

AFL-VTest:航天嵌入式软件模糊测试框架

王帅, 黄晨, 江云松, 肖喜, 王冠霖, 于婷婷, 许奇臻

引用本文

王帅, 黄晨, 江云松, 肖喜, 王冠霖, 于婷婷, 许奇臻. [AFL-VTest:航天嵌入式软件模糊测试框架](#)[J]. 计算机科学, 2025, 52(12): 9-17.

WANG Shuai, HUANG Chen, JIANG Yunsong, XIAO Xi, WANG Guanlin, YU Tingting, XU Qizhen. [AFL-VTest:Fuzzing Framework for Aerospace Embedded Software](#) [J]. Computer Science, 2025, 52(12): 9-17.

相似文献推荐 (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)

[一种结合静态分析的轻量化内存安全运行时检测方法](#)

Lightweight Memory Safety Runtime Detection Method Combined with Static Analysis
计算机科学, 2025, 52(11A): 241100060-8. <https://doi.org/10.11896/jsjcx.241100060>

[网络协议模糊测试技术研究进展](#)

Survey on Fuzz Testing Techniques for Network Protocols
计算机科学, 2025, 52(11A): 241100173-9. <https://doi.org/10.11896/jsjcx.241100173>

[自动化软件缺陷定位技术研究](#)

Advances in Automatic Software Defect Location Techniques
计算机科学, 2025, 52(11A): 250200024-14. <https://doi.org/10.11896/jsjcx.250200024>

[C2P-YOLO:一种轻量级的风电塔筒裂缝检测算法](#)

C2P-YOLO:A Lightweight Crack Detection Algorithm for Wind Turbine Towers
计算机科学, 2025, 52(11A): 250100126-6. <https://doi.org/10.11896/jsjcx.250100126>

[基于生成式数据增强与Faster-RCNN改进的发动机打刻面缺陷检测](#)

Defect Detection of Engine Engraved Surface Based on Generative Data Augmentation and Improved Faster-RCNN
计算机科学, 2025, 52(11A): 241200025-7. <https://doi.org/10.11896/jsjcx.241200025>

AFL-VTest: 航天嵌入式软件模糊测试框架

王帅¹ 黄晨² 江云松² 肖喜¹ 王冠霖¹ 于婷婷² 许奇臻³

1 清华大学深圳国际研究生院 广东 深圳 518055

2 北京控制工程研究所 北京 100094

3 厦门市软件供应链安全公共技术服务平台 福建 厦门 361000

(iwangshuai@foxmail.com)

摘要 航天嵌入式软件的可靠性是确保航天任务成功执行的关键之一。模糊测试已经成为当今缺陷检测与漏洞挖掘的一种主流方式,并在软件安全领域取得了较大的成功。研究针对航天嵌入式软件的模糊测试方法,对于强化此类软件的可靠性、推动航天科技的进步具有深远意义。因此,提出了一套面向航天嵌入式软件的模糊测试框架 AFL-VTest。AFL-VTest 针对航天嵌入式软件内存资源受限和包含较多校验和检查的特点,分别提出了一种精简源码插桩方法与一种校验和修补算法,在多个样例程序及实际航天嵌入式程序上的评估实验结果表明了所提精简源码插桩方法和校验和修补算法的有效性。最后,AFL-VTest 成功揭露了实际项目中未曾被发现的 3 个缺陷,从而证明了其在提升航天嵌入式软件安全性与可靠性方面的有效性与实用价值。

关键词: 嵌入式软件;模糊测试;软件测试;缺陷检测;源代码插桩

中图分类号 TP311

AFL-VTest: Fuzzing Framework for Aerospace Embedded Software

WANG Shuai¹, HUANG Chen², JIANG Yunsong², XIAO Xi¹, WANG Guanlin¹, YU Tingting² and XU Qizhen³

1 Tsinghua Shenzhen International Graduate School, Shenzhen, Guangdong 518055, China

2 Beijing Institute of Control Engineering, Beijing 100094, China

3 Xiamen Software Supply Chain Security Public Technology Service Platform, Xiamen, Fujian 361000, China

Abstract The reliability of aerospace embedded software is a critical determinant of space mission success. Fuzzing has become the mainstream method for defect detection and vulnerability discovery today, and has achieved significant success in the field of software security. The research on fuzzing methods for aerospace embedded software has profound significance for enhancing the reliability of such software and promoting the progress of aerospace technology. Therefore, this paper proposes AFL-VTest, a fuzz testing framework specifically designed for aerospace embedded software. It integrates a streamlined source code instrumentation method and a novel checksum-fixing algorithm tailored to address limited memory resources and the prevalence of checksum verifications in embedded systems. Evaluation experiments conducted on multiple sample programs and practical aerospace embedded software demonstrate the effectiveness of the proposed instrumentation method and checksum fixing algorithm. Finally, AFL-VTest successfully uncovers three previously undetected defects within the actual aerospace embedded software projects, thus verifying the effectiveness and practical value of the proposed method in bolstering the safety and reliability of aerospace systems.

Keywords Embedded software, Fuzz testing, Software testing, Defect detection, Source-level instrumentation

1 引言

航天嵌入式软件在航天活动中扮演着至关重要的角色,它控制和检测航天器的各种功能,包括导航、姿态控制、通信和数据处理等,保证航天器的安全、可靠和稳定运行。航天嵌

入式软件的质量和安​​全直接关系到航空航天器的性能和安​​全。然而,随着航天技术的发展,特别是随着软件定义卫星设计理念的提出,航天嵌入式软件的规模不断增大,复杂度不断提升,导致软件潜在缺陷也随之增多,并且越来越难以检测,数组越界、算术溢出、除零等软件缺陷时有发生^[1]。同时随着

到稿日期:2025-04-29 返修日期:2025-09-07

基金项目:广东省自然科学基金(2025A1515011946);厦门市软件供应链安全公共技术服务平台(3502Z20231042)

This work was supported by the Natural Science Foundation of Guangdong Province (2025A1515011946) and Xiamen Software Supply Chain Security Public Technology Service Platform(3502Z20231042).

通信作者:肖喜(xiaox@sz.tsinghua.edu.cn)

环境的改变,如商业航天发展、开源库的引入、太空军事竞赛等,航天器受到网络攻击的可能性也越来越大^[2]。

目前,航天嵌入式软件的质量主要通过软件测试、代码审查和静态分析^[1,3-6]来保证,但是这些方法存在需要耗费大量人力成本、效率低和质量不稳定等缺点。其中软件测试是目前保证航天嵌入式软件可靠性和安全性的重要手段之一,可以在软件正式应用于实际任务之前找到其中的缺陷并修复,从而提高软件的可靠性和安全性。传统的做法是根据程序设计文档或源代码,人工或根据规则自动地构造输入,对程序进行测试,但这种方法存在效率低、边缘情况覆盖不足、质量不稳定等问题,特别是测试大型项目时耗时费力,效率低下。

表 1 不同方法优缺点对比

Table 1 Comparison of advantages and disadvantages of different methods

方法	软件测试	代码审查	静态分析	模糊测试
执行方式	手动或半自动生成测试用例	人工阅读分析代码	工具扫描	自动生成用例输入并监控
缺陷类型	功能错误、逻辑错误等	逻辑、设计缺陷、编码规范等	编码规范、潜在安全漏洞	内存损坏、崩溃、安全漏洞
优点	灵活、理解上下文	发现早期缺陷	速度快、覆盖高、自动化	发现未知深漏洞、自动化
缺点	慢、覆盖率有限、成本高	耗时、依赖经验、难查运行时错误	高误报、漏报、难查运行时错误	覆盖率盲点、难查逻辑错误

由于嵌入式软件自身的特点,将模糊测试应用于嵌入式软件测试存在以下难点:

1)相比桌面软件系统,嵌入式软件系统与外设的关系更紧密且资源有限,因此需要考虑外设约束与资源约束。

2)嵌入式软件系统具有多样性的特点,包括操作系统、CPU架构、外设和通信方式的多样性等,要求模糊测试方法具有通用性或可扩展性。

3)嵌入式软件系统具有封闭性,其崩溃检测和分析会更加复杂,崩溃可能表现为特定功能异常而非系统崩溃,导致外界难以检测,其崩溃检测率仅为桌面软件(如 x86-64)的 37%左右^[8]。

4)传统的插桩机制难以应用到嵌入式软件系统。现有的插桩机制主要有源代码插桩和二进制插桩两种类型,然而嵌入式软件系统的源代码可能难以获取,即使可以获取,源码插桩后也会导致二进制代码膨胀突破嵌入式系统资源限制,程序可能运行失败;同时,嵌入式系统中的二进制程序由于架构和操作系统的多样性,现有二进制插桩工具也难以应用。

研究面向航天嵌入式软件的模糊测试技术,对于进一步保障航天嵌入式软件的可靠性具有重要意义。结合航天嵌入式软件测试的现状和嵌入式软件模糊测试的难点,本文实现了一种针对航天嵌入式软件的模糊测试方法。本文的主要贡献如下:

1)提出了一种针对航天嵌入式软件的模糊测试框架 AFL-VTest, AFL-VTest 实现了覆盖率引导的航天嵌入式软件灰盒模糊测试;

2)针对航天嵌入式软件内存资源受限的特点,提出了一种精简源代码插桩方法,缓解了源码插桩导致的二进制代码尺寸膨胀造成的程序异常问题;

3)针对航天嵌入式软件中存在大量校验和验证的特点,提出了一种架构无关的自动校验和修补算法,显著提升了代码覆盖率;

4)在两个真实航天嵌入式软件上对 AFL-VTest 进行了评估实验,成功揭露了 2 个实际项目中潜藏的 3 个缺陷。

模糊测试是一种自动化软件测试技术,其核心思想是通过自动或半自动方式构造大量测试用例,将其输入被测程序,并通过监视程序,来发现越界访问、断言失败等程序异常。近年来,模糊测试在应用软件的测试中取得了举世瞩目的成就。截止到 2023 年 8 月,Google 开发和维护的开源模糊测试平台 OSS-FUZZ^[7]在 1000 多个开源项目中发现了超过 36000 个漏洞;模糊测试在嵌入式软件领域的影响也越来越深远^[8-9]。Scharnowski 等^[10]使用嵌入式模糊测试工具 FUZZWARE^[11]测试卫星固件,成功在 3 个卫星固件中找到了 6 个漏洞,证实了模糊测试用于航天嵌入式软件的可行性。软件测试、代码审查、静态分析以及模糊测试方法的优缺点如表 1 所列。

2 背景和相关工作

2.1 覆盖率引导的灰盒模糊测试

覆盖率引导的灰盒模糊测试的基本思想是,通过不断提高模糊测试过程中的代码覆盖率来提高发现程序缺陷的可能性。目前流行的模糊测试工具有 AFL^[12], libFuzzer^[13]和 Honggfuzz^[14]等,它们都属于覆盖率引导的灰盒模糊测试。

覆盖率引导的灰盒模糊测试基本流程如图 1 所示,可以概括为以下 5 个步骤:1)插桩,在编译被测程序时,模糊测试工具插入可以获取代码覆盖率的代码;2)种子调度,根据一定策略从种子池中选择一个种子文件;3)变异,根据变异策略分配能量,即变异次数,应用不同的变异算子对挑选出的种子文件进行变异;4)输入并执行测试,将变异后的种子文件输入目标程序执行测试;5)反馈,监视器监控程序执行状态以及代码覆盖率,如果当前种子变异后可以覆盖新的代码则保存变异后的种子文件至种子池,如果导致程序崩溃则报告并保存当前种子文件为异常用例。然后跳转至步骤 2)继续模糊测试循环。

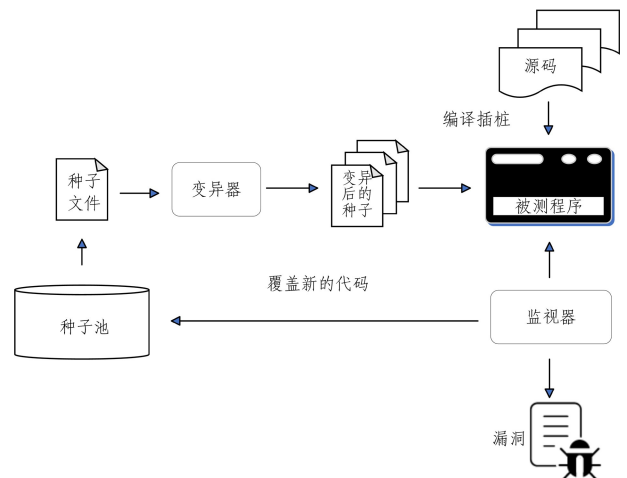


图 1 覆盖率引导的灰盒模糊测试流程

Fig. 1 Overview of coverage-based greybox fuzzing

2.2 嵌入式模糊测试

针对嵌入式软件系统复杂多样的特点,研究者已提出多种嵌入式软件模糊测试方法。目前,嵌入式模糊测试根据执行环境的不同大致可以分为基于真实设备的(接口)模糊测试和基于模拟执行的模糊测试两种类型。

2.2.1 基于真实设备的(接口)模糊测试

基于真实设备的(接口)模糊测试是指将待测固件直接运行于真实嵌入式设备上进行模糊测试。由于嵌入式设备外设的多样性以及嵌入式设备的性能,这种方法通常存在扩展性较差和性能较差的问题;同时,由于架构和无法获取源码等问题,插桩困难,因此大部分此类模糊测试方法是黑盒的。

对于存在操作系统(如 Linux)的嵌入式系统,通常采用覆盖率反馈的灰盒模糊测试。ARM-AFL^[15]是一种基于覆盖率引导的模糊测试框架,它在编译时插桩并直接在实际设备上执行灰盒模糊测试。AFLIoT^[16]设计了一种针对 ARM 平台的二进制静态插桩技术,可以对 Linux 操作系统的物联网程序进行通用的灰盒模糊测试。Tardis^[17]通过对源代码插桩实现了与操作系统无关的代码覆盖率收集和分析,能够测试多种嵌入式操作系统,而无需大量的手动调整。Biff^[18]是一个应用在物联网或移动设备程序上的二进制模糊测试器,其使用 Skorpion^[19]进行动态二进制插桩。

除了传统的二进制插桩方法,嵌入式系统还可以通过调试接口或侧信道方法获取覆盖率。GDBFuzz^[20]是一种使用调试接口进行模糊测试的方法,其关键思想是基于目标程序的控制流图,利用嵌入式系统中的调试单元和硬件断点在代码中系统地设置断点,并根据触发断点的情况获取覆盖率信息。 μ AFL^[21]、Fw-fuzz^[22]以及 Beckmann 等提出的方法^[23]同样使用调试接口实现嵌入式模糊测试。在难以插桩的场合,有研究者提出可以使用侧信道方法。例如,Sperl 等^[24]介绍了一种基于侧信道的模糊测试方法,通过功耗信息从嵌入式设备中提取反馈信息。

对于物联网设备,由于固件通常只存在二进制文件,因此常用黑盒模糊测试方法。SNIPUZZ^[25]是用于测试物联网设备固件安全性的黑盒模糊测试框架,它利用设备返回的响应消息建立反馈机制,用于指导模糊测试的变异过程,并通过推断消息片段中每个字节的特性以生成符合设备语法的测试用例。IoTFuzzer^[26]首先分析应用程序,找到与网络相关的方法和数据编码方法之间的路径,然后对这些路径上第一个处理用户输入的函数参数进行模糊处理,生成有效的模糊测试输入。DIANE^[27]通过从物联网设备伴生应用程序中调用特定的函数来模糊测试物联网设备,这些函数被称为模糊触发器。当执行这些模糊触发器时,生成的输入既不受应用程序端验证的限制,又具有良好的结构,不会被设备直接丢弃。

2.2.2 基于模拟执行的模糊测试

由于真实的嵌入式设备复杂多样,基于真实设备的(接口)模糊测试方法存在扩展性差的缺点。模拟器通过模拟指令在真实情况下的行为,能够快速测试软件或硬件的性能和

行为,且可以很方便地进行扩展,因此目前有大量基于模拟执行的方法。

对于存在操作系统的嵌入式设备,通常使用 QEMU^[28]作为模拟器,按照模拟程度通常可以分为用户模式模拟、全系统模拟以及增强改进的用户模式模拟。IoTSFT^[29]使用 QEMU 的用户模式来构建运行环境并使用静态二进制插桩获取覆盖率信息。Prospect^[30]使用 QEMU 的全系统模式仿真整个嵌入式系统,并通过代理方式将外设请求转发给真实设备,模糊测试时,将捕获的外设数据变异之后传递给代理接口。FirmAFL^[31],FIRM-COV^[32],EQUAFL^[33]使用增强改进的 QEMU 用户模式,即只对运行在用户态的代码进行模拟执行,内核态使用宿主机系统调用替代,极大地提高了模糊测试的效率。

嵌入式系统,尤其是单片机嵌入式系统模糊测试的一大难点是如何处理多种多样的外设输入,包括 MMIO(内存映射 I/O)、DMA(直接存储访问)和中断等。目前,研究者提出了 3 种方法:基于模式的外设建模、基于符号执行的外设建模和基于硬件抽象的模拟技术。P2IM^[34]是一种基于模式的建模方法,通过 MMIO 访问模式将外设寄存器分为状态寄存器、数据寄存器等,对不同的寄存器访问采用不同的处理方法。对于中断,首先从虚拟中断控制器获取使能的中断,然后每隔固定数量基本块使用轮询算法去触发中断。DICE^[35]通过观察嵌入式设备与外设交互时的 DMA 访问模式,对 DMA 建模。VeRa^[36]针对 P2IM 存在的没有正确处理状态寄存器、控制状态寄存器和更多异常的问题,提出了改进方法。Fuzzware^[11]在 MMIO 访问代码附近使用动态符号执行技术,对 MMIO 访问建模,并通过快照技术降低重启开销,该模型能够将模糊器产生的输入转换成硬件生成值。 μ Emu^[37]采用符号执行技术,通过无效性引导的推理方法,来模拟未知外设的正确访问点。HALucinator^[38]提出 HAL(硬件抽象库)模拟技术,首先通过库匹配技术从固件中匹配库函数,然后通过替代库函数取代实际的硬件操作。FIRMNANO^[39]使用 HAL 方法解决 DMA 访问的问题。EmberIO^[40]则通过控制重放输入方法处理 MMIO 访问。

嵌入式模糊测试的某些建模方法会将模糊器的原始输入分割至不同的上下文,导致效率降低。由于输入的离散不连续给输入-状态(Input-to-State)方法带来困难,SPLITS^[41]提出了一种针对此类离散输入的输入-状态映射方法。HOE-DUR^[42]分析了 Fuzzware 平坦输入的 3 个缺点,即丢失过程信息、丢失类型信息和不适用变异,提出通过识别不同的执行上下文,继而将输入调整为多流输入的方法。

2.3 VTest 和航天嵌入式软件测试

VTest^[43]是一款嵌入式系统仿真平台,支持被测软件无修改直接运行,构建了全数字仿真生态闭环,覆盖系统构建、数据处理、模型对接、测试用例自动生成与执行等流程,可满足多种需求。它提供图形化配置工具,可快速配置虚拟系统,支持多种开发工具集成和仿真系统生成。平台界面类似集成开发环境,支持调试、测试及故障注入,集成 SPARCV7/V8、PPC、龙芯等多款 CPU 及 1553 总线、CAN 总线、串口、AD 采

集、GPIO 等多种外设,广泛应用于航空航天等领域,并提供多种代码覆盖率分析功能。

目前,使用 VTest 对航天嵌入式软件测试的流程如下:1)根据设计文档,测试人员构建相应的测试环境,包括编译环境、各种虚拟外设开发等;2)根据功能需求文档,测试人员设计测试用例;3)生成测试报告。

2.4 模糊测试的路障问题与输入-状态变异策略

魔数 (Magic Number) 与 (嵌套) 校验和 (Checksum) 问题是模糊测试过程中的路障问题,航天嵌入式软件通常会遇到这两种问题。对于魔数问题 (见图 2 第 1 行),如果使用 AFL 的确定性变异策略,最坏可能需要 65 536 次变异才能通过。REDQUEEN^[44] 提出输入-状态 (Input-to-State) 变异策略,其主要思想是获取程序运行过程中比较指令 (如 `cmp`) 或比较函数 (如 `strcmp`) 的左右参数,然后把参数直接或经过简单数学、编码变换后替换原输入的某些位置,在最坏情况下只需数次就可以通过此类检验问题。

```

1. if(u16(buffer) == 0x1234)
2.     u16 length = u16(buffer+2)
3.     if(Xor(buffer+4, length+2) == u16(buffer+length+6))
4.         if(Add(buffer+4, length) == u16(buffer+length+4))
5.             switch(u16(buffer+4)) {
6.                 case 0x11: handle_11(); break;
7.                 case 0x22: handle_22(); break;
8.                 default: break;
9.             }

```

图 2 魔数与(嵌套)校验和示例程序

Fig. 2 Example of magic number and (nested) checksum

对于(嵌套)校验和问题 (见图 2 第 3-4 行) 只用原输入-状态方法也无法解决,因为内层比较参数的获取依赖于外层校验和的成立 (如图 2 第 3 行条件不满足时,第 4 行比较参数就无法获取,而要通过双层校验和需要先用第 4 行比较参数替换,再替换第 3 行比较参数)。REDQUEEN 等通过补丁 (Patch) 方式解决此问题。首先通过一些启发式方法 (比较指令或函数的左右参数均随输入改变而改变,且两参数之一满

足与输入的部分相等) 识别校验和检验相关的比较指令,比如可以识别出第 3 和第 4 行是校验和检验指令;然后替换第 3 行条件使之恒真 (例如对于 x86, 可以使用 `cmp al, al`), 这样就能获取到第 4 行比较指令。通过此方法可以建立比较指令之间的依赖关系,便于之后对比较指令进行拓扑排序,继而用排序结果逐层进行修补校验和。

3 AFL-VTest 设计与实现

3.1 整体框架

为了对航天嵌入式软件进行模糊测试,本文提出了 AFL-VTest, 其整体框架如图 3 所示。它主要由插桩模块 (Instrument)、模拟器 (Emulator) 和模糊器 (Fuzzer) 3 个部分组成。AFL-VTest 模糊测试流程可以概括为:1) 使用插桩模块对被测软件源代码编译插桩生成待测二进制可执行文件,插桩信息包括覆盖率、比较记录 (CMPLOG) 获取以及缺陷检测 (如数组越界、除 0) 代码等,插桩代码会将收集的信息写入 VTest 虚拟存储空间的共享内存区域 (见图 3 shm 区域), 通知测试夹具 (Harness) 虚拟设备收集处理这些信息并最终反馈给模糊器;2) 模糊器会启动 VTest 进程, VTest 加载插桩后的二进制程序并初始化各个虚拟外设以及 Harness 虚拟设备, Harness 虚拟设备被加载后将控制 VTest 模拟过程,通过管道和共享内存与模糊器通信;3) 开始一个测试循环,模糊器从种子池中选择出分数最高的种子进行变异,变异后的种子以字节流输入至 Harness 虚拟设备, Harness 虚拟设备接收到原始输入后,启动仿真过程。在仿真过程中,程序每执行一定数量的基本块 (Basic Block), Harness 就会主动触发中断并实时监控内存映射输入输出 (MMIO) 区域读写,读取原始输入中的一个或多个字节写入 MMIO 区域或通过现有的虚拟外设将原始输入转发至被测程序,待输入消耗完毕、程序发生运行时错误或达到指定的基本块数量时,结束本次测试循环并收集覆盖率信息、比较记录以及错误检测信息反馈给模糊器。随后在模糊引擎控制下进入下一个测试循环。

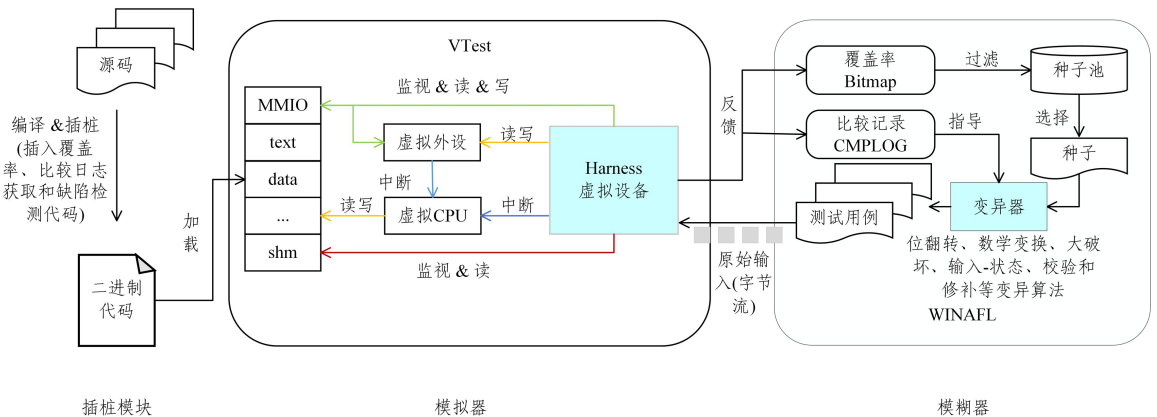


图 3 AFL-VTest 整体框架

Fig. 3 Overview of AFL-VTest

3.2 插桩模块

嵌入式程序通常难以获取源代码,因此现有的嵌入式模糊测试工具通常借助或修改模拟器仿真过程进行插桩,如在 QEMU 翻译成中间指令时插入指令。然而 VTest 是闭源

软件,无法实现上述插桩方案。考虑到对于航天嵌入式软件测试人员来说,一般可以获取项目源码,因此可以使用源代码插桩方法。然而,航天嵌入式软件一般硬件资源有限,内存大小受到限制,通常开发人员会通过人工计算内存布局来满足

硬件限制,此时源码插桩造成的二进制代码尺寸膨胀很可能导致软件执行失败,为此提出了一种精简源代码插桩方法,大大缓解了源码插桩造成的二进制代码尺寸膨胀。其原理是借助虚拟机的内存读写监视回调功能实现虚拟设备,将原本需要插桩代码(运行时)完成的功能转移到虚拟设备中,从而减少插桩代码指令的数量。

3.2.1 覆盖率获取

原始的源码插桩方法会为每个基本块(Basic Blocks)随机分配一个基本块 ID,代码执行时,每当进入一个基本块,会首先执行以下操作:从内存读取上一个基本块 ID,右移一位

后与当前基本块 ID 异或,作为边 ID 更新共享内存中的覆盖率位图(Bitmap),接着更新上一个基本块 ID。在此过程中,需要执行多次访存指令和算逻辑指令,导致代码段(Text)膨胀。

精简源码插桩方法同样会为每个基本块分配一个基本块 ID,与原始源码插桩不同的是,代码执行时,每当进入一个基本块,只需将当前基本块 ID 写入虚拟设备指定的 MMIO 地址,当写入完毕会立即触发虚拟设备中实现的写回调函数,之后由虚拟设备写回调函数完成之前由插桩代码完成的数据处理操作,从而大大减少了插入的指令数量。插桩前后程序执行流程如图 4 所示。

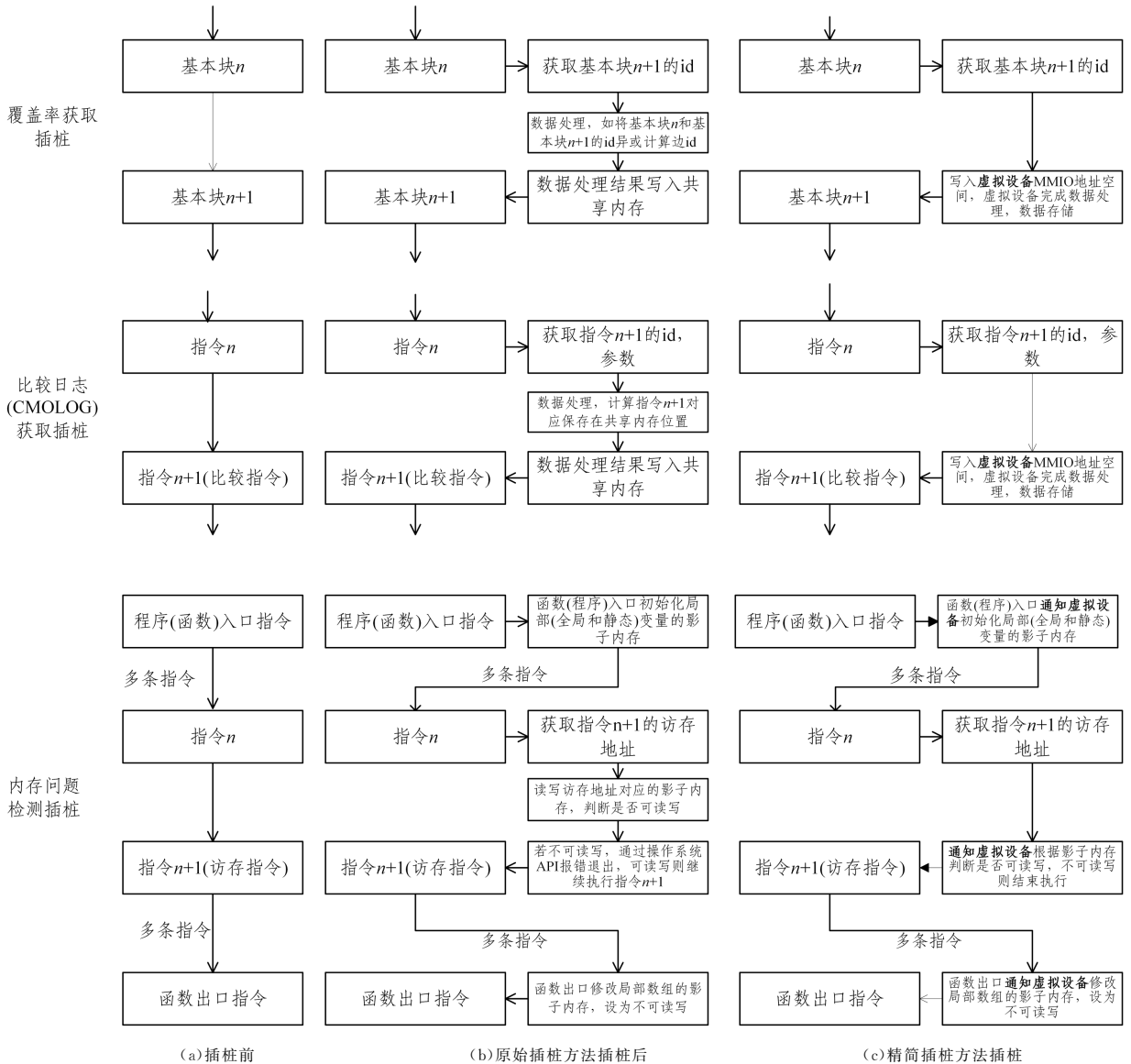


图 4 不同插桩方案对比

Fig. 4 Comparison of different instrumentation approaches

3.2.2 比较记录

比较记录(CMPLOG)是执行过程中执行的比较指令或比较函数的记录,每条记录包括当前比较的 ID、执行次数、左右参数以及数据类型等信息,原始插桩方案会在比较指令执行之前搜集这些信息写入共享内存,这需要插入大量指令,而精简之后只需要把必要信息写入虚拟设备指定的 MMIO 地址,后续处理逻辑由虚拟外设完成。插桩

前后程序执行流程如图 4 所示。

3.2.3 缺陷检测

为了解决嵌入式模糊测试崩溃检测问题,本文针对除 0 和内存缺陷开发了相应的检测工具。对于除 0,通过插桩方法对所有除法和求余指令的除数进行判断。对于内存问题,如果采用源代码插桩方法,检测问题则会相对变得简单,本文参照 Valgrind^[45], AddressSanitizer^[46] 等内存检测工具实现全

局部数组溢出、局部数组溢出检测等。其主要原理是在每块可以访问的内存之间增加红区(红区不可读写),并使用影子内存标记对应内存区域是否可以读写,若不可读写则报错。本文基于 AddressSanitizer 方法使用精简源码插桩实现了内存检测器。

原始插桩需要将影子内存初始化操作插入函数和程序入口处,将影子内存释放操作插入函数和程序出口处,以及将影子内存判读操作访存指令前。精简源码插桩方法把这些操作转移至虚拟外设中,由虚拟外设完成。插桩前后程序执行流程如图 4 所示。

3.3 Harness 虚拟设备

Harness 虚拟设备主要负责模糊器与 VTest 之间的交互,控制模糊测试的流程,是 AFL-VTest 具有扩展性的关键。一方面,它将模糊器需要的覆盖率、比较记录等信息提取出来,这是实现覆盖率引导的灰盒模糊测试以及输入-状态算法所必须的基本信息;另一方面,它将模糊器的输入以函数接口的方式进行封装,使得测试人员能够自由地使用这些原始输入,比如可以在被测程序读取外设寄存器时,直接使用这些原始输入的值或使用 VTest 已经实现的各类虚拟设备输入至待测程序。也就是说,对于不同的待测程序,只需简单修改此部件即可快速对其进行模糊测试。

3.4 校验和修补算法

现有方法通常采用基于补丁(Patch)的方法解决嵌套校验和问题:首先将外层校验和比较替换成恒为真的比较指令(例如在 x86 上使用 `cmp al, al`),然后使用输入-状态(Input-to-State)方法先修补内层校验和,再在满足内层校验和的基础上修补外层校验和。然而上述方法存在以下缺点:1)依赖于具体的架构,不同架构需要不同补丁方法,这依赖于专家知识;2)在某些情况下实现上述方法是困难的,如 AFL++^[4]的源码插桩模式。为了解决这两个问题,提出了一种近似的校验和修补算法。

原始的输入-状态算法^[44]分为染色和替换两个阶段。染色的主要作用是增加输入的熵,提高输入的随机性,染色要满足染色前后输入的路径不变。改进后的输入-状态算法则分为染色、替换及修补校验和 3 个阶段,具体如算法 1 所示。

算法 1 改进的染色算法 colorization

输入:原始输入(input)、校验和信息链(checksum_chain)

输出:染色后的输入(input)

```

1. ranges ← (1…len(input))
2. original_hash ← get_bitmap_hash(input)
3. for (rng, op) in checksum_chain do
4.   ranges ← remove_range(ranges, rng)
5. while rng = pop_biggest_range( ranges ) do
6.   backup ← input[rng]
7.   input[rng] ← random_bytes()
8.   input ← fixchecksum(input, checksum_chain)
9.   if original_hash != get_bitmap_hash(input) then
10.    add(ranges, (min(rng)…max(rng)/2))
11.    add(ranges, (max(rng)/2+1…max(rng)))
12.   input[rng] ← backup
13. return input

```

首先是染色,为了能够在存在校验和的情况下染色,本文对原有的染色算法进行如下改进。

首先,将校验和区域排除出染色范围(算法 1 第 3—4 行),再使用校验和修补算法修补校验和,这样就能在校验和正确的情况下完成染色。

然后是替换,这部分和原始的输入-状态算法相同,不同的是由于修补了外层校验和,因此可以获取内层校验和的比较指令,从而使用输入-状态算法完成内层校验和修补。

最后,使用校验和修补算法修补校验和(见算法 2),其主要思路是根据校验和修补链递归地从内而外地修补校验和。

算法 2 嵌套校验和修补算法 fixchecksum

输入:原始输入(input)、校验和信息链(checksum_chain)

输出:修补后的输入

```

1. if checksum_chain is None:
2.   return input
3. reverse_checksum_chain ← reverse(checksum_chain)
4. next_checksum_chain ← deepcopy(checksum_chain)
5. pop_front(next_checksum_chain)
6. fixchecksum(input, next_checksum_chain)
7. cmpmap ← get_cmpmap(input)
8. (rng, op) ← front(reverse_checksum_chain)
9. input[rng] ← op(input[rng], cmpmap)
10. fixchecksum(input, next_checksum_chain)
11. return input

```

在上述过程中,使用启发式的方法识别出可能的校验和比较指令(如染色前后比较指令的左右参数均发生变化),如果输入-状态算法可以成功找出不同路径,则保存当前替换的位置和替换方法,即(rng, op)到当前种子的校验和修补链队首,这样实际在模糊测试过程中就能完成原有需要拓扑排序才能实现的嵌套校验和识别及修补。

3.5 实现

本文使用 WinAFL^[48]作为模糊器, VTest 作为模拟器,基于 AFL++ 的 CMPLOG 获取和输入-状态算法以及 AddressSanitizer 实现了 AFL-VTest 原型。

4 实验

本章通过实验验证 AFL-VTest,并回答以下 3 个研究问题。

问题 1 AFL-VTest 插桩前后造成二进制代码膨胀带来了怎样的影响?

问题 2 校验和修补算法是否能提升模糊测试的效率?

问题 3 AFL-VTest 是否能够发现真实项目的缺陷?

其中,问题 1 主要评估精简源码插桩方法的有效性,问题 2 主要评估校验和修补算法的有效性,问题 3 则评估整个模糊测试框架的有效性。

4.1 实验设置

为了验证研究问题 1,大范围选择目标架构(SPARC)的项目进行实验,验证插桩前后目标码的大小,以及插桩前后目标码运行结果是否一致。为此,选取了 VTest 附带的 5 个样例程序 1553demo, candemo, iodemo, pindemo, timeruart 以及 ChecksumDemo(校验和样例程序,包含嵌套校验和处理,

参考图 2)和 2 个真实的航天嵌入式软件项目(ManagerSoft(某 CPU 软件,以下简称项目 1)和 HostControl(某主控软件,以下简称项目 2)作为测试对象;为了验证研究问题 2,即验证校验和修补算法是否有效,首先选择了 ChecksumDemo,其包含双重嵌套校验和验证过程的样例程序,同时由于需要验证校验和修补算法在真实情况下的效果,因此又选择了两个真实的航天嵌入式软件项目(项目 1 和项目 2)进行模糊测试;为了验证研究问题 3,对项目 1 和项目 2 进行长时间的模糊测试。所有实验运行在装有 Windows 7 操作系统的台式机上,该台式机配备 Intel^(R) Core^(TM) i5-7500 CPU @3.40 GHz 的中央中央处理器和 8 GB 内存,实验使用的 VTest 版本为 3.2.1。

4.2 插桩带来的影响

对 VTest 附带的 5 个样例程序 1553demo, candemo, iodemo, pindemo, timeruart 以及校验和检验样例程序 ChecksumDemo 和两个真实的项目(项目 1、项目 2)分别进行正常

编译和插桩编译,然后使用 size 命令分别获取插桩前后 text(代码)段、data(数据)段、bss 段和总计大小。

实验结果如表 2 所列,可以看出, text 段膨胀最大的是 ManagerSoft 项目,膨胀为插桩前的 1.91 倍; data 段膨胀最大的也是 ManagerSoft 项目,膨胀为插桩前的 26.29 倍。出现该现象的原因是此项目存在大量的全局数组,需要插入大量红区,因此数据段大幅膨胀。 bss 段膨胀最大的是 pindemo 和 timeruart 项目,均为插桩前的 16 倍。总计膨胀最大的是 ManagerSoft 项目,为插桩前的 1.65 倍。可以看出,最可能导致代码失效的 text 段和总计指标在所选项目的膨胀倍数均小于 2,其中 text 段膨胀为原来的 1~2 倍,总计膨胀为原来的 1~1.7 倍,甚至低于原有插桩方法仅使用内存检测器 AddressSanitizer 插桩时的 2.5 倍^[46],基本满足嵌入式设备对资源的要求。之后又使用 VTest 分别运行插桩前后的代码,对比发现插桩前后代码功能一致,插桩并不会改变程序原有功能,验证了插桩方法的有效性。

表 2 插桩前后代码尺寸变化

Table 2 Changes in code size before and after instrumentation

项目名称	插桩前				插桩后				比值			
	text	data	bss	总计	text	data	bss	总计	text	data	bss	总计
1553demo	10384	656	644	11684	11920	720	768	13408	1.15	1.10	1.19	1.15
candemo	9504	656	55	10215	10816	880	496	12192	1.14	1.34	9.02	1.19
iodemo	9216	656	2056	11928	10048	656	2056	12760	1.09	1.00	1.00	1.07
pindemo	8576	656	4	9236	8816	688	64	9568	1.03	1.05	16.00	1.04
timeruart	9616	656	4	10276	11216	752	64	12032	1.17	1.15	16.00	1.17
ChecksumDemo	9792	656	2064	12512	11616	752	2630	14998	1.19	1.15	1.27	1.20
ManagerSoft	68928	272	46516	115716	131776	7152	52468	191396	1.91	26.29	1.13	1.65
HostControl	96272	1536	8864	106672	161296	3168	8936	173400	1.68	2.06	1.01	1.63

4.3 校验和修补算法的有效性

为了验证校验和修补算法的有效性,选取 Checksum-Demo 和项目 1、项目 2 进行模糊测试,分别使用 AFL 基本的变异算法(Baseline)、基本变异算法附加输入-状态算法(i2s)以及基本变异算法附加校验和修补算法的输入-状态算法(i2s-fixchecksum)进行 6 个小时的模糊测试。其中 ChecksumDemo 使用 MMIO 和中断建模输入数据, ManagerSoft 使用 can 虚拟总线设备和 uart 虚拟总线设备输入, HostControl 使用 1553b 虚拟总线设备输入。

结果如图 5 所示。从 ChecksumDemo 项目结果可以看出, i2s-fixchecksum 算法覆盖率远高于其他两种方法,这是由于 ChecksumDemo 包含嵌套校验和检验,如果不能通过嵌套校验和检验,就无法执行主要的代码逻辑,因此隐藏其中的代码缺陷就无法被发现。对于 ManagerSoft(can),可以看出 i2s-fixchecksum 覆盖率远高于 baseline 并略高于 i2s。对于 ManagerSoft(uart),在模糊测试前期使用 i2s-fixchecksum 和 i2s 变异算法可以领先 baseline,这是由于输入-状态算法可以发现 AFL 确定性变异方法需要多次变异才能发现的路径,但输入-状态算法阶段在染色阶段(2~3h)需要额外执行更多的用例导致最终覆盖率低于 baseline,但这是测试吞吐低导致的, i2s-fixchecksum 算法拥有到达更深路径的潜力。对于 HostControl 项目, i2s-fixchecksum 变异方法相对于 i2s 和 baseline 大幅提高了代码覆盖率。

上述结果表明,校验和修补算法确实可以进入嵌套校验和内部探索新的路径,从真实项目实验结果来看,带校验和修补算法的输入-状态算法确实可以显著提高代码覆盖率,即使在最坏情况下也不明显低于基线,验证了校验和修补算法的有效性。

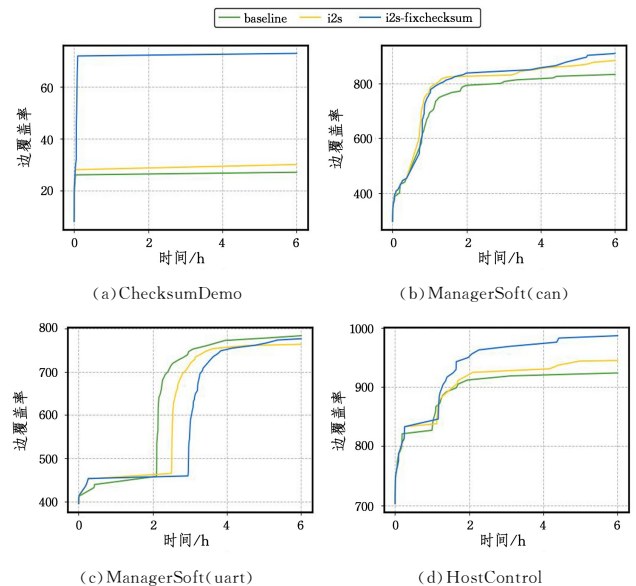


图 5 边覆盖率随时间变化图

Fig. 5 Edge coverage rate over time

4.4 发现漏洞的能力

为了验证 AFL-VTest 是否能发现真实项目中的漏洞, 对项目 1 和项目 2 进行长时间模糊测试。尽管这两个项目已经经过软件测试、代码审查、静态分析等方法测试, AFL-VTest 还是成功在这两个项目中发现了 12 个 unique crash, 经人工排查确认 3 处是先前未被发现的数组越界访问漏洞。

结束语 本文在分析航天嵌入式软件进行模糊测试的意义和困难的基础上, 提出了一套面向航天嵌入式软件的模糊测试框架 AFL-VTest。AFL-VTest 采用精简源码插桩方法在航天嵌入式软件上实现了最先进的覆盖率引导的模糊测试, 同时针对航天嵌入式软件校验和较多的问题, 提出了一种基于输入-状态算法的近似校验和修补算法。

为了测试精简源码插桩方法的有效性, 分别在多个测试程序上实现验证分析, 证实插桩方法的可用性和有效性。为了验证近似校验和修补算法的有效性, 在一个样例程序和两个真实项目上进行实验, 结果表明近似校验和修补算法可以显著提升代码覆盖率。最后为了验证 AFL-VTest 是否可以发现真实项目中的漏洞, 在两个真实航天嵌入式软件上对其进行了评估实验。实验结果表明, AFL-VTest 成功揭露了实际项目中潜藏的 3 个缺陷, 从而验证了所提方法在提升航天嵌入式软件安全性与可靠性方面的有效性与实用价值。

然而, 对于一个新的项目, AFL-VTest 需要开发测试人员重新配置环境, 以及编写编译脚本进行源码插桩, 耗费大量人力, 未来的工作将专注于提高整个模糊流程的自动化水平, 优化测试流程。

参 考 文 献

- [1] CHEN L Q, WU G F, JIANG J H. Static Analysis Technique for Aerospace Embedded Software[J]. Aerospace Contrd and Application, 2021, 47(2): 86-92.
- [2] WILLBOLDJ, SCHLOEGEL M, VÖGELE M, et al. Space odyssey: An experimental software security analysis of satellites [C]//2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023: 1-19.
- [3] ZUO W J, DONG Y, HUANG C, et al. Research on static testing method of aerospace embedded software [J]. Microelectronics & Computer, 2022, 39(5): 78-86.
- [4] ZUO W J, YU L K, WANG X L, et al. Typical Test Cases Design Faults Research of Aerospace Embedded Software[J]. Computer Measurement & Control, 2019, 27(10): 36-40.
- [5] ZUO W J, DONG Y, HUANG C, et al. Aerospace Embedded Software Code Logic Analysis[J]. Computer Systems & Applications, 2021, 30(8): 274-280.
- [6] ZUO W J, WANG X L, HUANG C, et al. Analysis and Practice of Implicit Requirement for Aerospace Embedded Software[J]. Measurement & Control Technology, 2023, 42(10): 24-29.
- [7] SEREBRYANY K. Oss-fuzz-google's continuous fuzzing service for open source software [EB/OL]. <https://github.com/google/oss-fuzz>.
- [8] YUN J, RUSTAMOV F, KIM J, et al. Fuzzing of embedded systems: A survey[J]. ACM Computing Surveys, 2022, 55(7): 1-33.
- [9] EISELE M, MAUGERI M, SHRIWAS R, et al. Embedded fuzzing: a review of challenges, tools, and solutions[J]. Cybersecurity, 2022, 5(1): 18.
- [10] SCHARNOWSKI T, BUCHMANN F, WÖRNER S, et al. A Case Study on Fuzzing Satellite Firmware[C]// Workshop on the Security of Space and Satellite Systems(SpaceSec). 2023.
- [11] SCHARNOWSKI T, BARS N, SCHLOEGEL M, et al. Firmware: Using precise MMIO modeling for effective firmware fuzzing[C]//31st USENIX Security Symposium(USENIX Security 22). 2022: 1239-1256.
- [12] ZALEWSKI M. AFL(American Fuzzy Lop)[EB/OL]. [2025-04-28]. <https://github.com/google/AFL>.
- [13] LLVM M. libfuzzer [EB/OL]. [2025-04-28]. <https://llvm.org/docs/Libfuzzer.html>.
- [14] Google. honggfuzz[EB/OL]. [2025-04-28]. <https://github.com/google/honggfuzz>.
- [15] FAN R, PAN J, HUANG S. ARM-AFL: coverage-guided fuzzing framework for ARM-based IoT devices[C]// International Conference on Applied Cryptography and Network Security. Cham: Springer, 2020: 239-254.
- [16] DU X, CHEN A, HE B, et al. Afllot: Fuzzing on linux-based IoT device with binary-level instrumentation[J]. Computers & Security, 2022, 122: 102889.
- [17] SHEN Y, XU Y, SUN H, et al. Tardis: Coverage-guided embedded operating system fuzzing[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022, 41(11): 4563-4574.
- [18] ZHANG C, LI Y, CHEN H, et al. Biff: Practical binary fuzzing framework for programs of iot and mobile devices[C]// 2021 36th IEEE/ACM International Conference on Automated Software Engineering(ASE). IEEE, 2021: 1161-1165.
- [19] QUYNHN A. Skorpio: Advanced binary instrumentation framework[EB/OL]. [2025-10-12]. <https://groundx.io/docs/Opcde2018-skorpio.pdf>.
- [20] EISELE M, EBERT D, HUTH C, et al. Fuzzing embedded systems using debug interfaces[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023: 1031-1042.
- [21] LI W, SHI J, LI F, et al. μ AFL: non-intrusive feedback-driven fuzzing for microcontroller firmware[C]// Proceedings of the 44th International Conference on Software Engineering. 2022: 1-12.
- [22] GAO Z, DONG W, CHANG R, et al. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware [J]. Concurrency and Computation: Practice and Experience, 2022, 34(16): e5756.
- [23] BECKMANN M, STEFFAN J. Coverage-Guided Fuzzing of Embedded Systems Leveraging Hardware Tracing[C]// European Symposium on Research in Computer Security. Cham: Springer, 2022: 362-378.
- [24] SPERL P, BÖTTINGER K. Side-channel aware fuzzing [C]//

- Computer Security-ESORICS 2019: 24th European Symposium on Research in Computer Security. Springer, 2019: 259-278.
- [25] 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.
- [26] CHEN J, DIAO W, ZHAO Q, et al. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing[C]// NDSS. 2018.
- [27] REDINI N, CONTINELLA A, DAS 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 (SP). IEEE, 2021: 484-500.
- [28] BELLARD F. QEMU: a fast and portable dynamic translator [C] // USENIX Annual Technical Conference, FREENIX Track. 2005.
- [29] ZHANG F, CUI B, CHEN C, et al. Simulation-Based Fuzzing for Smart IoT Devices[C]// Innovative Mobile and Internet Services in Ubiquitous Computing: Proceedings of the 15th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2021). Springer, 2022: 304-313.
- [30] KAMMERSTETTER M, PLATZER C, KASTNER W. Prospect: peripheral proxying supported embedded code testing [C]// Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security. 2014: 329-340.
- [31] ZHENG Y, DAVANIAN A, YIN H, et al. FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation [C] // 28th USENIX Security Symposium (USENIX Security 19). 2019: 1099-1114.
- [32] KIM J, YU J, KIM H, et al. FIRM-COV: high-coverage greybox fuzzing for IoT firmware via optimized process emulation[J]. IEEE Access, 2021, 9: 101627-101642.
- [33] ZHENG Y, LI Y, ZHANG C, et al. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation[C]// Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022: 417-428.
- [34] FENG B, MERA A, LU L. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling[C] // 29th USENIX Security Symposium (USENIX Security 20). 2020: 1237-1254.
- [35] MERA A, FENG B, LU L, et al. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis[C]// 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021: 1938-1954.
- [36] WANG C, LIANG H. Value Peripheral Register Values for Fuzzing MCU Firmware[C] // 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2023: 718-729.
- [37] ZHOU W, GUAN L, LIU P, et al. Automatic firmware emulation through invalidity-guided knowledge inference[C] // 30th USENIX Security Symposium (USENIX Security 21). 2021: 2007-2024.
- [38] CLEMENTSA A, GUSTAFSON E, SCHARNOWSKI T, et al. HALucinator: Firmware re-hosting through abstraction layer emulation[C] // 29th USENIX Security Symposium (USENIX Security 20). 2020: 1201-1218.
- [39] GUI Z, SHU H, YANG J. Firmnano: Toward iot firmware fuzzing through augmented virtual execution[C] // 2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS). IEEE, 2020: 290-294.
- [40] FARRELLY G, CHESSER M, RANASINGHE D C. Ember-IO: effective firmware fuzzing with model-free memory mapped IO [C] // Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security. 2023: 401-414.
- [41] FARRELLY G, QUIRK P, KANHERE S S, et al. SPlITS: Split Input-to-State Mapping for Effective Firmware Fuzzing [C] // European Symposium on Research in Computer Security. Cham: Springer, 2023: 290-310.
- [42] SCHARNOWSKI T, WÖRNER S, BUCHMANN F, et al. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs [C] // Proceedings of the 32nd USENIX Conference on Security Symposium. USENIX Association, 2023: 2885-2902.
- [43] Sunwiseinfo. VTest[EB/OL]. [2025-04-28]. <http://www.sunwiseinfo.com.cn/vtest>.
- [44] ASCHERMANN C, SCHUMILO S, BLAZYTKO T, et al. REDQUEEN: Fuzzing with Input-to-State Correspondence [C] // NDSS. 2019: 1-15.
- [45] NETHERCOTE N, SEWARD J. Valgrind: A program supervision framework[J]. Electronic Notes in Theoretical Computer Science, 2003, 89(2): 44-66.
- [46] SEREBRYANY K, BRUENING D, POTAPENKO A, et al. AddressSanitizer: A fast address sanity checker[C] // 2012 USENIX Annual Technical Conference (USENIX ATC 12). 2012: 309-318.
- [47] FIORALDI A, MAIER D, EIBFELDT H, et al. AFL++: Combining incremental steps of fuzzing research[C] // 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.
- [48] GOOGLE PROJECTZERO. WinAFL[EB/OL]. <https://github.com/googleprojectzero/win afl>.



WANG Shuai, born in 1997, postgraduate. His main research interest is fuzzing test.



XIAO Xi, born in 1979, Ph.D, associate professor. His main research interests include AI security and network security.