北京航空航天大學

BEIHANG UNIVERSITY

第三十四届“冯如杯”竞赛主赛道项目论文

Latex 模板

——基于 Latex 的论文模板

摘要

商用机器人、自动驾驶等领域的快速发展，带动了相关软件工具的进步。其中，机器人操作系统（ROS2），以其开源、跨平台、高效的软件框架及丰富的工具库而深受开发者喜爱，在学界、商业应用中得到广泛应用。与此同时，机器人操作系统也存在开源软件共有的安全漏洞多、不稳定等问题，制约着其进一步发展，也威胁着相关应用的安全。

机器人操作系统相关程序亟需一种高效的测试方法，面对这一场景，我们选择将自动模糊测试技术（Fuzzing）引入对机器人操作系统的测试。通过随机变异测试输入，并加入反馈引导，模糊测试不仅提高了测试的自动化程度，还能有效地探索软件异常路径，揭露隐蔽漏洞，显著改善了传统软件测试方法的繁琐性和低效性。

为了提高测试效率，我们的模糊测试对机器人系统进行针对性优化。针对通信网络环境的不稳定性，本框架选择将机器人收到的通信消息等多维度输入进行随机化变异；针对机器人多进程多线程的软件特点，本框架通过程序插桩延迟来提高数据竞争概率；针对机器人程序弱状态特点，本框架通过流量分析等黑盒手段，来辅助推测程序内部状态。

本作品设计的测试框架短期内共发掘 20 个 ROS2 相关软件的内存漏洞、并发漏洞，其中 11 个得到开发者确认修复，其中有 4 个重要漏洞被美国通用漏洞披露（CVE）收录，7 个通过中国国家漏洞库（CNVD）二级审核，正在等待三级审核。

综上，本作品——针对机器人系统的模糊测试框架在软件测试中有较高实践价值，已为开源社区贡献多次漏洞修复，并可高效地移植用于测试其他机器人程序。

关键词：自动模糊测试技术，机器人操作系统，并发漏洞检测，内存安全漏洞检测

Abstract

The swift advancement in commercial robotics and autonomous driving has spurred the development of associated software tools, notably the Robot Operating System (ROS2). Celebrated for its open-source, cross-platform capabilities, and extensive toolkit, ROS2 has garnered widespread adoption across academia and industry. However, its growth is hampered by common open-source software issues like security vulnerabilities and instability, posing risks to application safety.

Addressing the need for efficient testing in **ROS2-related programs**, we introduced automatic **fuzz** testing to enhance software reliability. By employing randomized mutations of test inputs and feedback guidance, fuzz testing significantly elevates automation and efficacy in exploiting vulnerabilities, outpacing traditional testing in identifing uncovering software exception.

Our framework, tailored for robotic systems, addresses communication network instability by randomizing inputs like received communication messages. It targets software's multi-process and multi-threaded nature by increasing data race probabilities via program instrumentation delays. Additionally, to navigate the challenges posed by robots' weak states, our framework utilizes black-box methods, such as traffic analysis, to infer internal states. The approach led to **the discovery of 20 ROS2 software vulnerabilities, with 11 confirmed and fixed by developers, including 4 recognized by the US's CVE and 7 awaiting final review by China's CNVD.**

This work demonstrates the high practical value of our fuzz testing framework for robotic systems, contributing significantly to the open-source community and efficiently adaptable for testing various robot programs.

Keywords: Fuzz, ROS2, Concurrent Vulnerability, Memory Vulnerability

目录

一、作品概述

（一）背景介绍

1.ROS 系统重要性

人类生活与机器人密不可分。传统上，机器人在农业和制造业中被广泛用于任务自动化。最近的发展让机器人更加贴近每个人的日常生活。例如，亚马逊和谷歌正在部署无人配送系统 [22,23]，使用无人驾驶飞行器，机器人吸尘器的市场规模年增长率达到 23%[24]，以及在 2012 年到 2018 年间，使用机器人进行手术的比例从 2% 增加到 15%[25]，显示出机器人产业为适应人类需求而快速增长的态势。同时，现代机器人，如自动驾驶车辆，正在变得更加复杂，要求集成复杂的子系统，例如感知、感觉、规划和执行。

为了应对开发复杂机器人系统的更高需求，机器人操作系统（ROS）[1] 正在获得越来越多的关注。ROS 是一个开源的中间件套件，用于机器人开发，它具有分布式机器人进程的消息传递机制、硬件抽象、广泛的开发工具（例如，模拟器）和机器人库（例如，路径规划算法）。使用 ROS，开发者可以加速开发过程，无需重新发明轮子，而是可以专注于他们机器人的核心功能。凭借其多语言和多平台支持的理念，ROS 正在成为机器人编程中的实际标准；它已被广泛采用于工业 [26,27]、军事 [28,29]、研究机构以及个人，预计到 2024 年将为 55% 的商业机器人提供动力 [30]。近些年 ROS 系统相关论文数量与 Wiki 用户数量如图 1 所示：

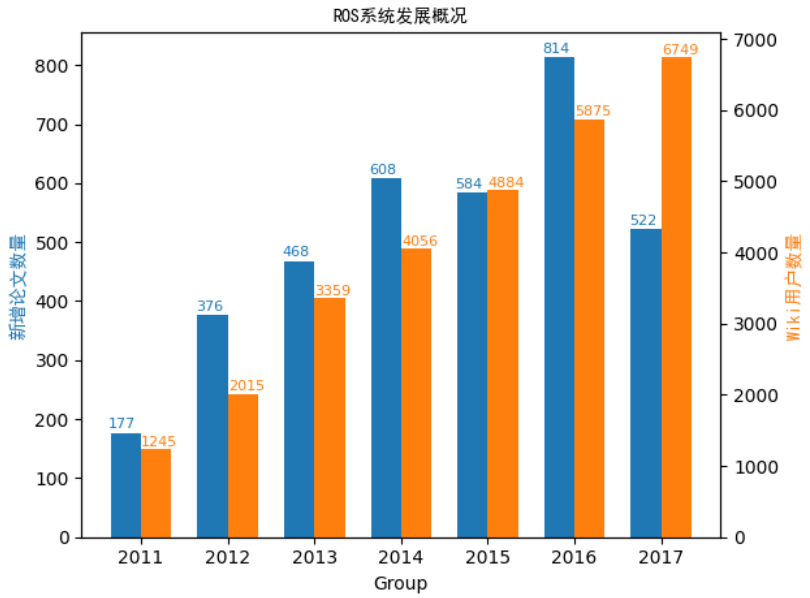

图 1 ROS 系统相关论文数量与 Wiki 用户数量

机器人操作系统（ROS）是一个开源平台，它为用 C++ 和 Python 等不同语言实现机器人软件提供了许多实用的工具和库。超过 740 家商业公司正在使用 ROS[2] 来开发数百种实际应用的机器人 [3]，这表明了其在机器人开发中的流行度。然而，开发可靠的 ROS 程序具有挑战性，因为机器人可以在复杂的物理环境中工作，并可能遇到各种异常情况（如无效的配置参数和异常的传感器信息）。如果 ROS 程序不够可靠，它们的错误可能在与人类和其他机器人交互时导致危险事故。

ROS 本身的缺陷或者开发者在应用中常用的 ROS 使用方式可能会影响依赖于 ROS 的广泛机器人系统，并对许多用户的安全和保障造成严重破坏。例如，ROS 版本 1 没有认证的概念，允许网络上的任何实体完全访问任何机器人系统，窃听内部消息，甚至在知道 IP 地址和端口的情况下劫持执行。实际上，在互联网上扫描默认的 ROS 端口两个月后 [31]，发现部署在 28 个国家的超过 100 个 ROS 系统，这些系统完全暴露于此类攻击之下。ROS 的最新版本（即 ROS 2）在设计和实现时考虑了这些问题，并邀请了更广泛的公众对安全性进行审查。不幸的是，现有的工作和解决方案要么专注于关于认证和授权方法的网络安全方面 [32,33,34,35]，要么专注于回归测试 [36]，使得 ROS 社区在寻找影响机器人系统的健壮性和正确性的未知错误方面缺乏一种系统的测试方法。

2. 自动驾驶兴起与 ROS

自动驾驶技术的兴起标志着交通运输领域的一次重大革命，其不仅预示着交通安全、效率和环境影响的根本改进，而且还预示着个人出行和货物运输方式的深刻变革，而这一过程在很大程度上得益于机器人操作系统（Robot Operating System，ROS）的发展和应用。近年来，由于计算能力的显著提升、大数据技术的发展、人工智能算法的进步以及传感器技术的优化，自动驾驶汽车从理论研究和小规模试验逐步走向了公路测试和商业化应用的初步阶段。

根据国际汽车工程师学会 (SAE) 的定义 [1]，自动驾驶分为六个级别，从 0 级（无自动化）到 5 级（完全自动化）。目前，多数公开测试和部分商业化应用的自动驾驶汽车处于 3 级（有条件自动化）到 4 级（高度自动化）。尽管完全自动化（5 级）的汽车尚未广泛部署，但多个技术开发者和制造商已经在进行相关的研发工作，并在不同国家和地区开展了路试。

随着自动驾驶技术的进步，对处理大量传感器数据、实现复杂决策逻辑和执行精确控制的需求不断增加，ROS 的可扩展性和灵活性在此过程中显示出其不可或缺的价值。例如，ROS 的消息传递系统支持多种编程语言，允许异构系统高效集成。此外，其庞大的开源社区贡献了大量的软件包，覆盖从 3D 视觉到路径规划等各种功能，极大地丰富了自动驾驶汽车的研发资源库。

经济学人智库的一份报告预测 [2]，到 2035 年，全球自动驾驶汽车的数量将达到近 7500 万辆。而麦肯锡公司的一项研究则估计 [3]，自动驾驶技术的全面部署将使交通事故造成的死亡人数减少 90%，同时还将大幅度提升道路运输效率和减少碳排放。图 2 展示了 PRECEDENCE RESEARCH 机构预测的未来自动驾驶市场规模：

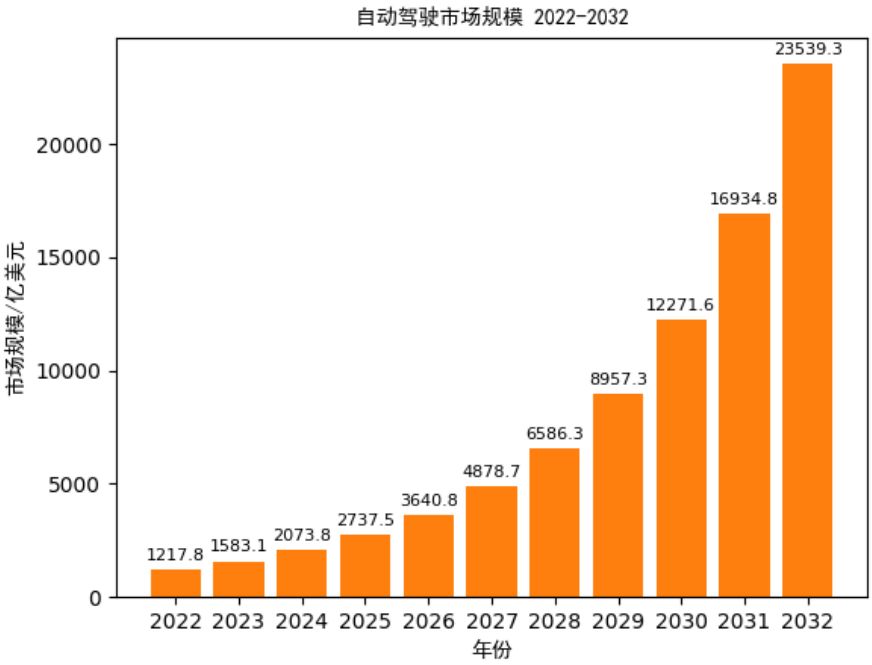

图 2 自动驾驶市场规模

尽管自动驾驶技术的前景被广泛看好，但其商业化应用和普及还面临多重挑战，包括技术完善度、法律法规、道德伦理、消费者接受度以及基础设施配套等。例如，自动驾驶汽车在复杂的城市交通环境中的表现，以及在极端天气条件下的可靠性，仍然是技术研发中的重要挑战。

未来，随着相关技术的进一步成熟和相关政策的完善，自动驾驶汽车有望在更广泛的领域得到应用，从而实现对交通系统的根本性改造和优化。这将不仅影响交通运输本身，还将对城市规划、能源消费、环境保护以及人们的生活方式产生深远影响。

3. 自动驾驶产生漏洞的危害

尽管自动驾驶汽车（AVs）承诺通过减少人为错误来增加道路安全，它们固有的技术漏洞却可能导致严重的安全后果。据国家公路交通安全管理局（NHTSA）估计，94% 的严重交通事故是由人为因素引起的，而自动驾驶技术的开发者们宣称，通过消除这一因素，可以极大地提高道路安全性 [4]。然而，这种安全性的提升假设基于自动驾驶系统能够无缺陷地执行其预定任务，这在实践中往往难以实现。

自动驾驶汽车的技术漏洞主要分为两类：软件漏洞和硬件故障。软件漏洞包括但不限于算法错误、安全漏洞以及对异常情况的处理不足。硬件故障可能涉及传感器故障、执行机构的故障或其他关键组件的损坏。这些技术漏洞不仅可能导致单一车辆的操作失败，还可能对整个交通系统产生连锁反应，特别是在高度自动化和互联的交通环境中。

例如，2018 年 3 月，在美国亚利桑那州发生了一起致命的自动驾驶汽车事故，一名行人在穿越街道时被一辆自动模式下的 Uber 汽车撞击，导致死亡。初步调查显示，该事故部分原因是系统无法正确识别并响应行人 [5]。这一事件凸显了即使在自动驾驶技术取得显著进步的情况下，技术漏洞仍然可能导致灾难性后果。

此外，安全研究人员已经证明，自动驾驶汽车系统的安全漏洞可以被黑客利用，进而控制车辆的行驶方向或速度，造成严重安全威胁 [6]。例如，通过篡改车辆的环境感知系统接收到的数据，攻击者可以使汽车忽略停车标志或其他关键交通信号，导致交通事故。

综上所述，尽管自动驾驶技术和机器人技术被寄予厚望，但频繁出现的技术漏洞和安全威胁，表明相关技术的安全可靠性还需要进一步提高。因此，对相关程序和基础 ROS2 系统的进一步安全测试、漏洞排除仍然任重道远，亟需新技术来高效、无痛地排除漏洞威胁，降低技术进步的血泪代价。

（二）研究现状

1.ROS 常见测试方案

近期有几种 [2,6,7,8,9,10] 应用于 ROS2 程序的新型测试方法，它们不同于由开发者手动构造测试样例的单元测试，而尝试自动生成 ROS 程序输入，并借助清洁工具（Sanitizer）自动监控程序内存或语义错误。它们大幅提高了针对机器人系统的测试效率，表1将这些方法进行了比较（包括本作品），可以看出这些方法在测试 ROS 程序时仍存在三个主要限制：

限制一：测试用例生成效率低。大部分测试方法从单一维度（即用户命令或传感器信息）生成输入，并且生成的测试样例随机性过强，无法识别出可能高效探索程序边界的变异维度，导致许多生成的输入无助于增加测试覆盖率。

限制二：程序反馈效果不佳。SMACH-Fuzz 和 ASTAA 随机生成输入，而没有反馈和指导；Ros2-fuzz 和 ROZZ 参考 AFL 工具 [11] 使用代码覆盖率作为程序反馈，但代码覆盖率对多进程运行情况不敏感，且忽略了不同执行路径；Phys-Fuzz 使用场景危险程度评分来指导进一步生成场景，对于检测 ROS2 自动导航程序的避障能力效果较好，但也仅限制在了此场景下；RoboFuzz 使用语义反馈来量化机器人执行上下文的正确性，但是这种特殊语义的设计编写需要很强专业知识。

限制三：自动化水平低。大部分方法都需要大量领域特定知识和手动努力来配置输入生成规则（ASTAA 和 RoboFuzz），编写有关机器人行为的规格（SMACH-Fuzz 和 RoboFuzz），检查执行日志（SMACH-Fuzz 和 Phy-Fuzz）等，这导致测试效率和便捷程度对比传统单元测试并没有很大提高。

表 1 最新针对 ROS 系统的测试技术对比

技术名	输入维度	反馈	自动化程度	漏洞类型
SMACH-Fuzz [6]	单一	无	差	行为错误
Phys-Fuzz [7]	单一	环境危险程度	差	碰撞损坏
Ros2-fuzz [10]	单一	代码覆盖率	较好	内存漏洞
ASTAA [8]	单一	无	差	行为错误
RoboFuzz [9]	单一	语义反馈	较差	行为错误
ROZZ [2]	多维度	代码覆盖率	较好	内存漏洞
本作品	多维度	代码覆盖率 + 流量分析	较好	内存与并发漏洞

2. 模糊测试及其应用

模糊测试是近年来最受欢迎的软件测试技术之一，通过向系统输入异常、随机或无效的数据来暴露软件中的缺陷、漏洞和崩溃。模糊测试在不需要了解应用程序内部逻辑的情况下，检测软件如何处理意外或错误的输入，非常适合寻找安全漏洞，如缓冲区溢出、执行流程控制错误、内存泄露等。同时，利用模糊测试发现了许多高影响力的安全漏洞（如 Heartbleed、Shellshock 等），进一步证明了其有效性和重要性。

尽管模糊测试在软件安全领域得到了广泛应用，但在 ROS 环境中的应用相对较少。随着人工智能术的发展和 ROS 在人工智能如无人驾驶等研究及应用中的重要性增加，对于 ROS 环境的模糊测试方法的需求和研究也在逐渐增长。开发专门针对 ROS 系统特点的模糊测试工具和技术，可以有效提高机器人软件的可靠性和安全性。

（三）作品概述

本作品给出针对 ROS2 系统的自动模糊测试程序，其核心框架如图3所示，旨在提高对 ROS2 系统相关的自动驾驶项目、机器人项目的漏洞测试的便捷性和高效性，为开发者提供一个有效测试工具。开发者首先将源码搭配本作品的编译器工具编译为可执行程序，此过程中本作品将自动使用 LLVM 架构对源码进行插桩修改，并链接相应清洁程序（如 Address Sanitizer 等第三方监控工具）；接着，本作品核心模糊测试驱动器会启动特殊 ROS2 执行环境执行开发者的 ROS2 程序，并将开发者提供的初始测试样例进行变异修改，生成一个测试用例输入进程序；本作品的监控器（Node Monitor）会对

被测程序的代码覆盖率、程序异常、通信流量、服务状态等信息进行收集分析；分析结果被作为该测试用例种群适应度输入到遗传算法中，遗传算法会对种群（测试用例池）进行进化和淘汰，选择出更有可能触发程序边界的测试用例，并指导对测试用例种群进行进一步变异。

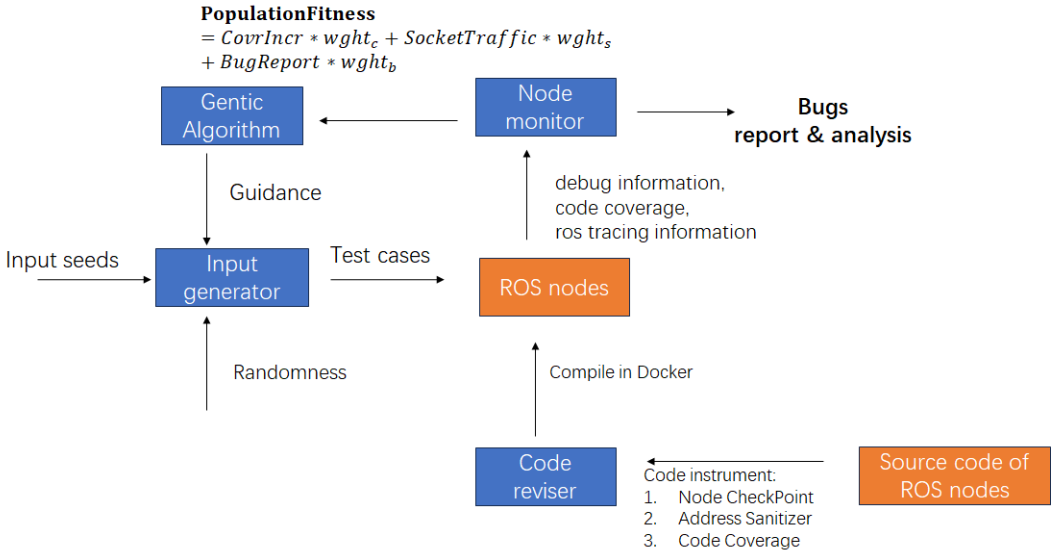

图 3 本项目实现的 Fuzz 框架

通过上述步骤，本作品会不断生成被测 ROS2 节点程序的高质量测试用例，覆盖更多节点程序的内部状态分支，以更高效率探索程序潜在漏洞。本作品的创新之处主要有几点：针对 ROS2 系统程序特点，不局限于传统测试输入，探索多维度的程序输入；以更高效的办法对 ROS2 节点程序进行监控，综合白盒插桩检测与黑盒流量分析等技术；支持检测多种漏洞类型，如内存安全漏洞与并发漏洞等。

二、作品设计与实现

（一）技术背景与预备知识

1. 漏洞种类及危害

释放后使用（UAF）：释放后使用问题发生在程序试图使用已释放的内存时。通常，内存释放后，操作系统会将该内存标记为可重新使用，但并不会立即清空其内容。如果程序在内存释放后继续访问该内存，就可能导致安全漏洞。该漏洞的介绍如图 1 所示。攻击者可能利用释放后使用漏洞来执行未经授权的代码，例如通过释放内存但保留对其的引用，然后在后续代码中使用该引用，从而导致恶意代码执行。同时可能导致程序崩溃或不稳定，因为操作已释放的内存区域可能导致未定义的行为。

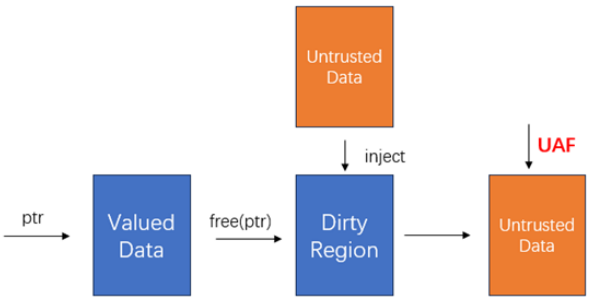

图 4 释放后使用漏洞图示

Heartbleed 漏洞是一个广为人知的释放后使用漏洞的例子。该漏洞影响了 OpenSSL 库中的 Heartbeat 扩展，攻击者可以发送恶意的 Heartbeat 请求，从而导致服务器上的内存泄漏和可能的敏感信息泄露。

异常（Exception）：

异常是一种用于处理错误或不正常情况的机制。在程序执行期间，如果发生异常，通常会中断当前执行流程，并转移到异常处理代码。如果程序未正确处理异常，可能会导致安全漏洞。

异常可能导致程序跳转到未经测试的代码路径，使得程序执行流程不可预测，从而导致意外行为或安全漏洞。同时，异常通常包含程序执行的上下文信息，攻击者可能利用这些信息来进一步渗透系统，进而导致信息泄露。

在 2014 年的 “Shellshock” 漏洞中，攻击者利用了 Unix/Linux 系统中的一个 bash shell 的异常处理漏洞。通过在 HTTP 请求的 User-Agent 头中注入恶意代码，攻击者能够利用 bash 的异常处理漏洞来执行任意命令，从而导致系统被入侵。

缓冲区溢出（Buffer overflow）：

缓冲区溢出发生在程序试图向一个缓冲区写入超过其分配大小的数据时。这导致数据溢出到相邻的内存区域，覆盖了那些数据或程序代码。通常，这种溢出可以修改程序的执行流程，因为溢出数据可能包含特定的指令地址，攻击者可以利用这一点来控制程序的行为。该漏洞的介绍如图 2 所示。

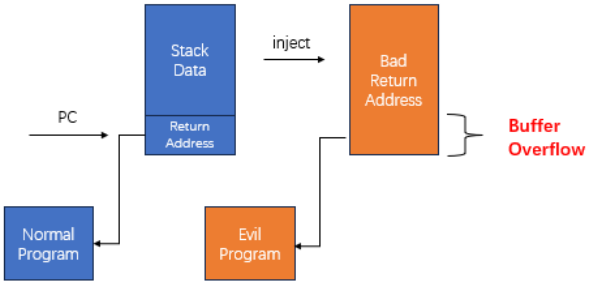

图 5 缓冲区溢出漏洞图示

攻击者可能利用缓冲区溢出漏洞来执行恶意代码，例如注入 Shellcode 并强制程序跳转到 Shellcode 的地址，从而获得系统权限。同时，也可能泄露敏感信息，例如通过溢出将重要的内存区域覆盖为攻击者所控制的数据，从而导致信息泄露。除此之外，因为缓冲区溢出可能导致程序崩溃或无法正常执行，使其无法提供正常的服务。

著名的 “Code Red” 蠕虫利用了 Microsoft IIS 服务器上的缓冲区溢出漏洞。攻击者通过发送特制的 HTTP 请求，导致 IIS 的缓冲区溢出，并在受感染的系统上运行恶意代码。这导致系统被感染并且在网络上传播蠕虫。

2. 模糊测试框架

模糊测试（fuzzing）作为一种基于缺陷注入的自动化软件漏洞挖掘技术，是现在最有效的漏洞挖掘技术。模糊测试向目标应用程序生成大量的正常和异常输入，并通过将生成的输入提供给目标应用程序并监视执行状态来检测异常。与其他技术相比，模糊化易于部署，具有良好的可扩展性和适用性，无论是否使用源代码，都可以执行。同时，由于模糊测试在代码运行时进行监测，所以其具有较高的精度。

具体而言，模糊测试的工作过程包括四个主要阶段：测试样例生成阶段、测试样例运行阶段、程序执行状态监控和异常分析阶段。模糊测试主要过程如图 X 所示。

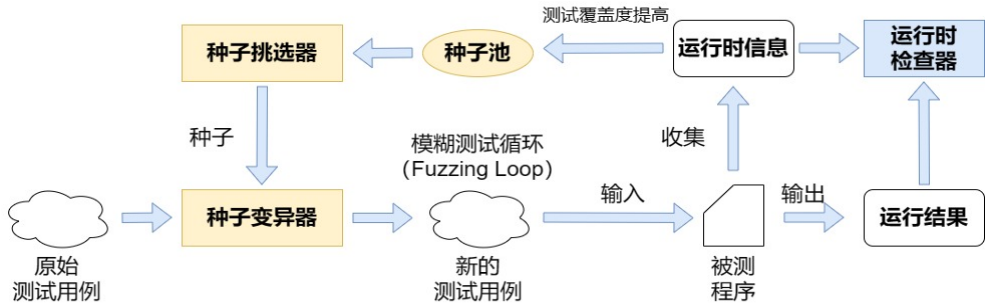

图 6 Example of an image

模糊测试从生成测试用例开始, 其生成的测试用例的质量直接影响模糊测试效果。其测试样例一方面应满足测试程序对输入格式的要求，另一方面输入数据需要包含足够的异常数据，以便这些输入在被程序处理时很可能导致程序失败或崩溃。根据目标程序的不同，输入可以是具有不同文件格式的文件、网络通信数据、具有特定特征的可执行二进制文件等。如何生成具有足够“破坏性”的测试用例是模糊器面临的一个主要挑战。一般来说，在现有的模糊器中使用两种生成器，即基于生成的生成器和基于突变的生成器。

在前一阶段生成测试样例后，这些测试样例会被送入目标程序。模糊测试工具会自动启动并结束目标程序的进程，并驱动目标程序的测试样例处理过程。在执行之前，分析人员可以配置目标程序的启动和结束方式，并预定义参数和环境变量。通常，模糊测

试过程会在预定义的超时时停止，或是在程序执行挂起或崩溃时停止。

模糊测试工具在目标程序执行过程中监控执行情况，期待发现异常和崩溃。常用的异常监控方法包括对特定系统信号、崩溃以及其他违规行为的监控。对于没有直观程序异常行为的违规情况，可以使用许多工具，包括 AddressSanitizer（Serebryany 等，2012 年）、DataFlowsanitizer（The Clang Team，2017a）、ThreadSanitizer（Serebryany 和 Iskhodzhanov，2009 年）、LeakSanitizer（The Clang Team，2017b）等。当捕获到违规行为时，模糊测试工具会存储相应的测试样例，以供后续重放和分析。在分析阶段，分析人员尝试确定捕获的违规行为的位置和根本原因。分析过程通常借助调试器完成，比如 GDB、windbg，或其他二进制分析工具，如 IDA Pro、OllyDbg 等。二进制插桩工具，如 Pin 等，也可以用来监控收集到的测试样例的确切执行状态，例如线程信息、指令、寄存器信息等。自动崩溃分析是另一个重要的研究领域。

在执行完上述四个阶段之后，将根据程序反馈即代码覆盖率是否提升来判断种子是否有趣，有趣的种子将被加入种子池中进行后续变异，进而形成新的测试样例重新输入待测程序。具体而言，判断种子是否有趣的原则为选取开销小，作用大的种子，即如果两个种子可以得到相同的代码逻辑覆盖情况，则保留文件较小的种子；如果两个种子可以得到相同的代码逻辑覆盖情况，则选择耗时较小的。

根据程序源代码的依赖程度和程序分析的程度，模糊测试工具可以分为白盒、灰盒和黑盒。白盒模糊测试假定能够访问程序的源代码，因此可以通过对源代码的分析以及测试用例如何影响程序运行状态来收集更多信息。黑盒模糊测试在没有任何关于目标程序内部的知识的情况下进行模糊测试。灰盒模糊测试也不需要源代码，通过程序分析来获得目标程序的内部信息。

具体而言，白盒模糊测试是一种利用程序的内部结构和实现细节来生成测试用例的测试方法。不同于黑盒模糊测试仅基于输入和输出进行测试，白盒模糊测试要求测试者对程序的内部结构有深入了解。通过分析程序的源代码，白盒模糊测试使用符号执行和约束求解技术来自动生成能够覆盖程序不同路径的输入数据，目标是深入程序内部逻辑以发现深层次的安全漏洞和错误。

白盒模糊测试的一个核心特点是它的能力来处理高度结构化的输入，如编译器和解释器等应用程序。这些应用程序按阶段处理输入，例如词法分析、解析和求值。白盒模糊测试结合了静态分析和动态测试生成的方法，通过执行程序的符号执行并在此过程中生成和求解约束，以自动化的方式生成测试输入。

白盒模糊测试在自动化测试生成、测试用例的执行以及测试结果的分析方面的广泛应用。这包括但不限于利用程序分析技术优化测试过程、缩小输入空间、提高测试的自

动化水平等。白盒模糊测试通过这些技术手段实现了对复杂软件系统的有效测试，能够更加准确地发现并定位软件中的安全漏洞和错误。

3.ASan

[5] 在模糊测试阶段自动分析运行时信息和输出结果时，程序会进行缺陷检测，而ASan（即 AddressSanitizer）是一种专为 C 和 C++ 等编程语言设计的快速的内存错误检测工具，可以有效地识别内存访问错误，包括堆、栈和全局变量的越界访问以及使用已释放堆内存（use-after-free）的错误。具体而言，它采用了一种特殊的内存分配器和足够简单到可以在任何编译器、二进制转换系统甚至在硬件中实现的代码插桩技术。ASan 在不牺牲全面性的情况下实现了高效率，其平均运行速度减慢仅为 73%，但它能够准确地在发生错误时立即检测到错误。

ASan 由两部分组成：一个插桩模块和一个运行时库。插桩模块修改代码以检查每次内存访问的影子状态，并在栈和全局对象周围创建有毒的红色区域以检测溢出和下溢。当前的实现基于 LLVM 编译器基础设施。运行时库替换了 malloc、free 及相关函数，围绕分配的堆区域创建有毒的红色区域，延迟已释放的堆区域的重用，并进行错误报告。

从高层次上看，ASan 对内存错误检测的方法与基于 Valgrind 的工具 AddrCheck 类似：使用影子内存记录每个应用程序内存字节是否安全访问，并使用插桩来在每个应用程序加载或存储时检查影子内存。与之不同的是，ASan 使用了更高效的影子映射、更紧凑的影子编码，在检测堆外还能检测栈和全局变量中的错误，并且比 AddrCheck 快一个数量级。接下来将描述 ASan 如何编码和映射其影子内存、插入其插桩，以及其运行时库如何操作。

（一）影子内存

malloc 函数返回的内存地址通常至少对齐到 8 字节。这导致任何对齐的 8 字节序列的应用程序堆内存都处于 9 种不同状态之一：前 k 个（$0 \leq k \leq 8$）字节是可寻址的，而剩下的 $8 - k$ 字节则不是。这种状态可以编码成一个影子内存的单字节。

ASan 将虚拟地址空间的八分之一专用于其影子内存，并使用一个比例和偏移的直接映射方式来将应用程序地址翻译成相应的影子地址。给定应用程序内存地址 Addr，影子字节的地址计算为 (Addr>>3)+Offset。如果 $Max - 1$ 是虚拟地址空间中的最大有效地址，那么 Offset 的值应该选择为从 Offset 到 $Offset + Max/8$ 的区域在启动时不被占用。在典型的 32 位 Linux 或 MacOS 系统上，其中虚拟地址空间为 0x00000000−0xffffffff，ASan 使用 Offset = 0x20000000 (2 的 29 次方)。在一个具有 47 个有效地址位的 64 位系统上，我们使用 Offset = 0x0000100000000000 (2 的 44 次方)。在某些情况下（例如，在

Linux 上使用-fPIE/-pie 编译器标志），可以使用零偏移来进一步简化插桩。

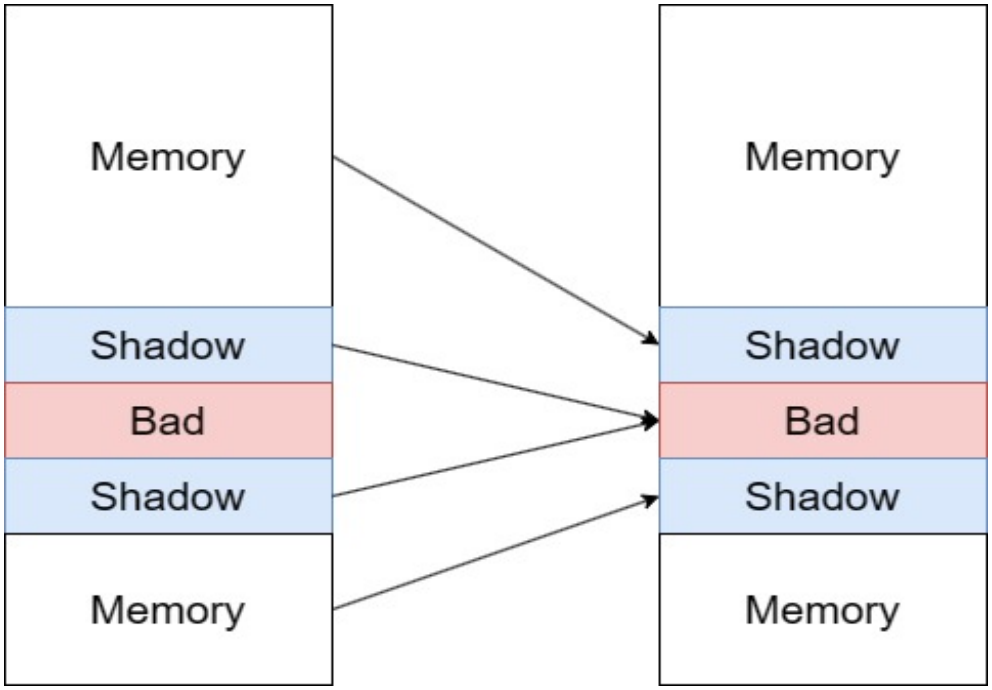

图 7 影子内存地址布局

图 X 展示了地址空间布局。应用程序内存被分为两部分（低和高），它们映射到相应的影子区域。将影子映射应用于影子区域中的地址会给我们在“坏”区域的地址，该区域通过页面保护标记为不可访问。

我们对每个影子字节使用以下编码：0 意味着相应的应用程序内存区域的所有 8 字节都是可寻址的；k（$1 \leq k \leq 7$）意味着前 k 个字节是可寻址的；任何负值表示整个 8 字节字是不可寻址的。我们使用不同的负值来区分不同种类的不可寻址内存（堆红区、栈红区、全局红区、释放的内存）。这种影子映射可以概括为形式 (Addr>>Scale)+Offset，其中 Scale 是 1...7 之一。当 Scale=N 时，影子内存占据虚拟地址空间的 $1/2^N$，红区（以及 malloc 对齐）的最小尺寸是 2^N 字节。每个影子字节描述了 2^N 字节的状态，并编码 $2^N + 1$ 不同的值。更大的 Scale 值需要较少的影子内存但更大的红区来满足对齐要求。Scale 值大于 3 需要对 8 字节访问进行更复杂的插桩，但为可能无法放弃其地址空间连续八分之一的应用程序提供了更多的灵活性。

（二）插桩模块

当对一个 8 字节的内存访问进行插桩时，AddressSanitizer 计算相应影子字节的地址，加载该字节，并检查它是否为零：

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
ReportAndCrash(Addr);
```

当对 1 字节、2 字节或 4 字节的访问进行插桩时，插桩过程会稍微复杂：如果影子值是正数（即，8 字节词中的前 k 个字节是可寻址的），我们需要将地址的最后 3 位与 k 进行比较。

```
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
  ReportAndCrash(Addr);
```

在这两种情况下，插桩仅在原始代码的每次内存访问中插入一个内存读取。我们假设一个 N 字节的访问是对齐到 N 的。ASan 可能会错过由于未对齐访问引起的错误。

ASan 的插桩阶段处在 LLVM 优化管道的最末端。这样我们只对那些经过 LLVM 优化器的所有标量和循环优化后仍然存在的内存访问进行插桩。例如，通过 LLVM 优化掉的本地栈对象的内存访问将不会被插桩。同时，我们不需要对 LLVM 代码生成器生成的内存访问进行插桩（例如，寄存器溢出）。

（三）运行库

运行时库的主要目的是管理影子内存。在应用程序启动时，整个影子区域都会被映射，以便程序的其他部分无法使用它。影子内存的坏段被保护起来。在 Linux 上，影子区域在启动时总是未被占用的，因此内存映射总是成功的。在 MacOS 上，我们需要禁用地址空间布局随机化（ASLR）。我们的初步实验表明，相同的影子内存布局也适用于 Windows。

malloc 和 free 函数被替换为特殊的实现。malloc 函数在返回的区域周围分配额外的内存，即红色区域（redzone）。红色区域被标记为不可寻址或“有毒”的。红色区域越大，将能检测到的溢出或下溢也越大。

分配器内的内存区域被组织为一个自由列表数组，对应于一系列对象大小。当与请求的对象大小相对应的自由列表为空时，将从操作系统（例如，使用 mmap）分配一大组带有红色区域的内存区域。对于 n 个区域，我们分配 n + 1 个红区，使得一个区域的右红区通常是另一个区域的左红区：

图 8 红区分布

左红区域用于存储分配器的内部数据（如分配大小、线程 ID 等）；因此，堆红区的最小尺寸当前为 32 字节。这些内部数据不会因为缓冲区下溢而被破坏，因为这种下溢会在实际存储之前立即被检测到（如果下溢发生在被插桩的代码中）。

free 函数将整个内存区域标记为有毒，并将其放入隔离区，以便这个区域在短期内

不会被 malloc 重新分配。目前，隔离区被实现为一个 FIFO 队列，随时持有固定数量的内存。

默认情况下，malloc 和 free 记录当前的调用堆栈，以提供更多信息的错误报告。malloc 调用堆栈存储在左红区域（红区越大，可以存储的帧数就越多），而 free 调用堆栈存储在内存区域的开始处。

为了检测对全局变量和栈对象的越界访问，ASan 必须在这些对象周围创建有毒的红区。对于全局变量，红区在编译时创建，并且红区的地址在应用程序启动时传递给运行时库。运行时库函数将红区标记为有毒，并记录地址以供进一步的错误报告。

对于栈对象，红区在运行时创建并标记为有毒。目前，使用了 32 字节的红区（加上最多 31 字节用于对齐）。例如，给定一个程序：

```
1  void foo() {
2    char a[10];
3    <function body>
4  }
```

转换后的代码将类似于：

为了检测对栈对象的越界访问，AddressSanitizer 对原有的函数 foo 进行了转换，增加了红区（redzones）来围绕数组 arr。这些红区在运行时被标记为“有毒”的，用于捕获任何对这些区域的非法访问。下面是转换后的代码示例：

```
1  void foo() {
2    char rz1[32]; // 前红区
3    char arr[10]; // 原始数组
4    char rz2[54]; // 后红区，大小为32-10+32
5    unsigned *shadow =
6    (unsigned*)(((long)rz1>>8)+Offset);
7    // 标记红区和数组周围为有毒
8    shadow[0] = 0xffffffff; // rz1
9    shadow[1] = 0xffff0200; // arr 和 rz2
10   shadow[2] = 0xffffffff; // rz2
11
12   // 函数体
13   // 解毒所有区域
14   shadow[0] = shadow[1] = shadow[2] = 0;
15 }
```

同时，ASan 是线程安全的。影子内存只在相应的应用程序内存不可访问时被修改（在 malloc 或 free 内部、在创建或销毁栈帧期间、在模块初始化期间）。所有其他对影子内存的访问都是读操作。malloc 和 free 函数使用线程局部缓存来避免每次调用都进行锁定（正如大多数现代 malloc 实现所做的）。如果原始程序在内存访问和该内存的删除之间存在竞态，ASan 有时可能将其检测为使用后释放错误，但不能保证总能检测到。每次 malloc 和 free 的线程 ID 都被记录，并在错误消息中与线程创建的调用栈一起报告。

4. 基于覆盖率的遗传算法

基于代码覆盖率的遗传算法是一种将遗传算法原理应用于软件测试的方法，特别是在模糊测试中，目的是通过优化测试用例来提高代码覆盖率，从而提升软件测试的全面性和效率。遗传算法是受自然选择原理启发的优化技术，通过模拟生物进化过程中的选择、交叉（杂交）和变异操作，来在解空间中搜索最优解。

具体而言，在进行模糊测试的过程中，程序通过代码覆盖率的高低来判断种子是否有趣。代码覆盖率是一种简单的动态分析方法，常用于衡量测试的全面性。它通过将源代码分割成具有一定粒度的“块”并追踪在运行时遇到了哪些块来实现。有时，还会记录在测试中遇到某个特定块的次数。测量的覆盖率的粒度可以有所不同，并且存在多个定义来命名不同粒度，但三种常用的定义包括块覆盖率、决策覆盖率和条件覆盖率。块覆盖率：在块覆盖率中，一个块是指没有 if 语句或其他控制语句将执行从块中引导出去的代码片段。在运行时计算这些块的覆盖率。决策覆盖率：决策覆盖率寻找应用程序做出决策的地方，主要是 if 语句，然后分析所有可能的决策被覆盖得有多好。条件覆盖率：条件覆盖率也寻找 if 语句，并尝试追踪这些分支中所有不同的布尔值是否已经被测试过。然而，如果应用程序的 if 语句包含多个条件进行检查，条件覆盖率并不保证决策覆盖率。

基于代码覆盖率的遗传算法首先定义一个适应度函数来衡量测试用例的优劣，这里的适应度函数即代码覆盖率。算法从一组测试用例开始，构成了初始种群。通过运行这些测试用例并根据运行信息得到它们的代码覆盖率，以此来评估种群中每个个体的适应度。并将产生的适应度高的新测试用例替换种群中适应度较低的测试用例。重复上述步骤，直到达到预定的迭代次数或代码覆盖率达到预期水平。

基于代码覆盖率的遗传算法在模糊测试中起着非常重要的作用，它可以自动化地生成能够触发新路径的输入，为软件安全和质量提供了一个强大的工具。通过不断地迭代和优化，这种方法能够有效地提升测试的深度和广度，帮助开发识别和修复软件中的漏洞。

5.ROS2 系统机制

在 ROS 2 的体系结构中，节点（Node）是最基本的执行单位，负责特定功能的实现，如数据处理或硬件接口。每个节点可以包含多种通信机制，包括订阅者（Topic Subscribers）、定时器（Timer）、服务服务器（Service Servers）和服务客户端（Service Clients），如图9。这些通信实体的目的是为了接收和发送数据，以及提供不同的服务。

节点注册到执行器（Executor）中，执行器是一个控制实体，负责协调节点的活动。当节点的一个通信事件发生时，比如收到一个主题消息或服务请求，执行器会调用相应

的处理函数，或称为回调函数（Callback）。这些回调函数是预先定义的，用来响应特定类型的事件，如 `topic_receive()` 用于处理主题消息，`request_receive()` 用于处理服务请求，`time_up()` 用于处理定时器完成计时。

在节点的生命周期中，可以使用生命周期状态机（Lifecycle SM）来管理节点的状态，这在管理复杂节点时特别有用。生命周期状态机允许节点在不同状态之间转换，如激活（activate）、去激活（deactivate）和清理（cleanup）。每个状态变化都可以有对应的回调函数，例如 `setting()` 在节点激活时调用，用于声明节点接口、节点执行器（Executor）、通信订阅和服务订阅等一系列节点功能实体；`activate()` 用于节点初始化完毕开始服务，此时节点处于正常工作状态；`cleanup()` 在节点清理资源时调用，用于释放节点所持有的微机资源。

ROS2 事件处理机制:

节点调用 `spin()` 函数来持续检查和处理事件，如伪代码1。

```
spin():
    while True:
        new_msg <- ros_dds_listener()
        handler <- get_callback(new_msg)
        Executor.execute(*handler)
        sleep(0.1)
```

伪代码 1: ROS2 事件循环示例

`spin()` 实质是一种事件循环，它会保持节点循环和监听，检查系统中是否有新的 ROS 消息或待处理的消息，获取到消息实体后，根据消息类型获取节点初始化时注册的回调函数，交付给节点执行器去执行，从而完成对该事件（消息）的响应。`spin()` 函数是一个事件循环，保持节点持续运行并响应事件。

在执行器内部，`execute_any_*()` 函数是对 ROS 核心细节的抽象，这个函数根据事件的类型和优先级选择一个事件来处理。处理过程包括执行注册的回调函数，这些函数是在节点初始化时注册的，如 `timer.registe_handler()` 用于注册定时器事件的回调；`subscriber.regeste_handler()` 用来注册监听到某话题时触发的回调函数，以处理该数据。

总体而言，ROS2 的程序机制基于事件驱动的模型，通过节点、执行器和回调函数协同工作来响应和处理各种事件，这些事件可能来源于数据的接收、定时器的触发、服务的请求和响应，以及节点生命周期状态的变化。通过这种灵活且模块化的设计，ROS 2 能够支持复杂且多样化的机器人系统开发。

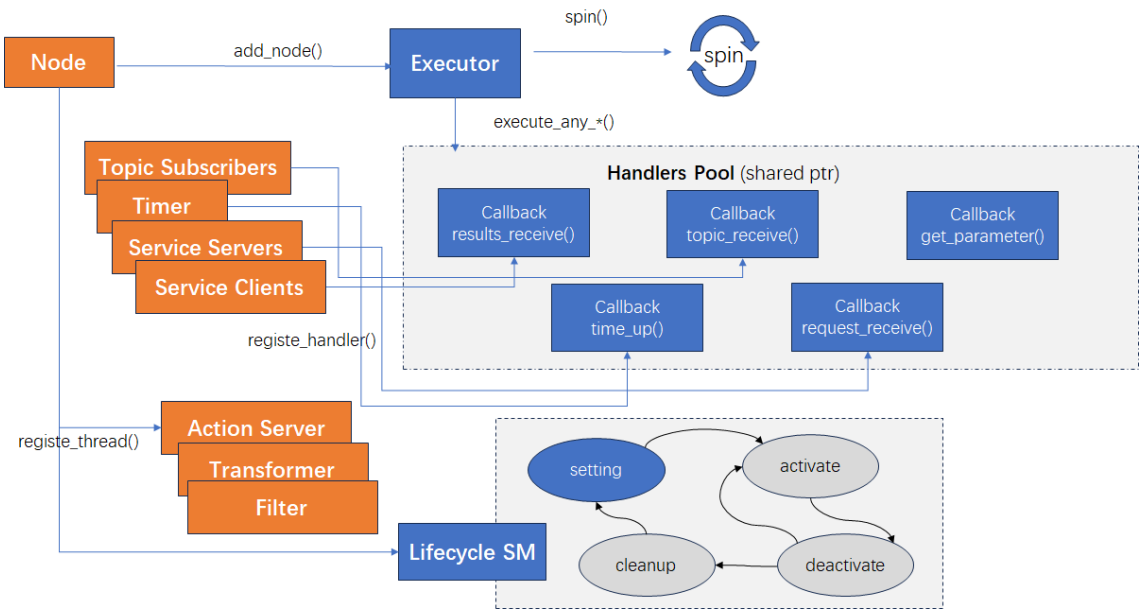

图 9 ROS 节点内部模型, 初始阶段

ROS2 节点释放

当 ROS2 系统的节点将释放时，节点的生命周期状态机会从 activate 状态转换为 deactivate 状态，此状态仅用于让节点完成必要的最后工作，如完成正在处理的数据传输或请求，然后进入非活跃状态。

紧接着，节点生命周期状态机转为 cleanup 状态，见图10，调用 `cleanup()` 回调函数来释放节点拥有的资源，这包括所有回调函数（callback handlers），如和话题订阅、定时器、服务等回调函数。然后清理节点所打开的文件，断开网络连接，释放 ROS2 上下文，最终释放内存。

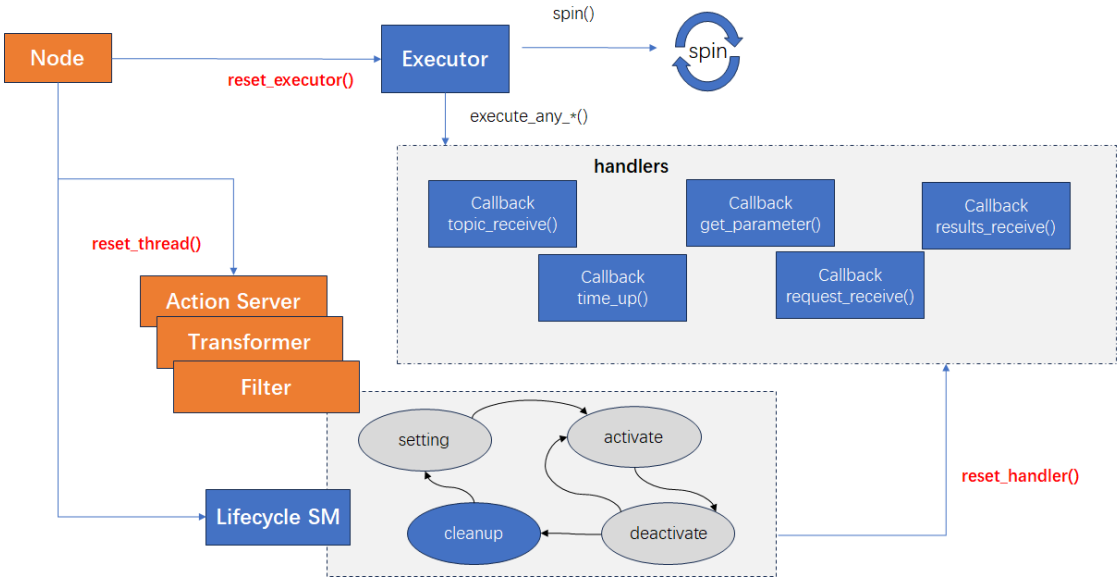

图 10 ROS 节点内部模型, 释放阶段

ROS2 通信机制

为了适应机器人系统中复杂的通信需求，ROS2 设计了一套专用的通信模型。其基础为“发布、订阅“模型，支持节点间的异步数据交换。该机制允许 ROS2 系统内的节点（Node）独立地发布（Publish）和订阅（Subscribe）消息，而不需要彼此之间的直接连接或相互了解。这种设计显著提高了系统的灵活性和扩展性，因为它允许任何数量的发布者和订阅者存在于同一个话题上，从而实现了高效的多对多通信。

在 ROS 2 的架构中，每个节点可以根据其功能需求，声明为发布者或订阅者，类似图 11，或同时充当这两种角色。发布者负责生成并发布特定类型的消息到一个命名的话题上，而订阅者则监听这个话题，接收并处理传入的消息。监听到并处理消息时，使用的是订阅该话题时传入的回调函数来自动处理，订阅者不必同步阻塞就能等待处理订阅消息。话题本质上是一个数据通道，它通过唯一的名称标识，确保消息的传递和接收的一致性。每个话题都与一个明确的消息类型相关联，这个消息类型使用 YAML 格式定义各字段的静态变量类型，从而定义了该话题传输数据的结构。

此通信机制的一个核心优点是其异步性，使得发布者和订阅者可以独立地操作，增加了处理并发消息的能力。此外，由于发布者和订阅者之间的解耦，系统组件可以被设计得更为模块化，易于开发和维护。例如，在自动驾驶的应用场景中，激光雷达传感器节点可以发布其测量到的数据到话题 \scan，而负责建图和规划路径的节点则通过订阅该话题来实时获取雷达信息，从而完成最终决策。

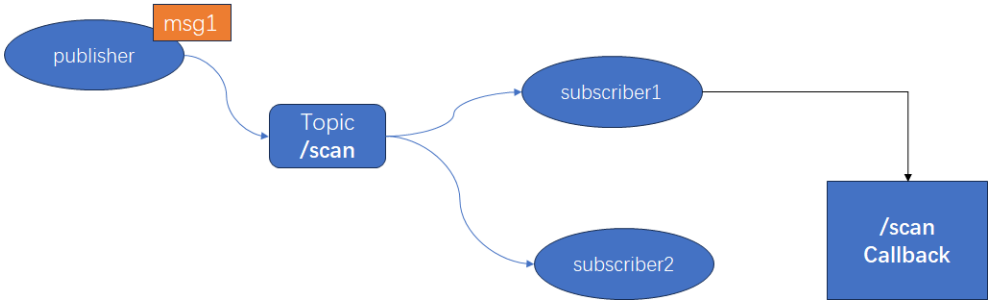

图 11 ROS 话题通信机制

ROS2 通信机制底层实现

ROS2 通信机制底层实质是使用 UDP Socket 来进程间通信，使用的协议为物联网 DDS 协议，ROS2 核心库对其进行了抽象封装，已便捷地被开发者调用。具体而言，对于每种 DDS 协议版本，ROS2 会实现一种网络中间件对其 API 进行包装，并实现由 ROS2 各类消息形式向 DDS 消息格式的序列化与结构化转化。如图12

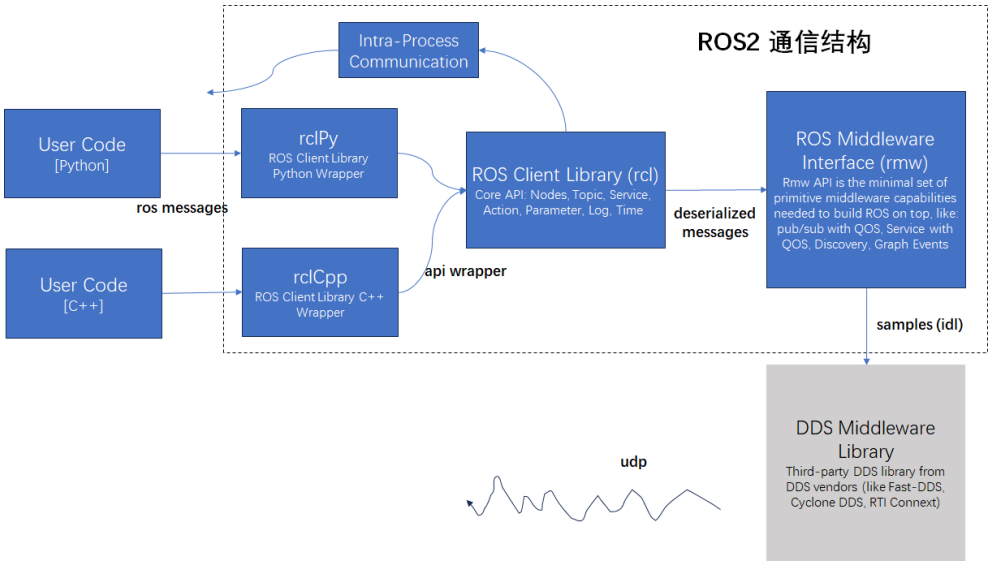

图 12 ROS 底层通信机制实现

（二）关键设计

（三）系统需求分析

本项目核心目标是将自动模糊测试（Fuzzing）框架迁移到 ROS2 系统上，并且针对 ROS2 特性对测试效率做出改进提升。具体而言，我们试图寻找更多可能的程序输入口，使测试覆盖的输入可能性尽可能多；尝试构造质量尽可能高的测试用例；寻找更多反馈信息，用于监控被测程序（即 ROS2 节点）的内部状态，以更高效地指导对输入的自动变异测试；尝试模拟更多 ROS2 执行环境，模拟程序可能遇到的各种实际情况。

1. 通信环境测试

如前所述，在机器人操作系统中，节点间通信方式可以基于本地网络套接字、有线通信、进程内部通信等方式。在实际应用场景中，不能保证网路中通信的稳定性，如图13所示，可能出现乱序、重复、丢包、变频等报文序列变化以及报文内容本身的变化。针对这一观察，我们的 ROS Fuzzer 尝试对 ROS2 系统内部通信报文进行类似的变异，以测试 ROS2 程序的鲁棒性。

已有测试框架 Rozz[2] 建议在 ROS2 系统话题通信机制的订阅者和发布者建立恶意中介节点，它会拦截消息发布者的消息，然后对消息进行变异后再转发给订阅者。本作品认为该方案适用性有限，首先完全拦截消息难度大，尤其是对于节点内部硬编码的话题；其次 ROS2 抽象层提供的 QoS 机制和时间戳机制能够过滤非法消息和保证通信质量，在 ROS2 上层进行消息变异容易被识别为非法，不能较好模拟实际的网路不稳定性。

ROS2 系统支持使用不同的 DDS 协议中间件实现，如 Fast-DDS、Cyclone-DDS

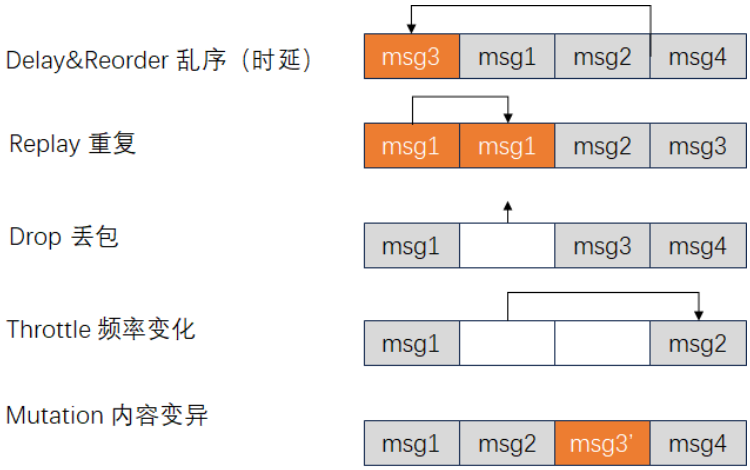

图 13 不稳定网路中的报文变化

等，从而灵活适配不同的需求。利用这一点，本作品创新性地用魔改的 DDS 通信中间件替代原有基础 DDS 中间件，如图14所示，直接在底层套接字通信实现中进行报文变异，从而更真实地模拟底层网路的不稳定性。

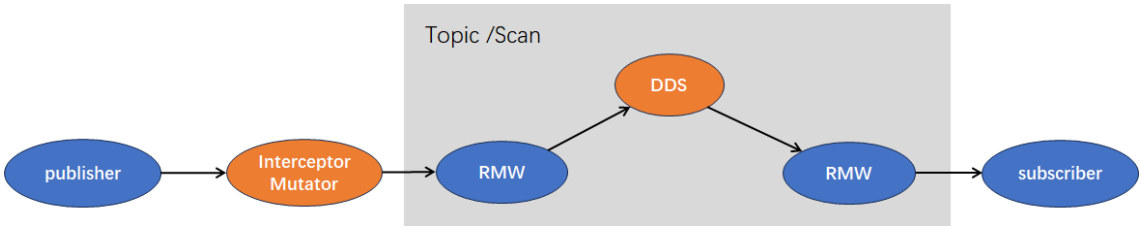

图 14 本作品对不稳定网路的模拟机制

2. 多维度输入变异

除了对上述 ROS2 内部通信消息进行变异，本作品同样也对传统程序输入进行变异，如输入 ROS2 节点程序的命令、ROS2 节点配置文件、用户交互数据等，见图15。通过多个维度输入的变异，本作品可以高效地探索程序内部状态，更高效地尝试触发异常状态，检测程序安全漏洞。

3. 基于延迟插入的并发漏洞检测

如背景知识一节所述，ROS2 系统有典型的并发特征，在节点间是基于消息通信的多进程并发模型，而在节点内则是基于共享内存的多线程并发模型。基于消息通信的并发模型容易出现死锁等阻塞漏洞，基于共享内存的并发模型则容易出现数据竞争等并发漏洞，这两点导致了基于 ROS2 的程序存在较多并发漏洞，对 ROS 系统漏洞的统计工作 [3] 也证明了这一点。

针对这一点，本作品尝试通过延迟插入技术来检测并发漏洞，工作 [1] 已说明这种方法具有漏洞探测的便捷性与高效性。通过举例来解释该技术：在正常程序中，当对象

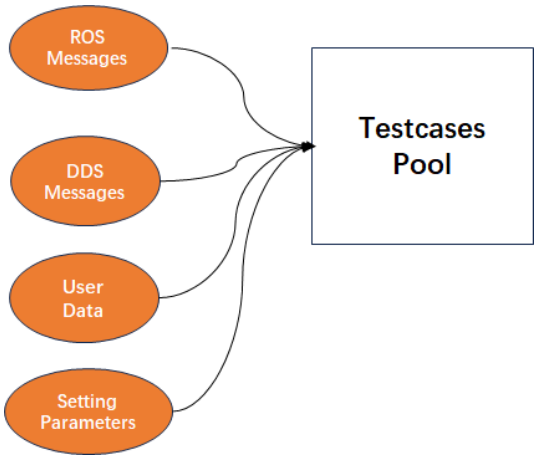

图 15 本作品针对多维度输入进行测试样例生成

A 被使用时，它不应被释放或已被释放，对对象 A 的释放操作应阻塞等待所有对对象 A 的使用操作结束再执行；本作品会尝试在某些边界情况中（如系统频繁申请与释放资源时）在对象 A 的使用操作前插入延迟，如图17右图，如果此时对 A 的释放操作没有正确阻塞等待对象 A 的使用完毕，而直接执行，就会造成严重的 UAF 内存错误，因为此时使用的对象 A 已经被释放。

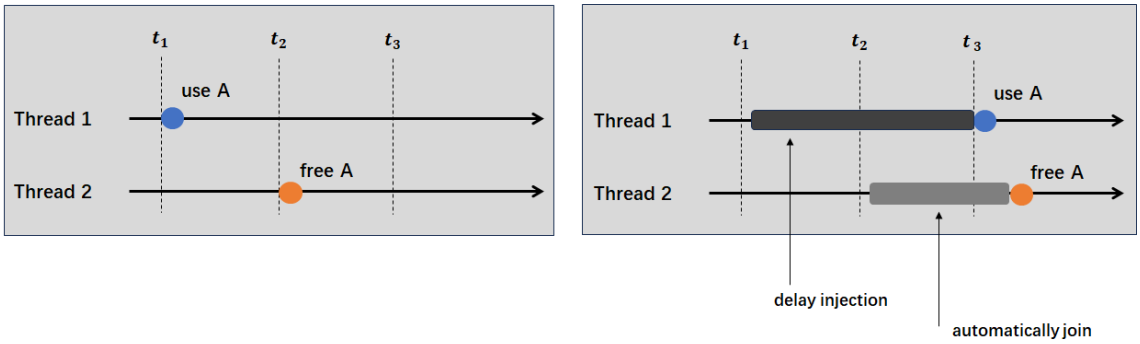

图 16 延迟插入技术的探测漏洞原理

延迟插入技术可能拖慢整个程序的运行速度，所以具体插入位置需要谨慎选择。如前所述，ROS2 系统高度抽象化了各个子功能组件，将各个功能将回调函数绑定给执行器，执行器负责监听事件并触发相应回调函数。统计 [3] 显示这种线程模型是导致潜在数据竞争的重要原因，由此，本作品基于 LLVM 框架对组件的回调函数前后插入检查点，如图 17所示，对子线程可疑行为进行延迟插入，观察能否触发数据竞争漏洞。

4. 测试反馈

模糊测试中，对“测试样例自动生成”的反馈指导质量直接影响了最终的测试效率，这种反馈则取决于对被测程序的信息收集。传统被收集的信息包括调试信息、代码覆盖率等，这些信息被认为反映了被测程序内部状态的变化。在此基础上，本系统创新性地

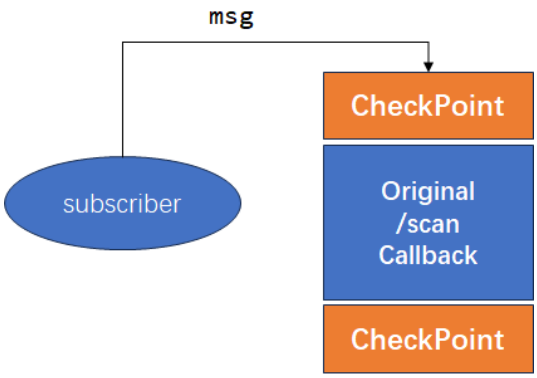

图 17 延迟插入技术的实现

利用我们魔改的 DDS 中间件，对 ROS2 节点的流量进行分析，如图18；同时利用插桩的检查点（如图17）收集节点的功能组件回调函数执行信息，如执行时间、频率、传入的消息内容等。这些额外信息更准确地反映了被测 ROS2 节点内部状态，指导我们更好地反馈生成新测试样例，从而提高发掘漏洞的效率。

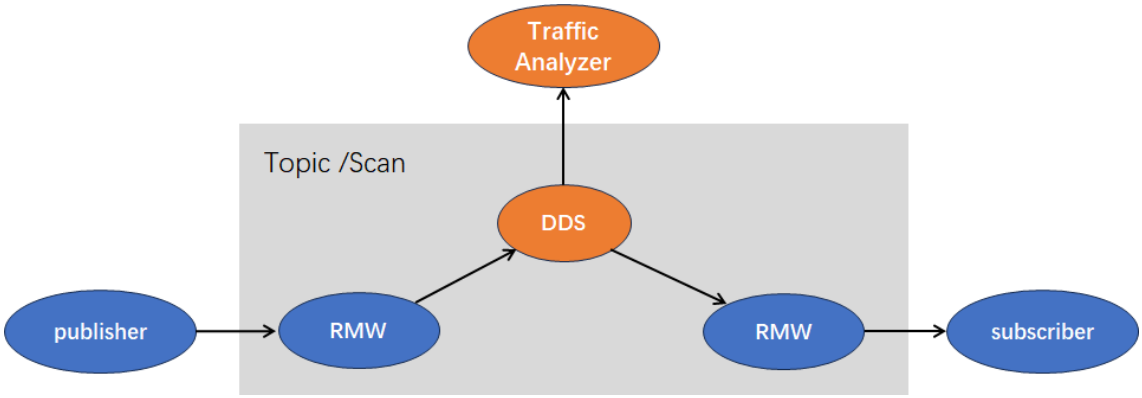

图 18 利用替换的 DDS 中间件进行流量分析

5. 测试加速

复杂机器人系统通常资源消耗大，对本地网络污染严重，进程管理复杂，这导致了对其测试的低效性。测试人员或自动测试程序难以实现对测试行为的并行化：本地网络中存在的多个同名程序实例（如多个 Navigation2 导航程序）会被其他实例的通信消息所干扰，这是因为 ROS2 系统会共享同一个通信守护进程；一个基于 ROS2 的项目大多是多进程多线程的，每个进程有独立进程组，关闭程序时容易遗留孤立进程，持续干扰整个 ROS2 环境。

针对这一情况，本作品创新性地使用 Docker 对不同测试实例进行进程、网络的隔离；搭配使用随机化 ROS2 命名空间，提供更轻量化的网络隔离。通过这种方式，本作品可以充分利用电脑性能进行并行化测试，变相大幅提升了自动测试效率，当然这也增加了本作品模糊测试程序的开发难度。

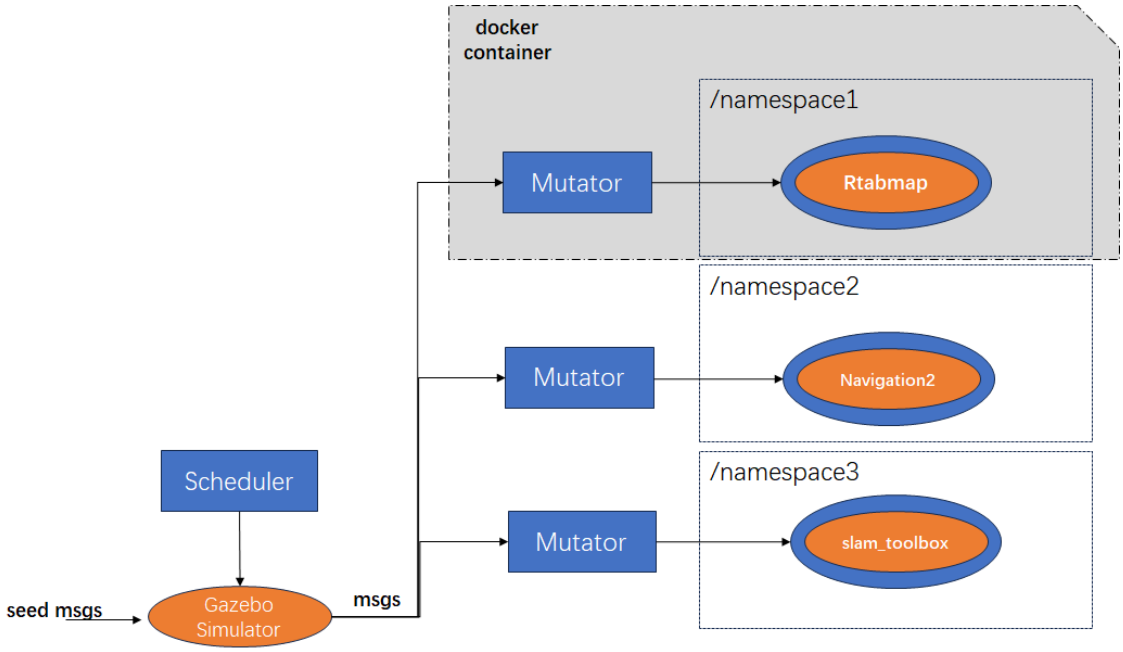

图 19 测试的容器化与并行化

三、作品测试与成果分析

（一）测试环境介绍

本作品在 ROS2 中测试了 10 个常见的机器人程序。根据最新版本的测试结果得到下列图表，表格2显示了这些用于测试的 ROS 程序的信息 (LOC 指代码行数)。为了执行机器人导航 (用于移动和定位程序) 和地图构建 (用于移动和 SLAM 程序) 任务，这些程序在机器人仿真框架 Gazebo11.5[4] 中的虚拟机器人 TurtleBot3 Waffle 上执行。本测试使用的虚拟传感器包括激光雷达，里程计，2D 相机和 IMU(惯性测量单元)。实验运行在一台 x86-64 台式机上，配有 32 个 Intel 处理器核心和 64GB 物理内存。使用的操作系统为 Ubuntu 22.04, ROS2 版本为 ROS2 Humble。

表 2 用于测试的 ROS 程序信息

类型	程序	描述	LOC
导航	navigation2	ROS2 导航系统	25.8K
定位	nav2_amcl	ROS2 导航中的定位模块	14.4K
定位	lama_loc	可选的定位和映射方法	9.5K
定位	ekf_loc	基于卡尔曼滤波的定位方法	0.8K
建图	rtab-map	实时 RGB-D SLAM 方法	26.1K
建图	slam_toolbox	一套用于 2D SLAM 的工具和功能	15.1K
底层	ros2rclcpp	ROS2 底层基础设施	177.1K

（二）测试成果与漏洞统计

在作品开发过程中，我们对上述项目进行了广泛测试，表格3显示了截稿时的漏洞发掘结果。

表 3 机器人程序在 ROS2 模糊测试结果

程序	覆盖分支		漏洞检测	
	模糊测试	测试样例	发现的漏洞	受修复漏洞
navigation2	37.6K	36.8K	12	8
lama_loc	20.8K	20.7K	2	0
ekf_loc	3.3K	5.4K	1	0
rtab-map	70.9K	59.3K	3	1
slam_toolbox	12K	10.7K	3	2
cartographer	51.9K	52.1K	4	0
rclcpp	None	47.2K	3	0
总计	None	None	28	11

测试覆盖率。本作品初始输入样例（即未经过自动生成的初始测试输入）包含了官方的测试用例，但基于这部分测试样例并没有发现新的漏洞，对代码覆盖率贡献也较低，侧面反映了传统测试用例的低效性。本作品能否稳定提高开源程序测试代码覆盖率，部分项目由于多版本问题覆盖率才略低于官方测试样例。

漏洞检测。本作品在测试程序中发现了 28 个真实的漏洞，并提供了稳定的复现办法。本团队已经向相关 ROS 开发者报告了这些漏洞，其中活跃项目（如 navigation2）中 11 个漏洞已经得到了确认和修复，部分低活跃程序仍在等待回复。其中 Navigation2 项目中 4 个危险漏洞被美国通用漏洞披露库（CVE）收录，这充分证明了本作品检测出的漏洞的价值。

（三）ROS 漏洞实例分析

下面挑选我们的漏洞测试系统在 ROS 相关项目中检测出的几个典型漏洞进行分析，它们充分体现了本系统的价值。在Navigation2 Issue#4177中，我们指出了一个典型的数组越界漏洞。

Navigation2 系统中的节点会监听 `\costmap`，来更新代价地图信息，但是缺乏对代价地图信息合法性的检查。如伪代码2中的 map 消息，width 和 height 字段指出了代价地图的宽高，data 字段给出了代价地图的数据。对于正常消息，$width \times height$ 应该等于 data 数组的大小，但这一点对于传输错误或恶意投放的消息这并不是被严格保证

的。

```
ros2 topic pub /map nav_msgs/msg/OccupancyGrid "
...
info:
  ...
  width: 2
  height: 2
  ...
data: [-1, 0, 1, 1, -1]  " <- would cause buffer overflow
```

伪代码 2: 漏洞 #4177 恶意消息

Navigation2 自动导航项目并没有对这一点进行检查，而直接用 width 和 height 声明存储 data 的数组变量。当 `width` 和 `height` 大于数组 `data` 的实际大小时，会导致程序访问越界（SEGV）。初始化地图时该 bug 可能触发，见3第 6 行；运行时，如果接收到外部更新的地图信息，也可能触发该问题，见3第 24 行。

```
/* 初始化时可能发生 */
// create the costmap
costmap_ = new unsigned char[size_x_ * size_y_];

for (unsigned int it = 0; it < size_x_ * size_y_; it++) {
  data = map.data[it]; // <- SEGV
  if (data == nav2_util::OCC_GRID_UNKNOWN) {
    costmap_[it] = NO_INFORMATION;
  } else {
    ...
  }
}

/* 过程中也可能发生，比如接收到外部更新的地图信息 */
void CostmapSubscriber::toCostmap2D()
{
  auto msg = std::atomic_load(&costmap_msg_);
  ...

  unsigned char * master_array = costmap_->getCharMap();
  unsigned int index = 0;
  for (unsigned int i = 0; i < msg->metadata.size_x; ++i) {
    for (unsigned int j = 0; j < msg->metadata.size_y; ++j) {
      master_array[index] = msg->data[index]; //<-SEGV
      ++index;
    }
  }
}
```

伪代码 3: 漏洞 #4177 问题代码

在 Naviagtion2PR#3972, PR#3958, Issue#3940中, 我们报告了一个典型的并发 bug，本系统通过插桩延迟技术，找到了 Navigation2 项目中 Costmap 结构数据竞争的根本原因，删除了之前大量不必要的检查。

此 bug 发生在关闭期间，ROS 系统使用 Lifecycle 状态机机制来管理节点生命周

期，Controller、Planner、CostmapROS 都是具有生命周期的节点，CostmapROS 是 Controller 和 Planner 的子节点（独立线程）。当 ROS2 系统关闭时，Lifecycle 状态机发送同一关闭信号，这三种节点都会直接对该信号进行响应然后执行资源释放程序，此时三者关闭顺序是不一致且不确定的。

如果子线程 CostmapROS 节点已经自行释放，Controller 或 Planner 实质失去了对其的控制权，如果它们仍试图访问 CostmapROS，就产生了重复释放（UAF）、资源冲突等一系列并发问题，如伪代码4所示。

```
// Planner or Controller:
Controller::on_cleanup(const rclcpp_lifecycle::State &){
  costmap_ros_->cleanup() // <-UAF, costmap already cleanup itself.
}
```

伪代码 4: 漏洞 #3972 问题代码

对于需要频繁开关的机器人节点，关闭期间内存漏洞可能产生严重后果，但往往被开发者忽略。在 Navigation2 Issue#4175，PR#4180，Issue#4166，Pull#4176，和 RclCpp Issue#2447 中，我们和 ROS2 开发者讨论了 ROS2 节点退出机制可能导致的一些并发问题。

ROS2 惯例是将收到订阅消息、收到服务请求的回调函数交给执行器（Executor）执行，因此执行器线程拥有该回调函数的函数指针，当该回调函数所属的类已被释放，如伪代码5第 5 行，执行器仍可能访问该回调函数，造成 UAF 错误。

```
nav2_util::CallbackReturn
AmclNode::on_cleanup(const rclcpp_lifecycle::State & /*state*/){

  ...
  initial_pose_sub_.rest(); // <- UAF

  ...
  executor_thread_.reset();
}
```

伪代码 5: 漏洞 #4176 问题代码

本团队还根据发现的 28 个漏洞的类型进行了分类，并将结果总结在表 4中。具体来说，有 7 个并发类型漏洞（包括数据竞争和少数死锁），7 个释放后使用错误（UAF），3 个缓冲区/堆栈溢出错误，3 个无效指针访问（SEGV）和 8 个未捕获异常。一旦这些错误被特定的输入触发，运行时故障和严重安全问题就会发生，严重威胁机器人的安全性和可靠性。

表 4 发现漏洞的种类

程序	数据竞争/死锁	UAF	溢出	SEGV	未捕获异常	总计
navigation2	1	4	3	2	2	12
lama_loc	2	0	0	0	0	3
ekf_loc	0	0	0	0	1	1
rtab-map	0	0	0	1	2	3
slam_toolbox	2	1	0	0	0	3
cartographer	1	0	0	0	3	4
rclcpp	1	2	0	0	0	3
总计	**7**	**7**	**3**	**3**	**8**	**28**

内存安全错误，如 UAF、BufferOverflow、SEGV，由于易被攻击者利用和植入恶意程序，往往更受开发者关注，但尽管如此，超过一半的漏洞威胁仍属于内存安全。由于机器人和自动驾驶相关项目的应用特殊性，程序异常应给予日志记录而不应直接造成程序异常终止；但由于复杂开源项目的异常处理管理难度大，常常出现开发者间合作不慎而导致异常未被捕获的情况，底层开发者可能认为应用层开发者会处理该异常，而顶层开发者却可能也认为底层开发者已经处理了此异常情况。部分 UAF 漏洞也可能由数据竞争等并发漏洞引发，部分数据竞争漏洞大部分时间可能不会影响程序运行，造成隐蔽性极高，开发者确认和修复难度也大；相反，死锁等阻塞型漏洞，通常由于错误使用 ROS2 通信机制导致，但容易修复和确认。

（四）创新性说明

1. 创新地将模糊测试技术迁移

本作品创新性的将模糊测试迁移到了 ROS 框架中来，拓展了传统软件测试方法的应用范围，这一创新性的做法为机器人系统安全性和鲁棒性的提升开辟了新的途径。特别是在自动驾驶车辆、工业及服务机器人等领域，安全性是至关重要的。通过针对 ROS 系统特有的模块化和分布式架构进行定制化的模糊测试，不仅能有效发现和修复潜在漏洞，提高系统对异常输入的处理能力，还能增强系统面对复杂物理环境的稳定性。

此外，这种方法的引入，促进了 ROS 社区对安全问题的广泛关注，有助于建立更安全的 ROS 生态系统。同时，这也为机器人软件测试领域带来了新的测试工具和思路，给机器人软件开发中测试和验证方法的重要进步。总的来说，将模糊测试技术应用于 ROS 框架，不仅为机器人系统的安全性和稳定性提供了强有力的支

持，也为机器人软件测试和验证方法的创新开辟了新的方向。

2. 创新测试方法，高效性

本作品相较于传统的软件测试方法，有效提升了代码的测试的覆盖率与准确率的同时，运行效率也实现了大幅度提高。本作品能也成功发现数十个被常规测试方法忽略的隐蔽漏洞，特别是那些与安全高度相关的漏洞，如缓冲区溢出、内存泄露等。由于其自动化特性，大大提高了测试效率。

此外，模糊测试易于集成进现有的开发和测试流程，尤其是与持续集成系统相结合时，可以实现持续的安全监测，进一步减少安全漏洞带来的风险。通过在软件发布前发现并修复潜在的安全问题，模糊测试有助于减轻安全事故的风险，保护企业和用户的数据安全，同时节省潜在的调查、修复和法律相关的成本，展现出其良好的成本效益比。这些优势共同促进了模糊测试成为提高软件安全性和可靠性的重要工具。

本作品的核心模糊测试程序优势如下：

1. 高覆盖率

ROS 系统的架构复杂，包含多个节点和话题，而本作品能够系统性地探索这些节点和话题之间的交互，从而实现全面覆盖。通过生成各种可能的输入和交互序列，本作品能够有效地发现潜在的漏洞和异常情况，为系统的安全性和稳定性提供保障。例如，它可以模拟不同的传感器输入和控制命令，以验证 ROS 系统对于各种情境的响应是否符合预期，从而增强系统的健壮性和可靠性。通过分布式分支覆盖，它能够捕获不同节点之间的代码执行路径，从而更全面地测试机器人程序。本作品使用多维生成方法生成 ROS 程序的测试用例，包括用户数据、配置参数和传感器消息。这种方法有助于覆盖不同维度的输入空间，提高测试的全面性。

2. 高效性快速收敛

通过自动化生成和执行测试用例，本作品大幅提高了测试效率。快速生成大量测试用例，并采用智能化的测试执行策略，使得工具能够在短时间内快速收敛到潜在问题，帮助开发人员及时发现和修复 bug，从而提高开发周期效率。举例来说，本作品可以根据程序执行路径的优先级和覆盖情况，动态调整测试用例的生成和执行顺序，优先测试具有潜在问题的部分，从而更快地发现关键问题。

3. 可拓展性

由于 ROS 系统的多样性和复杂性，模糊测试工具必须具备良好的可拓展性，以应

对不同类型和规模的 ROS 应用。这种工具通常设计成模块化的结构，能够轻松集成新的测试策略和技术，满足不断变化的 ROS 系统和应用场景的需求，使得测试工作更加灵活和适应性更强。例如，开发人员可以根据具体需求扩展工具的测试生成器、执行器或分析器，以适应新的 ROS 功能或更复杂的系统结构，从而提高测试的适用性和覆盖范围。

4. 准确性高

在生成测试用例时，工具会考虑 ROS 系统的特性和约束，例如消息格式、节点通信方式等，以确保生成的测试用例有效且具有代表性。通过使用各种静态和动态分析技术，工具能够准确检测潜在问题和异常情况，并提供详尽的测试反馈和报告，为开发人员提供准确的问题定位和解决方案，从而提高系统的质量和可靠性。例如，工具可以监视 ROS 节点之间的消息传递，分析消息格式和内容，以及节点的响应时间，从而发现潜在的性能瓶颈或通信异常，帮助开发人员改进系统的设计和实现。本作品使用时态变异策略生成带有时间信息的测试用例。这有助于模拟实际机器人系统中的时间相关行为，提高测试的准确性。

（五）前景分析

1. 创新维度

本作品创新性的将模糊测试迁移到了 ROS 框架中来，拓展了传统软件测试方法的应用范围，这一创新性的做法为机器人系统安全性和鲁棒性的提升开辟了新的途径。特别是在自动驾驶车辆、工业及服务机器人等领域，安全性是至关重要的。通过针对 ROS 系统特有的模块化和分布式架构进行定制化的模糊测试，不仅能有效发现和修复潜在漏洞，提高系统对异常输入的处理能力，还能增强系统面对复杂物理环境的稳定性。

2. 团队维度

团队成员均具备计算机学科专业知识，实践能力、创新能力较强，价值观念积极向上。项目成员曾参与重点研发计划课题“大学生创新创业计划”、“创新创业资助计划”。在整个作品实现过程中，团队成员能力互补、分工合理、互帮互助，在事先进行充分调研的基础上逐步完善作品功能。队长负责统筹规划及解决各方面疑难，其余成员分别负责代码编写、测试及论文撰写。每位队员优势均得到充分发挥，为作品最终完成奠定基础。团队与项目关系真实紧密，在充分吸收现有电子病历系统优点的基础上形成系统框架，能够保障系统平稳运行。团队未来投身

创新创业的可能性较大，具备将本项创新成果以创业形式进行实现进而服务社会的能力。

3. 商业维度

随着机器人技术的广泛应用，如自动驾驶车辆、服务机器人、工业自动化等，系统的安全性和可靠性变得尤为重要。一个专门针对 ROS 的模糊测试框架能够帮助开发者在产品发布前发现和修复潜在的漏洞和错误，从而减少安全事故的风险，提升产品的市场竞争力。同时，通过自动化测试减少手动测试的需求，模糊测试框架可以显著降低软件的开发和维护成本，进而加速开发流程，使开发团队能够更快地迭代和改进产品。

4. 就业维度

针对 ROS 的模糊测试框架对促进就业有显著贡献，通过创造新的专业技术岗位、促进跨学科人才培养、激发创业和技术创新，为软件开发、机器人技术和信息安全领域提供了丰富的就业和职业发展机会。本项目不仅提升了个人职业竞争力，也推动了整个行业的技术进步和经济发展，对社会就业格局产生了积极影响。

5. 社会服务

针对 ROS 的模糊测试框架对社会服务的贡献主要体现在提升机器人技术在各类社会服务中的安全性和可靠性上。通过确保机器人系统的健壮性和防御潜在漏洞，这些技术能够更安全、有效地服务于公共安全、医疗卫生、教育、灾难响应等关键领域。例如，在公共安全领域，经过严格测试的机器人系统可以用于搜索与救援任务，降低人员伤亡风险；在医疗卫生领域，可靠的机器人辅助手术系统可以提高手术精准度，改善患者预后；在灾难响应中，经过模糊测试的机器人系统能够在极端环境下执行救援任务，提高救援效率。总而言之，模糊测试框架通过提高 ROS 系统的安全性和稳定性，为社会各领域提供了更加可靠和高效的机器人服务解决方案，对提升公共福利和应对社会挑战具有重要意义。

结论

傻逼冯如杯

参考文献

[1]Stoica, B.A., Lu, S., Musuvathi, M., & Nath, S. *WAFFLE: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection*. In EuroSys'23, May 2023.

[2]K. -T. Xie, J. -J. Bai, Y. -H. Zou and Y. -P. Wang. *ROZZ: Property-based Fuzzing for Robotic Programs in ROS*. 2022 International Conference on Robotics and Automation (ICRA), Philadelphia, PA, USA, 2022, pp. 6786-6792.

[3]Timperley, C.S., van der Hoorn, G., Santos, A., Deshpande, H., & Wasowski, A. *ROBUST: 221 bugs in the Robot Operating System*. Empirical Software Engineering, 29(3), 57, 2024.

[4]"Gazebo: a robot simulation framework." Available: http://gazebosim.org/.

[5]Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. *AddressSanitizer: A Fast Address Sanity Checker*. In USENIX ATC 2012, 2012.

[6] Delgado, R., Campusano, M., & Bergel, A. *Fuzz testing in behavior-based robotics*. In Proceedings of the 2021 International Conference on Robotics and Automation (ICRA), 9375–9381, 2021.

[7] Woodlief, T., Elbaum, S., & Sullivan, K. *Fuzzing mobile robot environments for fast automated crash detection*. In Proceedings of the 2021 International Conference on Robotics and Automation (ICRA), 5417–5423, 2021.

[8] Hutchison, C., Zizyte, M., Lanigan, P.E., Guttendorf, D., Wagner, M., Le Goues, C., & Koopman, P. *Robustness testing of autonomy software*. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 276–285, 2018.

[9] Kim, S. & Kim, T. *RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs*. In Proceedings of the 30th International Symposium on the Foundations of Software Engineering (FSE), 447–458, 2022.

[10] "Ros2-fuzz: automatic fuzzing for ROS2." Available: https://github.com/rosin-project/ros2 fuzz.

[11] Zalewski, M. *American Fuzzy Lop: A Security-Oriented Fuzzing Tool*. In Proceedings of the Black Hat USA, 2015.