AgentCtx的值非0，而TargetCtx的值为0，表明本次处理的请求是TEST\_ALIVE请求，因此把TargetCtx的值设为当前执行进程及目标测试进程的页目录地址ctx，并通过管道将该值返回给afl-fuzz进程实现同步；④重置TargetCtx：如果AgentCtx和TargetCtx的值都非0，表明本次处理的请求是RESTART\_TARGET请求，因此把TargetCtx的值重置为0，表示需要重新通过TEST\_ALIVE请求确定TargetCtx。具体代码实现如图3-21所示。

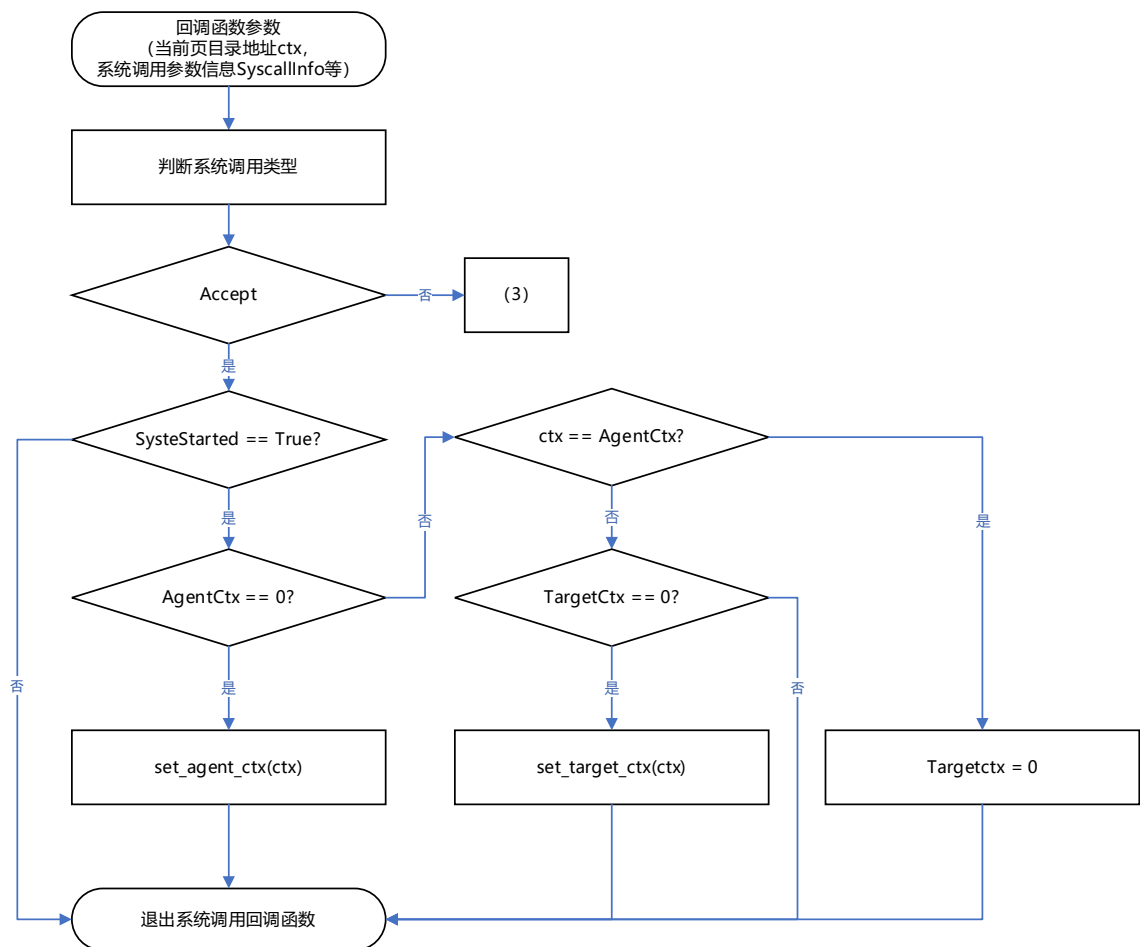


图 3-20 “Accept 系统调用”回调函数流程图

```
plugin_spy > aflspy.c > vcpu_syscall_spy(qemu_plugin_id_t, CPUState *, CPUArchState *, void *)
87 void vcpu_syscall_spy(qemu_plugin_id_t id, CPUState *cpu, CPUArchState *env, void *data)
352     case ACCEPT: {
353         if (system_started) {
354             if (!agent_ctx) {
355                 set_agent_ctx(info->ctx);
356             } else if (info->ctx != agent_ctx && !target_ctx) {
357                 set_target_ctx(info->ctx);
358             } else if (info->ctx == agent_ctx) {
359                 target_ctx = 0;
360             }
361         }
    }
```

图 3-21 “Accept 系统调用”回调函数代码

(3) 系统调用回调函数对Send/SendTo/Sendmsg等测试进程用来返回请求响应的系统调用的处理逻辑如图3-22所示，当监测到Send/SendTo/Sendmsg等系统调用时，首先会检查系统同是已经启动，若系统已经启动，且当前执行进程是目标进程，则进一步检查trace\_enabled和next\_step两个变量的值，若trace\_enabled的值为1且next\_step的值为0，则说明当前目标进程恰好处理完测试请求的时间点，于是把next\_step的值设为0，以通知afl-fuzz进程可以开始准备下一次测试了。具体代码实现(以sendmsg系统调用为例)如图3-23所示。



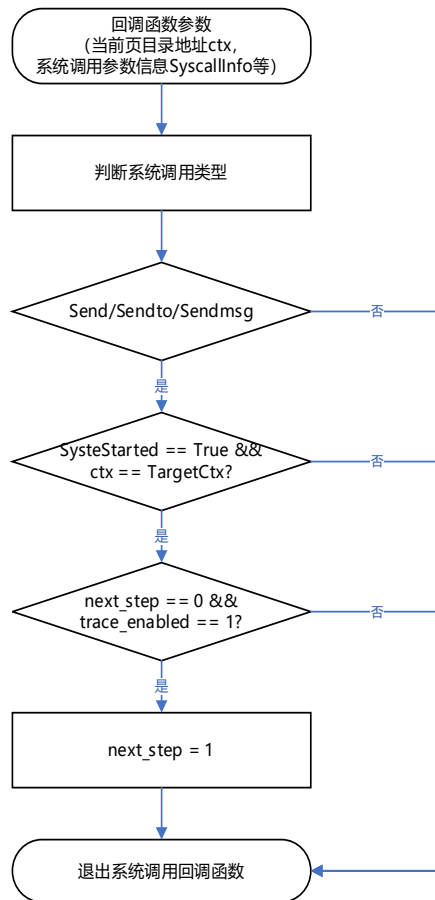


图 3-22 “Send/Sendto/Sendmsg 等系统调用”回调函数流程图

```
plugin_spy > aflspy.c > vcpu_syscall_spy(qemu_plugin_id_t id, CPUState *, CPUArchState *, void *)
87 void vcpu_syscall_spy(qemu_plugin_id_t id, CPUState *cpu, CPUArchState *env, void *data)
274     case SENDMSG: {
275         if (system_started && info->ctx == target_ctx) {
276             if (spy_signal->next_step == 0 && is_trace_enabled()) {
277                 spy_signal->next_step = 1;
278                 LOG_STATEMENT("Target SENDMSG\n");
279             }
280         }
281     }
```

图 3-23 “Sendmsg 系统调用”回调函数代码

### 3.5 本章小结

本章首先对系统级灰盒模糊测试方法的两个组成部分，即获取客户机内目标进程的代码执行信息的方法和对目标进程的状态进行探测和控制的方，进行了介绍和示例说明。然后分析了本文实现的原型系统AFLNetSpy相对于现有框架和工具的优势，并且介绍了AFLNetSpy系统的整体架构和工作流程，对QEMU-SPY子系统的具体设计和系统调用回调函数的实现逻辑进行了详细讲解。

## 第4章 实验结果与分析

本章首先介绍实验前的环境准备工作，然后从有效性、稳定性和性能三个方面对AFLNetSpy原型系统进行实验分析。

### 4.1 环境构建

在开始实验前，需要进行一些准备工作：

(1) 准备AFLNet：由于AFLNet继承自AFL，其使用的QEMU版本为较老的2.10.0。经测试该版本的QEMU及AFL提供的修改，无法在Ubuntu22.04系统上成功进行编译。因此要想使用AFLNet的动态插桩模式，首先需要对AFLNet使用的QEMU进行修改。本文成功将AFL为QEMU-2.10.0版本提供的修改移植到了较新的8.2.0版本，并通过调试验证其运行的正确性。

(2) 准备AFLNetSpy：根据待测固件的架构，设置QEMU-SPY子系统的编译参数，并验证AFL-SPY插件libaflspy.so的可用性和正确性。

(3) 准备固件镜像：选择的固件镜像需要保证能够使用QEMU-SPY子系统成功模拟启动。本文选择使用开源固件系统OpenBmc的比较成熟的romulus版本进行后续实验，由于OpenBmc采用yocto构建系统，因此对磁盘空间较高要求且编译耗时较长。

(4) 准备测试程序：本文使用跨平台网络框架CROW编写了一个简单二进制网络程序hello-crow.cpp。共需编译三个版本：①使用afl-g++编译测试程序，供AFLNet的静态模式使用；②使用arm-linux-gnueabi-g++编译测试程序，供AFLNet的动态插桩模式即QEMU-USER模式使用；③向OpenBmc项目中为测试程序添加一个recipe，供AFLNetSpy测试使用。

(5) 编写请求脚本：AFLNetSpy系统要求用户根据测试目标的具体情况，编写三个请求脚本：①负责执行TEST\_ALIVE请求的test\_alive.sh；②负责执行TEST\_AGENT请求的test\_agent.sh；③负责执行RESTART\_TARGET请求的restart\_target.sh。三个请求脚本的具体内容见代码仓库。

本实验的其它软硬件环境信息，如表4-1所示。

表 4-1 软硬件环境信息

软件环境	操作系统	(WSL)Ubuntu 22.04.3 LTS
	gcc/g++版本	11.4.0
硬件环境	CPU 核心数	24
	CPU 频率	2.10 GHz
	内存	16.0 GB
	存储	1 T

## 4.2 有效性分析

AFLNetSpy系统的有效性验证，包括四个方面：①无CRASH情况下正常运行；②目标进程崩溃时能监测到CRASH；③目标进程崩溃后能够自动重启；④能够发现新的有效的CRASH。下面逐项进行验证：

(1) 无CRASH情况下正常运行。当初始测试用例为如图4-1所示的普通测试请求时，目标进程中处理该请求的代码如图4-2所示，不会出现CRASH。运行如图4-3所示的脚本启动模糊测试，得到如图4-4所示的运行截图和如图4-5所示的输出目录。可以看到模糊测试能够成功启动并正常运行，输出目录中的内容也一切正常。

```
spy-test-http > inputs > 1
1 GET / HTTP/1.1
2 Host: 127.0.0.1:18080
```

图 4-1 普通测试请求

```
CROW_ROUTE(app, "/")([&request_count]() {
    ++request_count; //increment the request count
    return "Hello world";
});
```

图 4-2 普通测试请求处理代码

```
spy-test-http > $ runFuzz.sh
1 KERN=/home/czx/openbmc-workspace/openbmc/build/romulus/tmp/deploy/images/romulus
2 QEMUSPY=/home/czx/qemu-workspace/qemu-spy
3 AFLSPY=$QEMUSPY/plugin_spy
4
5 v ../afl-fuzz -i ./inputs -o ./outputs -P HTTP -N tcp://127.0.0.1/18080 -m 4096M -QQ \
6 -d -q 3 -s 3 -R -W 2 -w 1000 -t 30000 \
7 -- \
8 $QEMUSPY/build/qemu-system-arm \
9 -m 256 -machine romulus-bmc \
10 -drive file=$KERN/obmc-phosphor-image-romulus.static.mtd,if=mtd,format=raw\
11 -net nic \
12 -net user,hostfwd=:127.0.0.1:18080-:8080,hostfwd=:127.0.0.1:14817-:4817,hostname=qemu \
13 -d plugin \
14 -plugin $AFLSPY/build/libaflspy.so \
15 -D qemu_log.txt \
16 -nographic
```

图 4-3 模糊测试启动脚本

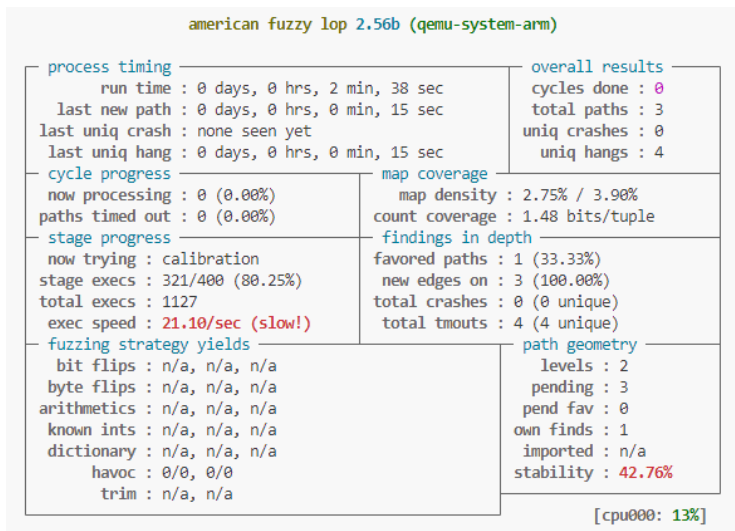


图 4-4 正常情况运行截图

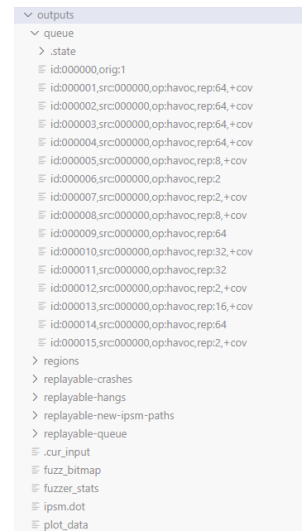


图 4-5 输出目录展示

(2) 目标进程崩溃时成功监测CRASH。将初始测试用例设为如图4-6所示的测试请求，它会触发目标测试进程如图4-7所示的空指针访问，从而导致目标进程崩溃。再次使用图4-3所示的脚本启动模糊测试，得到如图4-8所示的错误信息。根据错误信息可以看到系统成功监测到了此次CRASH。

```
spy-test-http > inputs > 2
1 GET /null HTTP/1.1
2 Host: 127.0.0.1:18080
```

图 4-6 CRASH 测试请求

```
CROW_ROUTE(app, "/null")([ ](){
    int* ptr = NULL;
    return "data: " + std::to_string(*ptr);
});
```

图 4-7 CRASH 测试请求处理代码

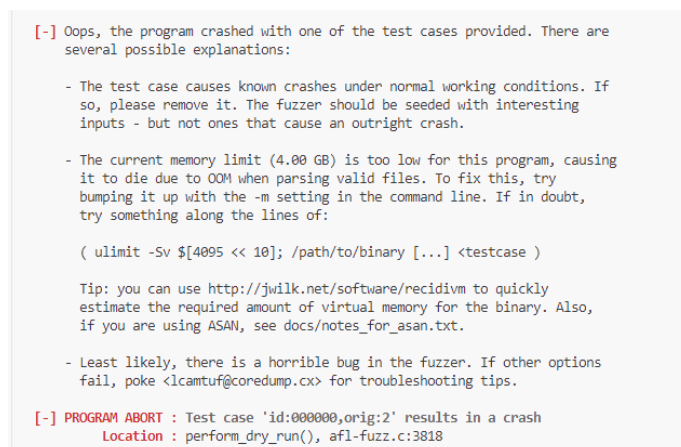


图 4-8 初始测试用例触发 CRASH

(3) 目标进程崩溃后自动重启。为了方便进行此项验证，首先需要对源码进行一些小修改，以允许初始运行时出现CRASH用例。然后重复(2)中的操作，得到如图4-9所示的运行结果和如图4-10所示的可重放CTRASH的目录。根据运行结果可以看出当触发CRASH导致目标进程崩溃后，系统能够自动重启目标进程，继续进行模糊测试。

american fuzzy lop 2.56b (qemu-system-arm)			
process timing		overall results	
run time : 0 days, 0 hrs, 2 min, 11 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 0 min, 0 sec		total paths : 3	
last uniq crash : 0 days, 0 hrs, 0 min, 0 sec		uniq crashes : 4	
last uniq hang : 0 days, 0 hrs, 0 min, 0 sec		uniq hangs : 9	
cycle progress		map coverage	
now processing : 0 (0.00%)		map density : 2.95% / 3.63%	
paths timed out : 0 (0.00%)		count coverage : 1.48 bits/tuple	
stage progress		findings in depth	
now trying : calibration		favored paths : 1 (33.33%)	
stage execs : 6/10 (60.00%)		new edges on : 2 (66.67%)	
total execs : 32		total crashes : 4 (4 unique)	
exec speed : 9.00/sec (zzzz...)		total tmounts : 9 (9 unique)	
fuzzing strategy yields		path geometry	
bit flips : n/a, n/a, n/a		levels : 2	
byte flips : n/a, n/a, n/a		pending : 3	
arithmetics : n/a, n/a, n/a		pend fav : 0	
known ints : n/a, n/a, n/a		own finds : 0	
dictionary : n/a, n/a, n/a		imported : n/a	
havoc : 0/0, 0/0		stability : 53.69%	
trim : n/a, n/a			
[cpu000: 13%]			

图 4-9 目标进程崩溃重启验证图

outputs	
> queue	
> regions	
replayable-crashes	
id:000000,sig:00,src:000000,op:havoc,rep:8	
id:000001,sig:00,src:000000,op:havoc,rep:32	
id:000002,sig:00,src:000000,op:havoc,rep:8	
id:000003,sig:00,src:000000,op:havoc,rep:32	
README.txt	
> replayable-hangs	
> replayable-new-ipsm-paths	
> replayable-queue	
.cur_input	
fuzz_bitmap	
fuzzer_stats	
plot_data	

图 4-10 可重放 CRASH 目录

(4) 发现新的有效的CRASH。重复(1)中的步骤，使用不会触发CRASH的请求作为初始测试用例，启动AFLNetSpy执行模糊测试，运行一段时间后，得到如图4-11所示的结果，可以看到成功发现了CRASH。且利用aflnet-replay工具能够成功复现该CRASH，复现过程和结果如图4-12所示。

american fuzzy lop 2.56b (qemu-system-arm)			
process timing		overall results	
run time : 0 days, 0 hrs, 50 min, 2 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 0 min, 1 sec		total paths : 989	
last uniq crash : 0 days, 0 hrs, 47 min, 47 sec		uniq crashes : 1	
last uniq hang : 0 days, 0 hrs, 0 min, 53 sec		uniq hangs : 210	
cycle progress		map coverage	
now processing : 297 (30.03%)		map density : 2.34% / 6.54%	
paths timed out : 0 (0.00%)		count coverage : 1.54 bits/tuple	
stage progress		findings in depth	
now trying : splice 12		favored paths : 93 (9.40%)	
stage execs : 84/256 (32.81%)		new edges on : 925 (93.53%)	
total execs : 44.7k		total crashes : 1 (1 unique)	
exec speed : 17.67/sec (zzzz...)		total tmouts : 2701 (210 unique)	
fuzzing strategy yields		path geometry	
bit flips : n/a, n/a, n/a		levels : 3	
byte flips : n/a, n/a, n/a		pending : 961	
arithmetics : n/a, n/a, n/a		pend fav : 0	
known ints : n/a, n/a, n/a		own finds : 986	
dictionary : n/a, n/a, n/a		imported : n/a	
havoc : 170/5663, 813/29.2k		stability : 29.87%	
trim : n/a, n/a			
[cpu000: 12%]			

图 4-11 发现新的 CRASH

```
czx@xuan-910:~/afl-workspace/aflnet/spy-test-http$ ./hello-crow-8080 ① 启动测试进程
(2024-05-18 03:09:04) [INFO] Crow/1.1.0 server is running at http://0.0.0.0:8080 using 2 threads
(2024-05-18 03:09:04) [INFO] Call 'app.logLevel(crow::LogLevel::Warning)' to hide Info level logs.
(2024-05-18 03:09:06) [INFO] Request: 127.0.0.1:60818 0x1e8e1d0 HTTP/0.9 GET /data/108080
Segmentation fault

CROW_ROUTE(app, "/data/<int>")([&data](int n){
|   return "data: " + std::to_string(n) + " = " + std::to_string(data[n]);
});

czx@xuan-910:~/afl-workspace/aflnet/spy-test-http$ aflnet-replay outputs-qemu-system-1/replayable-crashes/id\:000000\,si
g\:00\,src\:000000\,op\:havoc\,rep\:32 HTTP 8080 ② 重放CRASH用例

Size of the current packet is 1262

-----
Responses from server:0-
++++
Responses in details:
-----
czx@xuan-910:~/afl-workspace/aflnet/spy-test-http$
```

图 4-12 aflnet-replay 复现 CRASH 用例

### 4.3 稳定性分析

对于灰盒模糊测试框架来说，系统的稳定性，即相同测试请求所获得的代码执行信息的稳定程度。灰盒模糊测试中，代码执行信息被用于指导测试用例的选择和变异，因此它的稳定性会对灰盒模糊测试的确定性和可复现性产生影响，具有重要意义。AFLNetSpy系统的稳定性，取决于QEMU-SPY子系统得到的目标进程的代码执行信息的稳定性。

本节通过比较和分析AFLNet的静态插桩模式、AFLNet的动态插桩即QEMU-USER模式和AFLNetSpy系统模式三种情况下，针对测试程序发送相同测试请求得到的trace\_bits数据的稳定程度，来分析AFLNetSpy系统的稳定性。

比较两个trace\_bits数组相似度的算法如算法1所示。

---

**算法 1:** trace\_bits 相似度计算

---

输入: trace\_bits\_1 和 trace\_bits\_2

输出: trace\_bits\_1 和 trace\_bits\_2 的相似度 similarity

---

- 1 获取 trace\_bits\_1 的所有非 0 值的下标构成集合 keys1
  - 2 获取 trace\_bits\_2 的所有非 0 值的下标构成集合 keys2
  - 3 取 keys1 和 keys2 的并集,得集合 keys
  - 4 count = 0
  - 5 for key in keys:
  - 6     if (key in keys1) and (key in keys2) and (trace\_bits\_1[key] == trace\_bits\_2[key]):
  - 7         count += 1
  - 8 similarity = count / len(keys)
-

当两个trace\_bits数组的相似度超过指定的相似度阈值时，认为二者属于同一簇（Cluster）。后续实验将通过设定不同的相似度阈值，以“所在簇的成员个数（ClusterMember\_Count）”为指标，分析三种情况下，相同测试请求对应的多个trace\_bits的波动性和稳定性。

(1) AFLNet的静态插桩模式下，100次请求生成的100个trace\_bits(实际为110次请求筛去前10项不稳定数据)的“所在簇的成员个数(ClusterMember\_Count)”的统计结果(三次实验取均值)如表4-2所示：

表 4-2 AFLNet 静态插桩模式下的 ClusterMember\_Count 统计结果

相似度阈值	AFLNet 静态插桩模式: ClusterMember_Count				
	最大值	最小值	平均值	中位数	众数
0.9	100.00	100.00	100.00	100.00	100.00
0.95	100.00	100.00	100.00	100.00	100.00
0.99	100.00	100.00	100.00	100.00	100.00
0.999	99.00	34.00	98.03	99.00	99.00
0.9999	3.00	1.00	1.38	1.00	1.00

(2) AFLNet的动态插桩模式即QEMU-USER模式下，100次请求生成的100个trace\_bits(实际为110次请求筛去前10项不稳定数据)的“所在簇的成员个数(ClusterMember\_Count)”的统计结果(三次实验取均值)如表4-3所示：

表 4-3 AFLNet 动态插桩模式下的 ClusterMember\_Count 统计结果

相似度阈值	AFLNet 动态插桩模式: ClusterMember_Count				
	最大值	最小值	平均值	中位数	众数
0.9	100.00	100.00	100.00	100.00	100.00
0.95	100.00	100.00	100.00	100.00	100.00
0.99	96.00	3.33	93.53	96.67	96.67
0.999	89.67	1.33	80.08	89.33	89.33
0.9999	3.00	1.00	1.26	1.00	1.00

(3) AFLNetSpy系统模式下，100次请求生成的100个trace\_bits(实际为110次请求筛去前10项不稳定数据)，的“所在簇的成员个数(ClusterMember\_Count)”的统计结果(三次实验取均值)如表4-4所示：

表 4-4 AFLNetSpy 系统模式下的 ClusterMember\_Count 统计结果

相似度阈值	AFLNetSpy 系统模式: ClusterMember_Count				
	最大值	最小值	平均值	中位数	众数
0.5	100.00	100.00	100.00	100.00	100.00
0.6	99.00	1.00	98.02	99.00	99.00
0.7	99.00	1.00	97.48	99.00	99.00
0.8	97.33	1.00	81.90	89.33	89.33
0.9	81.33	1.00	65.59	80.67	80.67
0.95	71.67	1.00	50.61	70.00	70.33
0.99	61.33	1.00	38.32	59.83	61.00
0.999	23.33	1.00	10.61	10.00	1.00
0.9999	23.33	1.00	10.60	10.00	1.00

根据以上实验结果,可以看到三种情况下的实验都表明,对于网络应用来说,相同的测试用例,得到的代码执行信息并不一定完全相同,可能存在波动情况。根据AFL白皮书的解释,这可能是由于网络应用的多线程或轮询机制导致的,属于正常现象。

同时,还可以发现AFLNetSpy系统收集到的trace\_bits数据的稳定性要比AFLNet略差一些,原因主要有两点:①AFLNet每次测试会重启测试进程,而AFLNet会持续使用同一个测试进程直到该进程崩溃;②AFLNetSpy会记录测试进程在存活期间执行的所有指令信息,而AFLNetSpy只会记录测试进程接收请求(Accept)到返回响应(Send/Sendto/Sendmsg)这一过程中执行的指令信息。

#### 4.4 性能分析

本节通过设计实验,分别在AFLNet的静态插桩模式、AFLNet的动态插桩即QEMU-USER模式和AFLNetSpy系统模式三种情况下运行300次测试(实际运行310次筛去前10项不稳定数据),收集得到300项速率数据,进而比较三种情况的执行效率。实验结果(三次实验取均值)如表4-5所示:



表 4-5 三种模式执行速率的对比数据

模式	每秒执行次数			
	最大值	最小值	平均值	中位数
AFLNet 静态插桩模式	33.30	23.70	31.76	32.14
AFLNet 动态插桩模式	7.29	6.40	6.94	6.95
AFLNetSpy 系统模式	19.57	14.50	17.05	17.01

根据以上实验结果，可以看出：①AFLNet静态插桩模式的效率要远高于动态插桩模式。略高于AFLNetSpy，这主要是因为QEMU的二进制翻译过程带来了较大的时间消耗；②AFLNetSpy的执行速率约为AFLNet动态插桩模式的2.5倍，主要原因为AFLNet每次测试前都会重启一个新的测试进程，而AFLNetSpy会持续利用同一个测试进程直到该进程崩溃，从而节省时间提高执行效率。

图4-13、4-14、4-15分别为AFLNet静态插桩模式、AFLNet动态插桩模式、AFLNetSpy系统模式三种模式对hello-crow程序进行测试，运行11小时(其中AFLNetSpy模式由于内存问题运行7小时即停止)的截图。三种模式发现的CRASH数量的情况如表4-6所示。

表 4-6 三种模式发现的 CRASH 数量统计

模式	发现的 CRASH 数量(个)	有效 CRASH(个)	独特 CRASH(个)
AFLNet 静态插桩模式	4	4	2
AFLNet 动态插桩模式	5	5	2
AFLNetSpy 系统模式	6	6	2

经过测试，AFLNetSpy系统模式比另外两种模式发现的CRASH多的原因主要为，OpenBmc环境下运行的测试进程的栈空间比另外两种情况下要小，更容易触发栈溢出。如图4-16所示，同样的请求前两种模式下能够正常处理，而第三种模式下则会发生崩溃。这种现象进一步证明了系统级灰盒模糊测试的必要性。

```
american fuzzy lop 2.56b (hello-crow-static-inst-8080)

process timing
  run time : 0 days, 11 hrs, 6 min, 25 sec
  last new path : 0 days, 0 hrs, 0 min, 26 sec
  last uniq crash : 0 days, 0 hrs, 53 min, 41 sec
  last uniq hang : none seen yet
cycle progress
  now processing : 2924 (99.63%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : splice 14
  stage execs : 10/25 (40.00%)
  total execs : 1.17M
  exec speed : 22.19/sec (slow!)
fuzzing strategy yields
  bit flips : n/a, n/a, n/a
  byte flips : n/a, n/a, n/a
  arithmetics : n/a, n/a, n/a
  known ints : n/a, n/a, n/a
  dictionary : n/a, n/a, n/a
  havoc : 1195/378k, 1741/763k
  trim : n/a, n/a
map coverage
  map density : 4.59% / 12.18%
  count coverage : 2.01 bits/tuple
findings in depth
  favored paths : 660 (22.49%)
  new edges on : 2374 (80.89%)
  total crashes : 4 (4 unique)
  total tmouts : 0 (0 unique)
overall results
  cycles done : 3
  total paths : 2935
  uniq crashes : 4
  uniq hangs : 0
path geometry
  levels : 38
  pending : 1081
  pend fav : 4.29G
  own finds : 2932
  imported : n/a
  stability : 25.30%

^C [cpu001: 14%]
```

图 4-13 AFLNet 静态插桩模式运行截图

```
american fuzzy lop 2.56b (hello-crow-static-arm-8081)

process timing
  run time : 0 days, 11 hrs, 6 min, 21 sec
  last new path : 0 days, 0 hrs, 0 min, 13 sec
  last uniq crash : 0 days, 0 hrs, 6 min, 1 sec
  last uniq hang : none seen yet
cycle progress
  now processing : 1400 (94.72%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : splice 4
  stage execs : 1/16 (6.25%)
  total execs : 305k
  exec speed : 7.54/sec (zzzz...)
fuzzing strategy yields
  bit flips : n/a, n/a, n/a
  byte flips : n/a, n/a, n/a
  arithmetics : n/a, n/a, n/a
  known ints : n/a, n/a, n/a
  dictionary : n/a, n/a, n/a
  havoc : 452/62.3k, 1028/230k
  trim : n/a, n/a
map coverage
  map density : 23.41% / 29.05%
  count coverage : 1.23 bits/tuple
findings in depth
  favored paths : 544 (36.81%)
  new edges on : 1117 (75.58%)
  total crashes : 5 (5 unique)
  total tmouts : 0 (0 unique)
overall results
  cycles done : 0
  total paths : 1478
  uniq crashes : 5
  uniq hangs : 0
path geometry
  levels : 14
  pending : 602
  pend fav : 0
  own finds : 1475
  imported : n/a
  stability : 27.01%

^C [cpu002: 11%]
```

图4-14 AFLNet动态插桩模式运行截图

```
[17225.701411] [
202] 0 202 3641 258 22528 0
american fuzzy lop 2.56b (qemu-system-arm)80 0
[+] Test Alive timed out, res = 7168

process timing
  run time : 0 days, 7 hrs, 36 min, 43 sec
  last new path : 0 days, 0 hrs, 1 min, 9 sec
  last uniq crash : 0 days, 2 hrs, 49 min, 14 sec
  last uniq hang : 0 days, 5 hrs, 16 min, 46 sec
cycle progress
  now processing : 2119* (90.94%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : havoc
  stage execs : 0/204 (0.00%)
  total execs : 481k
  exec speed : 1.36/sec (zzzz...)
fuzzing strategy yields
  bit flips : n/a, n/a, n/a
  byte flips : n/a, n/a, n/a
  arithmetics : n/a, n/a, n/a
  known ints : n/a, n/a, n/a
  dictionary : n/a, n/a, n/a
  havoc : 1244/224k, 1089/234k
  trim : n/a, n/a
map coverage
  map density : 2.66% / 8.33%
  count coverage : 2.43 bits/tuple
findings in depth
  favored paths : 217 (9.31%)
  new edges on : 2033 (87.25%)
  total crashes : 6 (6 unique)
  total tmouts : 28.0k (500+ unique)
overall results
  cycles done : 1
  total paths : 2330
  uniq crashes : 6
  uniq hangs : 500+
path geometry
  levels : 21
  pending : 1664
  pend fav : 0
  own finds : 2327
  imported : n/a
  stability : 13.69%

[+] Test Alive timed out, res = 7168 [cpu000: 26%]
```

图 4-15 AFLNetSpy 系统模式下运行截图

# 北京理工大学本科生毕业设计（论文）

```

czz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ ./replay-crash-8888 outputs-qemu-system/replayable-crashes/id\000003\,sig\00\,src\001899+000255\,op\splice\,rep\0
Size of the current packet 1 is 126

Responses from server:0-200-
++++
Responses in details:
HTTP/1.1 200 OK
Content-Length: 23
Server: Crow/1.1.0
Date: Sat, 18 May 2024 05:56:57 GMT
data: 1888 = 1987519436

czz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ curl 127.0.0.1:8080
Hello worldczz@xuan-918:~/afl-workspace/aflnet/spy-test-http$

czz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ ./replay-crash-8888 outputs-qemu-system/replayable-crashes/id\000003\,sig\00\,src\001899+000255\,op\splice\,rep\0
Size of the current packet 1 is 126

Responses from server:0-
++++
Responses in details:
Hello worldczz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ curl 127.0.0.1:8081
Hello worldczz@xuan-918:~/afl-workspace/aflnet/spy-test-http$

czz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ ./replay-crash-18808 outputs-qemu-system/replayable-crashes/id\000003\,sig\00\,src\001899+000255\,op\splice\,rep\0
Size of the current packet 1 is 126

Responses from server:0-
++++
Responses in details:
czz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ curl 127.0.0.1:18808
curl: (52) Empty reply from server

czz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ ./hello-crow-static-inst-8888
(2024-05-18 05:56:53) [INFO] Crow/1.1.0 server is running at http://0.0.0.0:8088 using 2 threads
(2024-05-18 05:56:53) [INFO] Call 'app.loglevel(crow:LogLevel::Warning)' to hide Info level logs.
(2024-05-18 05:56:57) [INFO] Request: 127.0.0.1:52739 0xb650c0 HTTP/0.9 GET /data/1888
(2024-05-18 05:56:57) [INFO] Response: 0xb650c0 /data/1888 200 0
(2024-05-18 05:57:34) [INFO] Request: 127.0.0.1:48394 0x72014000c30 HTTP/1.1 GET /
(2024-05-18 05:57:34) [INFO] Response: 0x7f2014000c30 / 200 0

czz@xuan-918:~/afl-workspace/aflnet/spy-test-http$ ./hello-crow-static-arm-8881
(2024-05-18 05:54:07) [INFO] Crow/1.1.0 server is running at http://0.0.0.0:8081 using 2 threads
(2024-05-18 05:54:07) [INFO] Call 'app.loglevel(crow:LogLevel::Warning)' to hide Info level logs.
(2024-05-18 05:54:10) [INFO] Request: 127.0.0.1:46564 0x23d8b0 HTTP/0.9 GET /data/1888
(2024-05-18 05:54:10) [INFO] Response: 0x23d8b0 /data/1888 200 0
(2024-05-18 05:54:16) [INFO] Request: 127.0.0.1:46576 0x3ec0668 HTTP/1.1 GET /
(2024-05-18 05:54:16) [INFO] Response: 0x3ec0668 / 200 0

root@romulus:~# kill -9 192
root@romulus:~# hello-crow
(2024-05-18 06:03:12) [INFO] Crow/1.1.0 server is running at http://0.0.0.0:8088 using 2 threads
(2024-05-18 06:03:12) [INFO] Call 'app.loglevel(crow:LogLevel::Warning)' to hide Info level logs.
(2024-05-18 06:03:25) [INFO] Request: 10.0.2.2:40374 0x165b578 HTTP/0.9 GET /data/1888

*Segmentation fault (core dumped)
root@romulus:~#

```

图 4-16 三种情况处理同一请求结果对比

## 结 论

本文提出了一种基于QEMU插件实现系统级灰盒模糊测试的方法，并基于该方法实现了一个新颖的模糊测试框架AFLNetSpy，它基于AFLNet，通过结合QEMU的AFL-SPY插件，能够实现对固件中网络应用的灰盒模糊测试。本文首先介绍了模糊测试领域的研究概况，并对和本文工作相关的几项工作的特点和不足进行了较为详细的说明。然后在第三章阐述了系统级灰盒模糊测试方法的具体内容，并对AFLNetSpy系统的整体架构和实现细节进行了详细的讲解。最后，在第四章从有效性、稳定性和性能三个方面出发进行实验设计，将AFLNetSpy系统和AFLNet的两种模式进行比较分析，从而验证AFLNetSpy系统的可用性。

本文所构建的AFLNetSpy系统的主要创新点包括：

- (1) 通过QEMU-SPY监测客户机系统中目标进程和网络相关的系统调用，精准识别目标进程处理测试请求的时间区间。
- (2) 结合QEMU的HELPER机制和插件机制实现客户机系统中目标进程的标识，并进一步实现目标进程代码执行信息的收集。
- (3) QEMU-SPY的关键功能以QEMU插件的形式实现，即AFL-SPY插件(libaflspy.so)，使得系统具有较高的可扩展性。

本文提出的系统级灰盒模糊测试方法，以及开发出的AFLNetSpy原型系统，将灰盒模糊测试方法成功拓展到了固件网络应用领域，对于促进相关领域的研究具有重要意义。本文开发的AFLNetSpy系统及相关代码，可访问代码仓库链接(<https://github.com/czxvan/AFLNetSpy>)进行查看。

同时，本文的工作还存在一定不足，如

- (1) 局限于固件内的网络应用而难以泛化到非网络应用。本文通过监测网络应用的特殊系统调用来确定目标进程的页目录地址，同时通过发送测试请求对目标进程进行崩溃判断，当目标应用为非网络应用时，这些方法就难以发挥作用。
- (2) 效率还有提升空间。尽管重复利用同一个测试进程执行多次测试，避免了大量fork的时间开销，但由于网络通信固有的延迟性，最终表现出的测试效率仍然较低。

上述问题有待后续进一步研究。

## 参考文献

- [1] 任泽众, 郑晗, 张嘉元, 等. 模糊测试技术综述[J/OL]. 计算机研究与发展, 2021, 58(5): 944-963. DOI:10.7544/issn1000-1239.2021.20201018 .
- [2] Zhu X, Wen S, Camtepe S, et al. Fuzzing: A Survey for Roadmap[J]. ACM Computing Surveys, 2022, 54(11s): 1-36.
- [3] 计江安, 井靖, 王奕森, 等. 嵌入式固件模糊测试研究综述[J/OL]. 小型微型计算机系统. <https://link.cnki.net/urlid/21.1106.TP.20231221.1414.006> .
- [4] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.
- [5] Sutton M, Greene A. The art of file format fuzzing//Proceedings of the Black Hat USA. Las Vegas, USA, 2005.
- [6] Mozilla Fuzzing Security. Funfuzz: a collection of fuzzers in a harness for testing the spidermonkey javascript engine. USA: Mozilla Fuzzing Security. 2008.
- [7] Zalewski M. American Fuzzy Lop[J]. 2016. <https://lcamtuf.coredump.cx/afl/> .
- [8] Vyukov D, Konovalov A. Syzkaller: an unsupervised coverage-guided kernel fuzzer. USA: Google. 2015.
- [9] Serebryany K. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. <https://github.com/google/oss-fuzz>.
- [10] Google. ClusterFuzz: scalable fuzzing infrastructure. USA: Google. 2019.
- [11] Böhme M, Pham V-T, Roychoudhury A. Coverage-based Greybox Fuzzing as Markov Chain[A]. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security[C]. Vienna Austria: ACM, 2016: 1032-1043.
- [12] Lyu C, Ji S, Zhang C, et al. MOPT: Optimized Mutation Scheduling for Fuzzers[J].
- [13] Circumventing Fuzzing Roadblocks with Compiler Transformations[J]. 2016. <https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/>.
- [14] Aschermann C, Schumilo S, Blazytko T, et al. REDQUEEN: Fuzzing with Input-to-State Correspondence[A]. Proceedings 2019 Network and Distributed System Security Symposium[C]. San Diego, CA: Internet Society, 2019.
- [15] Schumilo S, Aschermann C, Gawlik R, et al. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels[J].
- [16] Michael Eddington. Peach fuzzing platform[J]. <https://peachtech.gitlab.io/peach-fuzzer-community>.
- [17] Pham V-T, Boehme M, Santosa A E, et al. Smart Greybox Fuzzing[J]. IEEE Transactions on Software Engineering, 2020: 1-1.
- [18] Fioraldi A, Maier D, Eißfeldt H, et al. AFL++: Combining Incremental Steps of Fuzzing Research[J].
- [19] Fioraldi A, Maier D C, Zhang D, et al. LibAFL: A Framework to Build Modular and Reusable Fuzzers[A]. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security[C]. New York, NY, USA: Association for Computing Machinery, 2022: 1051-1065.
- [20] Pham V-T, Bohme M, Roychoudhury A. AFLNET: A Greybox Fuzzer for Network Protocols[A]. 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)[C]. Porto, Portugal: IEEE, 2020: 460-465.

- [21] Andronidis A, Cadar C. SnapFuzz: high-throughput fuzzing of network applications[A]. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis[C]. Virtual South Korea: ACM, 2022: 340-351.
- [22] Muench M, Stijohann J, Kargl F, et al. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices[A]. Proceedings 2018 Network and Distributed System Security Symposium[C]. San Diego, CA: Internet Society, 2018.
- [23] Chen J, Diao W, Zhao Q, Zuo C, Lin Z, Wang X, Lau WC, Sun M, Yang R, Zhang K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. InNDSS 2018 Feb 18.
- [24] Zhang H, Lu K, Zhou X, Yin Q, Wang P, Yue T. SloTFuzzer: fuzzing web interface in IoT firmware via stateful message generation. Applied Sciences. 2021 Apr 1;11(7):3120.
- [25] Hertz J, Newsham T. Project Triforce: Run AFL On Everything[J].
- [26] Bellard F. QEMU, a Fast and Portable Dynamic Translator[J]. 2005.
- [27] Zheng Y, Davanian A, Yin H, et al. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation[A]. 2019: 1099-1114.
- [28] Zheng Y, Li Y, Zhang C, et al. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation[A]. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis[C]. Virtual South Korea: ACM, 2022: 417-428.
- [29] Meng R, Mirchev M, Böhme M, et al. Large Language Model guided Protocol Fuzzing[A]. Proceedings 2024 Network and Distributed System Security Symposium[C]. San Diego, CA, USA: Internet Society, 2024.
- [30] Deng Y, Xia CS, Peng H, Yang C, Zhang L. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. InProceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis 2023 Jul 12 (pp. 423-435).
- [31] Deng Y, Xia CS, Yang C, Zhang SD, Yang S, Zhang L. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. InProceedings of the 46th IEEE/ACM International Conference on Software Engineering 2024 Feb 6 (pp. 1-13).
- [32] Xia CS, Paltenghi M, Le Tian J, Pradel M, Zhang L. Fuzz4all: Universal fuzzing with large language models. InProceedings of the IEEE/ACM 46th International Conference on Software Engineering 2024 Apr 12 (pp. 1-13).
- [33] Zhang C, Bai M, Zheng Y, Li Y, Xie X, Li Y, Ma W, Sun L, Liu Y. Understanding large language model based fuzz driver generation. arXiv preprint arXiv:2307.12469. 2023 Jul 24.
- [34] Henderson A, Prakash A, Yan L K, et al. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform[A]. Proceedings of the 2014 International Symposium on Software Testing and Analysis[C]. San Jose CA USA: ACM, 2014: 248-258.
- [35] Dovgalyuk P, Fursova N, Vasiliev I, et al. QEMU-based framework for non-intrusive virtual machine instrumentation and introspection[A]. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering[C]. Paderborn Germany: ACM, 2017: 944-948.

## 致 谢

本科生涯即将结束，回顾过去四年，可以看到自己在能力上的提升和性格上的改变，这是四年北理时光为我留下的印记。

首先，感谢谭毓安老师对本次毕业设计中几个关键点的建议和指导，感谢李元章老师对论文提出的修改意见，感谢实验室的师兄和同学提供的帮助，感谢我的家人们对我的鼓励和支持。

然后，感谢一起参与各项竞赛的队友以及网安俱乐部的朋友们，和你们一起成长进步讨论交流，让我受益匪浅。感谢我的舍友们，四年里尤其是最后一年我们一起度过了大部分的时光，对此我感到十分幸运和愉快。愿友谊长青，时常联系。

最后，祝大学四年里遇到的老师们身体健康，工作顺利。祝我的同学朋友们前程似锦，后会有期。