中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

基于关键位置的模糊测试改进研究

作者姓名：王化磊

指导教师：程亮 副研究员

中国科学院软件研究所

学位类别：工学硕士

学科专业：软件工程

培养单位：中国科学院软件研究所

2022 年 6 月

Research on Fuzzing Improvement Based on Key Position

A Thesis Submitted to

The University of Chinese Academy of Sciences

In partial fulfillment of the requirement

for the degree of

Master of Science in Engineering

in Software Engineering

By

Wang Hualei

Supervisor： Professor Cheng Liang

Institute of Software, Chinese Academy of Sciences

June, 2022

中国科学院大学
研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论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。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：王化丽

日　　期：2022 年 5 月 27 日

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关保存和使用学位论文的规定，即中国科学院大学有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：王化丽　　导师签名：程亮

日　　期：2022 年 5 月 27 日　　日　　期：2022 年 5 月 27 日

摘　要

基于覆盖的灰盒模糊测试在发现程序漏洞、提高程序安全性方面发挥了巨大的作用。模糊测试使用随机性算法对程序进行分支探索以发现程序潜在的漏洞，但随机性算法在提高速度的同时包含了大量的无效操作。许多研究人员致力于提高模糊测试的准确性，但额外的性能开销使得模糊测试的总体效率难以提高。

针对上面讨论的问题，本文提出了一个基于聚类的模糊测试改进方法，并实现了原型工具-AgileFuzz。首先我们提出了关键位置的概念，并在变异过程中针对关键位置进行细粒度变异；然后为了降低关键位置的长度以便于 AgileFuzz 针对单字节的关键位置进行细粒度变异，本文基于 laf-intel 设计并实现了改进约束拆分插桩工具 lag-intel，能够将复杂约束拆分为单字节约束的同时避免巨大的 hash 冲突；最后在种子的能量分配阶段，AgileFuzz 提取种子的新覆盖和静态分支转移信息优化种子评分。

为了验证 AgileFuzz 能够高效地发现程序漏洞，我们与现有的模糊测试改进工具 AFL2.52b（-d）、MOPT 以及 EcoFuzz 进行比较，对 binutils、libxml2、harfbuzz 等开源程序进行了多次实验。实验结果表明，AgileFuzz 在相同时间内发现了更多的程序分支覆盖和程序漏洞，并且在测试过程中发现了 fontforge、harfbuzz 等开源软件最新版本中多个未知的漏洞，其中三个漏洞获得了 CNNVD 编号。

关键词：模糊测试，漏洞挖掘，聚类算法，静态分析

Abstract

Coverage-based graybox fuzzing(CGF) plays an important role in finding program vulnerabilities and improving program security. Graybox fuzzers use random algorithms to explore programs to find potential vulnerabilities, but random algorithms bring in a lot of invalid operations during fuzzing while improving fuzzing speed. Many researchers are committed to improving the accuracy of fuzzing, but the extra performance overhead makes it difficult to improve the overall efficiency of fuzzing.

To solve such problems, we propose an improved fuzzing method based on clustering and implement the prototype tool AgileFuzz. First, we propose the concept of key location and perform fine-grained mutate operations targeting key positions during fuzzing. Then to shorten the length of each key position for AgileFuzz to erform fine-grained mutation for single-byte key positions, we propose a compilation and splitting tool lag-intel, which can split complex constraints into single-byte constraints while avoiding huge hash conflicts. Finally,in the power scheduling stage of fuzzing, AgileFuzz extracts the new coverage and static branch transfer information of seeds to optimize the seed scoring strategy.

To verify that AgileFuzz can find program vulnerabilities efficiently, we compare the performance of AgileFuzz with state-of-the-art fuzzers, including AFL 2.52b (-d), MOPT, and EcoFuzz, and conducted several groups of experiments on binutils, libxml2, harfbuzz, and other open-source programs. Experimental result shows that AgileFuzz achieves higher program branch coverage and finds more program vulnerabilities within the same time, and found many unknown vulnerabilities in the latest versions of open-source software such as fontforge and harfbuzz, three of which are confirmed by CNNVDs.

Key Words: Fuzzing,Vulnerability mining, Clustering algorithm, Static analysis

目 录

图形列表

表格列表

第 1 章 引言

1.1 研究背景

1.1.1 软件安全研究的重要性

随着计算机和网络技术的发展，软件与我们的生活、工作的关系越来越密切。在生活方面，我们使用软件进行娱乐、通信；在工作中，我们使用软件进行数据处理、文本编辑。计算机的高效和便捷最终都需要通过软件得以体现，而软件复杂的功能，即使是软件的开发人员仍然无法确保软件按照正确的逻辑运行，从而导致软件中存在漏洞。在模糊测试首次被提出的论文[1]中提到“常用系统中可能会潜伏着严重的漏洞”，随着软件被广泛的使用，软件安全也变得越来越重要，一款通用的软件往往被数以亿计的设备使用，而如果软件出现漏洞，其导致的问题可能会使得全球的设备受到安全攻击，并因此造成巨大损失。比如：2010 年披露的震网蠕虫漏洞[2]是第一种攻击工业控制系统的病毒；2015 年 6 月，三星被爆出了高危的输入法漏洞。利用该漏洞，攻击者可以暗中监控用户的摄像头和麦克风、读取输入和传出的短信以及安装恶意应用程序，影响全球超过 6 亿的三星手机用户[3]。2015 年 7 月 Android 2.2 到 5.1 的版本中被爆出存在 Stagefright 高危漏洞，约 95% 的安卓设备受到该漏洞影响。利用 Stagefright 漏洞，攻击者只需向符合版本的安卓手机发送一条特殊的彩信，就可以完全控制用户手机[3]。2020 年 8 月，研究人员发现 CSP 绕过漏洞(CVE-2020-6519)，该漏洞使攻击者可以完全绕过 Chrome 73 版至 83 版的 CSP 规则，数十亿的用户可能会受到影响[4]。

如图 1. 1 所示，根据美国国家标准与技术研究所统计数据显示，2021 年总计报告 18554 个不同程序的安全漏洞，其中高危漏洞高达 3646 个。相较于 2020 年，漏洞总数增加 203 个。尽管软件潜在的漏洞被不断地挖掘出来，但是每年披露的漏洞数量还在增加，这主要是因为以下两个原因：1）软件规模过于庞大，其潜在的漏洞较多，随着漏洞挖掘方法的不断改进，越来越多隐藏的漏洞被挖掘；2）软件在不断的进行更新、迭代等，或者由于新的需求产生，新的软件不断涌现，但是新程序由于缺少足够的安全测试，存在漏洞的可能性更大。

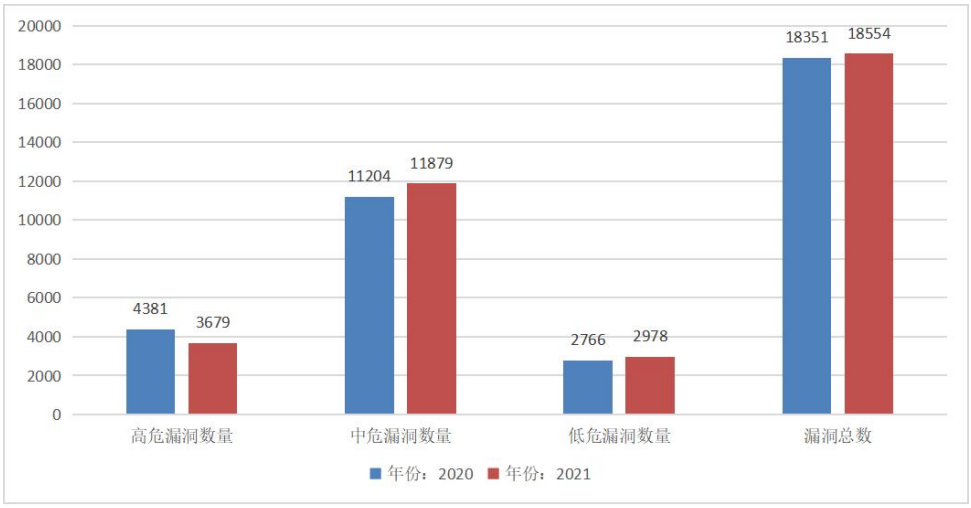

图 1.1 2021 年与 2020 年漏洞数量比较

Figure 1.1 Comparison of number of vulnerabilities in 2021 and 2020

1.1.2 现有软件安全分析技术概述

广泛存在的软件安全漏洞对国家和社会造成巨大的危害，因此软件安全一直是安全人员研究的重点。当前主流的程序安全分析方法有：污点分析、符号执行、代码审计和模糊测试等。如何高效地发掘软件可能存在的漏洞一直是这些主流安全分析方法研究的重点。这些方法的策略不同，存在不同方面的优点和局限性。

污点分析技术[5]是一种数据流分析技术，是一种通过分析程序数据传播的程序分析技术。污点分析一般可以分为动态污点分析和静态污点分析，静态污点分析技术的特点是在程序执行之前推断出污点源与目标代码的数据流关系，但是在分析非开源程序时，由于无法获取精确的程序静态信息，导致分析结果的精确性大大降低。动态污点分析的特点是通过真实地运行软件，并在程序运行过程中传播污点源数据。与静态污点分析相比，动态污点分析由于获取了真实的运行时数据，因而分析结果准确性更高，并且应用场景更加广泛。动态污点分析[6]由于程序执行与污点传播逻辑代码的紧密耦合，即污点传播代码与程序执行交织在一起，导致应用程序和污点传播代码之间频繁的“上下文切换”，最终污点分析在运行时具有很大的开销，难以实际分析规模较大的软件。

符号执行是一种常用的程序分析技术，符号执行可以分为静态符号执行、动态符号执行和混合符号执行。符号执行的特点是系统地探索程序可能的执行路径，而不需要具体的输入，通过抽象地将输入表示为符号，利用约束求解器来构造可

能满足条件的输入[6][7]。符号执行优势是能够以尽可能少量的程序测试用例访问更多的测试覆盖率。符号执行的缺点是处理像循环这样的语言结构时可能会成倍地增加执行状态的数量，从而出现路径爆炸的问题，在分析真实的软件中，往往难以达到理想的效果[8]。

代码审计是一种对源代码分析的技术，通过对源代码的全面分析，发现程序中可能存在的安全漏洞，但是代码审计非常依赖安全人员自身的经验[9]，自动化程度较低，难以对程序进行规模化分析，只能作为辅助分析技术。综上分析，符号执行等方法虽然通过提取丰富的代码细节达到程序分析的功能，但是由于其效率等多方面的原因，在程序安全性分析方面存在较多的局限。

模糊测试技术[1]是一种软件漏洞挖掘技术，通过随机变异原始种子或者基于规则生成种子文件，这些不同的种子构造程序多样的输入语料库，利用这些输入对程序进行安全性测试。其中典型的灰盒模糊测试工具-AFL[11]，通过静态代码插桩的方式使得模糊测试测试过程中能够获取程序运行时路径覆盖信息，并且这种插桩方式效率很高。综上，模糊测试在程序安全性分析方面弥补了以上方法的局限性[10]。

尽管模糊测试技术与其他安全研究技术相比，具有更大的优势。但是模糊测试本身由于其广泛使用随机性算法，在种子变异、种子筛选等阶段消耗了大量的时间在无效的操作或无效的种子。如何提高模糊测试的效率，降低其变异的盲目性成为了安全人员研究的热点。通过提高模糊测试的效率，使得安全人员能够更快地发现程序潜在的漏洞，降低软件在使用时带来的危害。

1.2 国内外研究现状

1.2.1 模糊测试概述

模糊测试[1]的概念最早于 20 世纪 90 年代提出，早期的模糊测试并非以程序安全性为目标，主要目的是评估代码的质量和可靠性。随着软件安全逐渐受到重视，安全人员尝试使用模糊测试技术进行软件安全性测试，其中 google、微软等大型的科技公司都对模糊测试投入了大量的科研力量和资金。

模糊测试是一种高效的软件测试技术，其核心思想是随机或者依据一定的规则生成程序的输入然后执行程序。在程序执行过程中监视程序行为，当程序异常时保留对应的输入，安全人员针对该输入进行程序异常复现以及崩溃分析，发现

程序可能存在的堆溢出、栈溢出等漏洞。模糊测试测试一般分为以下几个模块：种子生成、种子选择、程序反馈。其中程序反馈可以引导种子生成和种子选择，通过程序反馈可以极大地降低模糊测试的随机性，但是程序反馈也带来了巨大的运行时开销，降低了执行速率。

根据程序反馈的不同程序，模糊测试一般可以分为：白盒模糊测试、灰盒模糊测试和黑盒模糊测试。白盒模糊测试能够完整地提取程序运行时各种信息，引导模糊测试过程，进而有效地解决变异的盲目性，但是白盒模糊测试存在发现漏洞能力较弱、路径爆炸、约束求解复杂这些典型的问题[12][13]，这些问题目前缺乏有效的解决手段，限制了白盒模糊测试实际应用价值，漏洞挖掘更多使用黑盒和灰盒模糊测试。黑盒模糊测试[14]忽视对程序内部状态和结果进行分析，只需获取目标程序输入的数据格式等与程序运行时无关的信息。黑盒模糊测试适合分析输入数据高度结构化以及复杂难以分析的程序，典型的黑盒模糊测试工具如 Peach[15]和 DELTA[16]。同时对于某些闭源的程序，黑盒模糊测试也能够适用，这也是其主要优势之一。黑盒模糊测试的缺点在于：由于缺乏对目标程序的内部信息的提取，会产生大量的覆盖等价输入，导致对程序路径的覆盖率偏低，难以有效发现程序代码深层的缺陷和漏洞。灰盒模糊测试介于白盒模糊测试与黑盒模糊测试之间，因为提取程序有限的运行时的信息，从而可以使用轻量级的插桩方式。

根据种子生成策略的不同，模糊测试可以分为基于协议的种子生成技术和基于随机变异的种子生成技术。有些程序的输入存在严格的格式要求，如工业控制程序或者网络数据包解析协议等，如果通过随机变异的方式修改输入生成新的输入，较大的概率会破坏文件正常结构。待测程序以错误文件结构的种子作为输入时，会直接判断格式异常并直接退出程序。基于变异的种子生成技术针对大多数格式要求较低的程序，这种方式的好处是适用性较高，不需要针对程序的特定协议编写规则，同时基于变异的种子生成方式可以通过路径反馈情况，学习简单的格式信息，从而针对特定位置进行变异，提高有效变异的几率。

结合上述分析，本文研究的基于覆盖的灰盒模糊测试，其特点通过轻量级的编译时插桩，记录程序运行时分支覆盖情况，从而引导模糊测试，使得模糊测试能够高效、准确的对程序路径进行探索，在发现程序漏洞方面具有了较好的效果。

1.2.2 经典的基于覆盖灰盒模糊测试

基于覆盖的灰盒模糊测试[17][18]由于轻量级的程序运行时反馈，大大提高了模糊测试的效率，在发现漏洞方面起到了巨大的作用，并且受到了研究人员的广泛关注。如近年来优秀的模糊测试研究 AFL、FairFuzz[21]、EcoFuzz[20]、AFLFas[28]等都是这种结构模糊测试，灰盒模糊测试的主要工作流程如图 1.2 所示。

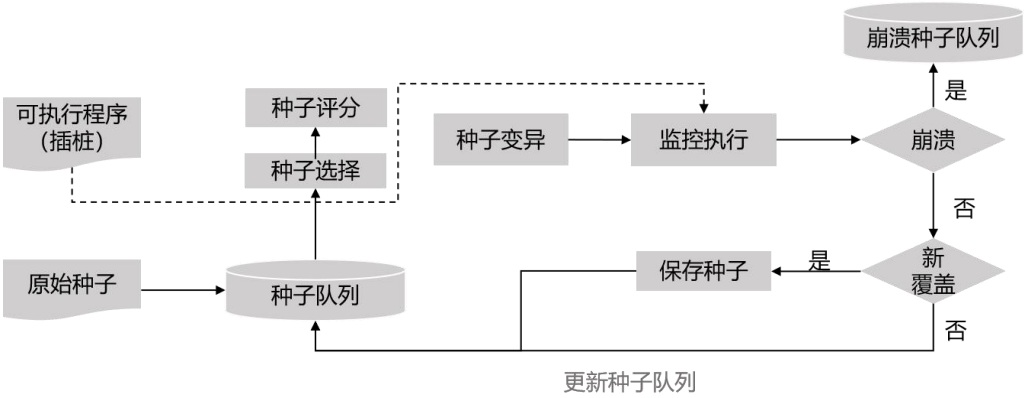

图 1.2 经典的灰盒模糊测试框架

Figure 1.2 Classic grey box fuzzing framework

从图中可以看出，基于覆盖的灰盒模糊测试主要包括以下几个部分：

1）程序插桩：程序插装模块可以分为源代码插桩和运行时插桩，其目的都是为了在程序执行时得到路径覆盖反馈。两种插桩方式不同之处在于，当待测程序不提供源码时，只能使用运行时插桩。根据不同的场景和需求，程序插桩可以分别以指令、基本块和函数为粒度进行插桩。插桩后的代码如果被执行，能够将执行信息反馈给模糊测试主进程，主进程根据反馈信息（是否执行了新的路径的种子）决定当前是否保存，并且程序插桩还可以记录不同基本块可能存在漏洞的概率（根据指令数量、内存读写情况等判断）。总之，通过程序插桩，模糊测试可以更准确地选择种子以及对种子进行变异。

2）种子选择：种子选择模块是模糊测试十分重要的模块。通过程序插桩的路径反馈机制，每个种子在执行后都会记录运行时路径信息，模糊测试根据该信息对种子进行评估，选择最有价值的种子保存在种子集合中，以便参与后续的模糊测试。

3）种子变异：种子变异阶段主要由模糊测试工具自定义的变异策略组成，

大部分模糊测试的变异策略并不针对特定程序的输入特征进行改进，而是考虑到通用性，采用固定的变异方式对种子进行变异。并且种子变异阶段可以通过插桩后的路径反馈信息，判断种子变异的位置是否有效或者变异后产生负面的效果，进而对种子变异位置进行调整，提高变异的有效性。

4）崩溃检测：崩溃检测模块与代码插桩反馈不同，监视的是程序执行时出现的异常行为。如果检测出执行阶段程序出现异常，模糊测试主进程会将当前种子单独保存在崩溃种子集合中。由于程序漏洞多种多样，进一步确定崩溃原因，则需要研究人员手动分析。

通过上述核心步骤，对程序进行重复测试，可以设置发现漏洞即退出模糊测试或无限重复执行直到用户手动停止。

1.2.3 现有的模糊测试改进策略

根据基于覆盖的灰盒模糊测试的基本流程，现有的相关研究改进大致有以下几个方面：对变异方式和变异位置的优化、对种子筛选和种子选择的优化和对能量分配策略的优化。

1.2.3.1 对变异方式和变异位置的优化

模糊测试在对种子进行变异时，一般分为包括确定性变异和非确定变异，确定性变异会对所有选中的种子的所有位置进行相同的变异，主要包括以下几种方式：

1）bitflip：对种子文件的不同长度进行翻转。

2）arithmetic：使用整数加/减算术运算对样本进行变异。

3）interest，使用特别数值替换对样本进行变异。

4）dictionary，使用测试过程中生成或用户提供的关键值替换/插入到样本中。

在确定性变异执行完成后，对种子进行非确定性变异。非确定性变异分为随机组合变异和拼接变异阶段，随机组合变异阶段对一个种子进行多次确定性变异，变异后达到组合变异的效果。拼接变异阶段选择两个种子切割后重新拼接生成新的样本，然后再对其进行随机组合变异。在对种子进行变异过程中，由于缺少对程序内部信息的分析，变异过程中大量的变异操作都是无效变异的。

许多基于覆盖的灰盒模糊测试改进研究对变异操作和变异位置进行了优化。MOPT[19]在实验过程中发现，确定性变异阶段消耗了模糊测试大量的变异时间，

并且总体效率较低，而非确定性变异则能够更快地产生有趣的种子，所以 MOPT 会选择性关闭确定性变异操作。EcoFuzz 为了更高的执行效率也选择直接跳过确定性变异阶段。MOPT 通过对不同的程序进行模糊测试实验发现，不同的变异操作产生新覆盖的概率是不同的，所以 MOPT 通过定制粒子群算法（PSO）优化算法从模糊测试有效性的角度寻找最优的变异操作，从而有效的提高变异效率。但是 MOPT 并没有提供新的变异方式，无法对种子进行更为多样性的变异。FairFuzz 通过计算掩码得到变异后仍然能够覆盖稀有分支的位置，从而使得更多的非确定性变异操作能够遍历到稀有分支；并且 FairFuzz 判断种子稀有分支是否已经进行了确定性变异，如果对应的稀有分支已经进行过确定性变异，则选择性跳过确定性变异阶段。Angora[22]通过可伸缩的字节级别的污点分析和梯度下降有效地解决了路径约束问题，提高了变异效率。Driller[23]利用符号执行生成满足复杂条件检查的种子，有效地提高了模糊测试的效率。Skyfire[24]考虑到输入格式高度结构化的程序，只对种子的特定位置进行变异。

1.2.3.2 对种子筛选和种子选择的优化

目前很多基于覆盖的模糊测试研究都针对种子选择进行了优化。种子筛选阶段主要决定变异后的种子是否需要保留，现有的策略大多以路径覆盖的唯一性进行保留，如果变异后的种子访问了新的路径（同一个分支执行的频率不同，也会被认为是新的路径），则会将该种子进行保留。

更多的研究改进主要集中在种子选择策略，种子选择策略从被保留下来的种子队列中选择更有价值的种子子集合进行变异。对于每个已经覆盖的程序分支，AFL 选择覆盖该分支、执行时间短且种子大小较小的种子，保存在 top_rated 集合中，然后从 top_rated 选择部分种子标记 favored，AFL 会优先变异标记为 favored 并且没有经过变异的种子。AFL 的种子选择策略只考虑了种子是否被变异过、种子的执行时间以及种子大小等因素进行选择，忽视了种子已经被选择的次数、路径覆盖信息等因素，导致选择了大量价值较低的种子。许多模糊测试改进对种子选择进行了进一步优化。FairFuzz 在执行过程中统计所有覆盖访问的频率，并计算访问频率最低的稀有分支；然后当每一个种子执行时，统计执行过程中是否访问了稀有分支，种子选择阶段会优先选择覆盖低频路径的种子，从而使得模糊测试能够更多次命中稀有分支。VUzzer[26]则认为访问更深路径的种子更容易发

现程序漏洞，在程序执行过程中，统计种子访问程序的最深路径，在种子选择阶段，优先选择到达更深路径的种子。Peiyuan Zong 等人提出了 FuzzGuard[27]，使用深度学习的方法过滤不能到达指定代码的种子。

1.2.3.3 对能量分配策略的优化

经典的灰盒模糊测试变异分为确定性变异和非确定性变异。其中非确定性变异更为重要，因为模糊测试通过对种子进行评分以及能量分配。种子的能量越多，在非确定性变异阶段就会进行更多次的变异，从而保证对更有可能发现漏洞的种子进行更多次变异。比如模糊测试工具 AFL 根据种子的大小，覆盖分支数、执行速度等因素对种子进行评分，分数越高的种子能量越大，在非确定性变异阶段变异的次数就会越多。但是这种能量分配策略缺少对程序内部执行状态和种子可能发现新覆盖的分析，将大量的变异时间花在没有价值的种子上，总体变异效率不高。

针对能量分配不合理的问题，许多研究人员对此作出了改进并开发出相应的改进工具。EcoFuzz 使用多臂老虎机（muti-armed bandit）理论对模糊测试进行更准确的建模以完成能量分配。AFLGo[25]和 Hawkeye[29]是导向型的灰盒模糊测试工具，通过静态代码插桩计算当前执行的种子距离目标代码的距离，确保距离目标代码越近的种子会被赋予更多的能量。AFLFast 观察到基于覆盖的灰盒模糊测试（CGF）产生了过多的具有类似覆盖的输入，所以 AFLFast 通过马尔科夫链模型将模糊测试更多的能量集中在低频路径，更少的能量分配给高频路径。

1.3 现有工作不足以及本文出发点

1.2.3 节讨论的模糊测试工具的改进大多基于经典的 AFL 框架进行改进，虽然针对 AFL 的多个模块做了优化，取得了一定的改进效果，但是 AFL 采用了大量随机性算法，导致在种子选择、变异等阶段浪费了大量的资源，总体效率不高，主要包括以下几个方面。

1.3.1 变异方式不够细粒度

定义一：我们所讨论的细粒度变异强调的是变异后结果的细粒度，细粒度变异能够将选中位置的每个字节变异出所有可能的结果，即每个字节都能变异出 0-255 的 ASCII 值。

现有的变异策略由于考虑到细粒度变异带来的巨大开销，而选择使用粗粒度变异策略，例如：通过对比特位、字节等长度内容进行位翻转、字节翻转等操作对种子进行变异。虽然比特位变异的位置粒度最低，但是经过变异后只能变异出少数的结果。模糊测试并未针对程序进行复杂求解，在遇到复杂约束条件语句时，无法针对特定约束值变异出正确的结果。典型的模糊测试工具，如 FairFuzz、AFLFast 等模糊测试改进，都是直接采用了 AFL 原有的确定性变异策略，导致即使通过种子选择优化策略选中了有价值的种子并变异到影响特定约束语句的字段位置，但是仍然可能无法变异成目标结果。其它模糊测试工具，如：MOPT，在变异方式做出了改进，主要体现在非确定性变异阶段，通过粒子群算法学习最优的变异操作，但是通过算法选择变异方式只能缩小变异操作的选择范围，降低无效变异的概率，但是无法做到细粒度变异。

1.3.2 关闭确定性变异导致变异的盲目性

以 MOPT、EcoFuzz 为典型的模糊测试改进工具，在大量的实验过程中发现，确定性变异消耗了大量的变异时间，但变异的效率却较低。为了对种子进行高效地变异，这些模糊测试工具跳过了确定性变异阶段，并在实验中取得了较好的实验结果。但是只使用非确定性变异导致变异策略更加盲目，整个变异过程完全随机选择变异位置和变异操作，导致以下两个问题：1）难以对种子的关键字段（我们定义：通过变异某个连续的位置能够影响程序路径，这样的连续位置称为关键字段）进行细粒度地持续性变异；并且 2）非确定性变异是将多个不同的确定性变异进行组合，组合变异的结果可能会造成单个变异之间相互影响，导致即使组合位置中某一个位置变异成有效结果，但是仍然无法产生新覆盖。所以，通过直接关闭确定性变异提高模糊测试效率并不合理，但是保留确定性变异又使得模糊测试效率较低。

1.3.3 难以求解复杂约束

很多程序的条件分支语句是长字符串匹配或长整数匹配等复杂的约束求解型语句，现有的传统灰盒模糊测试工具无法有效地针对复杂约束进行求解，而一些针对复杂约束求解的模糊测试工具则带来了巨大的性能开销，影响模糊测试总体效率。程序复杂的条件约束分为两种情况：1）复杂的条件约束由多个简单约束组成，这种情况下模糊测试理论上可以通过变异种子，逐步满足单个约束，最

终得到满足全部约束的种子。但是由于模糊测试的随机性，难以针对特定位置进行持续性变异，导致几乎很难变异成最终约束的种子。2）复杂的条件约束语句使用类似 strcmp 系统调用，无法通过逐步变异单个字节求解。这种情况下，相关研究人员研究出 Driller[23]等借助符号执行的模糊测试工具，能够针对复杂约束进行求解，但是这种方式的运行时开销较大，导致模糊测试的整体效率较低。其它一些研究，如 laf-intel[30]能够将程序的复杂约束分解为单字符约束组合，但是约束拆分的同时带来了大量的 hash 冲突，从而严重影响了模糊测试的执行效率。

1.3.4 种子能量分配依赖机器学习算法

种子的能量决定了种子在非确定性变异时变异的次数。现有的模糊测试改进的种子能量分配阶段优化依赖机器学习算法进行建模，但是忽略了种子的新覆盖数量和静态分析信息对种子进一步发现新路径的影响。典型的模糊测试工具，如：EcoFuzz、AFLFast 等工具，通过遗传算法对种子发现路径能力得到反馈进而学习，但是无法给予种子合适的初始能量；并且这些模糊测试工具依赖遗传算法，而没有提取程序的静态分支信息，导致评分方式单一，并不能有效对种子进行评分。

1.4 本文研究内容

综上所述，本文认为现有的模糊测试研究缺少细粒度的变异方式，缺少对变异策略的研究，无法针对复杂条件约束进行快速求解，种子的能量分配策略也缺失更为合理的计算方式，依赖机器学习算法，而忽略了种子覆盖信息和程序静态分支信息。因此本文的研究内容主要包括以下几个方面：

1）本文设计并实现了**基于关键位置的细粒度变异策略**，针对关键位置的变异策略使得 AgileFuzz 减少了其在无效的变异位置上消耗的时间，**提高了变异效率**。细粒度变异能够保证变异结果的全面性，**提高了变异的准确性**。

2）本文设计并实现了**基于聚类的关键位置确定算法**，使用执行反馈确定种子位置是否关键的策略**保证了关键位置的准确性**，结合聚类算法则**提高了确定关键位置的效率**。

3）本文设计并实现了基于 laf-intel 的**改进约束拆分覆盖反馈策略**，并实现了

对应的工具-lag-intel。lag-intel 拆分约束后使得关键位置的长度为 1，**提高了针对关键位置细粒度变异的效率**；lag-intel 的双 bitmap 路径保存方式**降低了**新增约束基本块带来的 **hash 冲突**；针对不同条件约束进行不同的插桩方式**提高了变异的效率**。

4）本文设计并实现了种子筛选优化策略，该策略在优先选择包括关键位置的种子前提下，选择种子执行时间*种子大小更小的种子。

5）本文设计并实现了静态分支转移提取模块以及结合静态分支信息和新覆盖的种子评分策略。通过这种种子评分策略，保证存在双分支转移的新覆盖越多的种子评分越高，从而将更多的能量集中在更有价值的种子，**提高变异效率**。

6）本文基于上述方法，设计并实现了灰盒模糊测试工具 AgileFuzz。在真实程序中验证后，AgileFuzz 能够更快地发现程序漏洞和程序覆盖。并且在实际漏洞挖掘过程中，发现多个程序最新版本存在的漏洞，其中三个漏洞获得了 CNNVD 编号。

1.5 本文组织结构

本文共分为 6 章，每个章节的内容概要如下：

第一章，引言。本章主要介绍研究背景、国内外研究现状、现有工作不足以及本文出发点、本文研究内容和本文组织结构。研究背景部分论述软件安全的重要性，进而介绍常用的软件安全研究方法以及其优缺点，最后说明模糊测试在软件安全方面的优势。国内外研究现状部分主要阐述了灰盒模糊测试的优势并介绍了近年来针对灰盒模糊测试的相关研究。现有工作不足以及本文出发点部分则主要介绍现有工作的局限性以及本文改进的目的。本文研究内容部分则概述了本文针对灰盒模糊测试改进的核心思路、方案。本文组织结构部分介绍了文章各个部分的组织结构。

第二章，相关理论和技术。这一章主要介绍了论文涉及的主要技术，包括 AFL 的核心技术、静态分析、约束拆分插桩以及聚类算法。

第三章，基于关键位置的模糊测试改进。本章详细介绍上节提到的本文 6 点研究内容，3.1 节介绍了灰盒模糊测试工具 AgileFuzz 的整体架构，3.2 节介绍了改进约束拆分覆盖反馈策略，3.3 节介绍了种子筛选优化策略，3.4 节介绍了基于聚类的关键位置确定算法，3.5 节介绍了基于关键位置的细粒度变异策略，3.6

节介绍了新覆盖与静态分支信息结合的种子评分策略。

第四章，模糊测试工具实现。本章主要介绍 AgileFuzz 的实现过程，包括使用到的技术、工具等。

第五章，分析和评估。本章主要从通过多个实验验证 AgileFuzz 改进的有效性。结合程序真实运行时逻辑，解释 AgileFuzz 如何更快地发现程序覆盖以及程序潜在漏洞。

第六章，总结与展望。本章主要总结了本文的研究内容，并对本文的成果进行分析和总结，同时分析了本文仍然存在的问题以及下一步的工作计划。

第 2 章　相关理论和技术

2.1　以 AFL 为代表的灰盒模糊测试技术细节

灰盒模糊测试，尤其是 AFL 自其出现以来，一直受到安全人员的广泛关注。相关研究人员在 AFL 基础上进行了很多研究工作，并且这些研究工作都取得了巨大的成果。这说明灰盒模糊测试在软件安全性方面具有巨大的潜力和优势。因此本文的研究工作也是在灰盒模糊测试的基本框架进行。本节以 AFL 为例介绍灰盒模糊测试的基本工作流程和重要的技术细节，主要涉及程序插桩、覆盖反馈、种子变异、能量分配以及种子筛选。

2.1.1　工作流程

AFL 的主要工作流程与 1.2.2 节介绍的基于覆盖的灰盒模糊测试结构相比，补充了重要的模块细节，主要包括程序插桩和覆盖反馈、种子变异、能量分配和种子筛选。

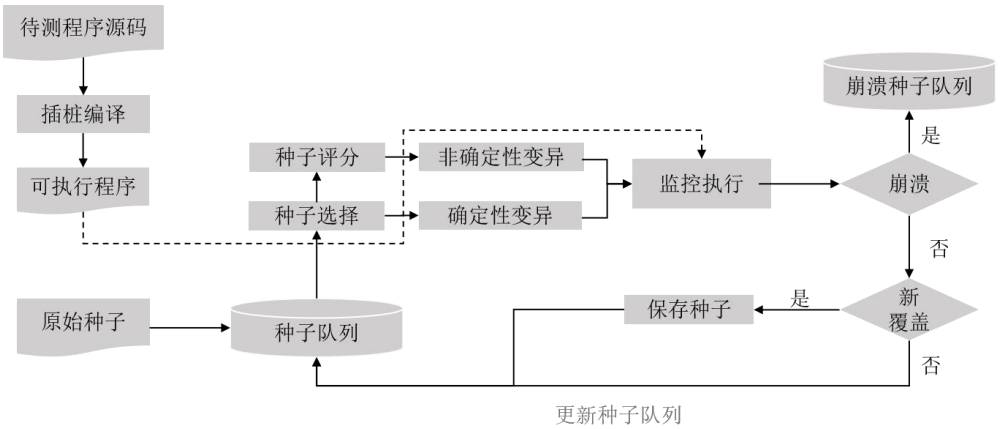

图 2.1　AFL 的工作流程

Figure 2.1　AFL workflow

2.1.2　程序插桩与覆盖反馈

AFL 有两种插桩方式。分别针对有源码的程序和没有源码的程序。对于有源码的程序，AFL 使用 gcc 插桩或者 llvm 插桩。对于没有源码的程序，AFL 使用 qemu 实现运行时插桩。这两种插桩方式逻辑相同，都是粗粒度的插桩，统计的是基本块之间的转移。

AFL 在编译程序时，以基本块为单位进行插桩。在编译每个基本块之前，首

先会产生一个随机数，该随机数作为参数与对应的插桩算法写入到基本块中。如表 2.1 所示，插桩算法的功能是将当前基本块的随机数和上一个执行基本块的随机数进行异或操作（代码行：4），异或的结果记录在共享内存对应的偏移位置（代码行：6），不同的偏移对应不同的程序分支编号。共享内存是 AFL 进行覆盖反馈的主要方式，AFL 主进程维持一块共享内存，每次测试开始前，将共享内存重置，待测程序执行完后，共享内存记录执行过程中访问的程序分支转移编号。AFL 在将相邻的基本块随机数进行异或操作时，会将前一个基本块的随机数向右一位（代码行：5），这样做的目的是如果同一个基本块发生循环指向时，避免记录的程序路径值为 0。

表 2.1 AFL 插桩算法

Table 2.1 AFL Instrumentation Algorithm

算法 1	AFL 插桩算法
输入	待插桩基本块：basic_block，随机数：random 以及共享内存：shared_memory
输出	插桩后的基本块
1:	procedure instrumentation_normal(random,shared_memory,basic_block)
2:	prev_loc=get_afl_prev_loc()
3:	insert_num(random)
4:	branch=prev_loc^random
5:	set_afl_prev_loc(random>>1)
6:	shared_memory[branch]=shared_memory[branch]+1

总的来说，AFL 所采用的粗粒度插桩和覆盖反馈方式有以下几个特点：

1）模糊测试主进程与待测程序进程之间使用共享内存进行数据交互，相较于其它进程间通讯方式，速度较快且稳定。共享内存的大小为 65536 个字节，最多保存 65536 个不同的分支覆盖。

2）使用随机数插桩，每个基本块插桩时，产生[0,65535]范围的随机数。

3）使用单字节大小的变量保存程序路径，记录程序执行过程中访问的程序分支转移以及分支访问的频率，但不记录程序执行的分支转移顺序。

2.1.3 种子变异

AFL 的变异策略由确定性变异和非确定性变异组成。其中确定性变异由多种

变异操作，在变异阶段对种子的所有位置进行变异。

表 2.2 AFL 的变异策略

Table 2.2 Mutation strategies for AFL

变异方式		描述
确定性变异	flip	位翻转
	arithmetic	算术操作
	interest	特殊的内容进行填充、替换
	dictionary	用户提供的字典进行填充、替换
非确定性变异	havoc	将多个确定性变异随机组合，对多个随机位置进行变异
	splice	将两个种子拼接为一个新的种子，然后执行 havoc 变异

从表 2.2 可以看出，AFL 的变异方式以预定义的操作为主，用户也可以提供程序的字典文件，使得 AFL 变异时能够对程序进行针对性变异。AFL 在实际测试中，需要用户提供指定的测试用例集合，然后针对这些初始用例进行变异产生新的种子。所以 AFL 的变异方式对整个测试过程具有十分重要的影响，下面我们详细介绍 AFL 的变异方式。

1）flip

表 2.3 flip 变异策略

Table 2.3 flip mutation strategy

flip 变异策略	变异策略解释
bitflip 1/1	翻转长度为 1 个 bit，移动长度为 1 个 bit
bitflip 2/1	翻转长度为 2 个 bit，移动长度为 1 个 bit
bitflip 4/1	翻转长度为 4 个 bit，移动长度为 1 个 bit
bitflip 8/8	翻转长度为 1 个 byte，移动长度为 1 个 byte
bitflip 16/8	翻转长度为 2 个 byte，移动长度为 1 个 byte
bitflip 32/8	翻转长度为 4 个 byte，移动长度为 1 个 byte

flip 变异以翻转作为变异方式，对单个比特位、双比特位和字节（四比特位）执行翻转操作。flip 变异的优点是变异位置粒度细，能够对单个字节的每个比特位进行变异，缺点是变异的方式过于单一，变异后只能产生翻转后的一种结果。例如：单比特位翻转能够使得字节大小的内容产生 8 种新的结果，但是每个字节

存在 256 种结果。为了让 flip 变异产生更多的变异结果，AFL 设置多种不同长度的 flip 变异方式，具体如表 2.3 所示。

2）arithmetic

arithmetic 变异策略是对变异字段进行算术运算。AFL 自定义了 ARITH_MAX 变量，在进行算术运算时，ARITH_MAX 变量决定了算术运算的范围。对于变异的字段，AFL 会对该字段进行 0-ARITH_MAX 范围的加和减操作。相比于 flip 运算，arithmetic 以字节为最小的变异单位，但是能够将字段变异成更多可能的结果，具体如表 2.4 所示。

表 2.4 arithmetic 变异策略

Table 2.4 arithmetic mutation strategy

arithmetic 变异策略	变异策略解释
arithmetic 8/8	运算长度为 1 个 bit，移动长度为 1 个 bit
arithmetic 16/8	运算长度为 2 个 bit，移动长度为 1 个 bit
arithmetic 32/8	运算长度为 4 个 bit，移动长度为 1 个 bit

3）Interest

表 2.5 interest 变异策略

Table 2.5 interest mutation strategy

interest 变异策略	变异策略解释
interest 8/8	替换长度为 1 个 bit，替换集合为 INTERESTING_8，移动长度为 1 个 bit
interest 16/8	替换长度为 2 个 bit，替换集合为 INTERESTING_8 和 INTERESTING_16，移动长度为 1 个 bit
interest 32/8	替换长度为 4 个 bit，替换集合为 INTERESTING_8、INTERESTING_16 和 INTERESTING_32，移动长度为 1 个 bit

flip 和 arithmetic 变异都是在种子原有结果的基础上进行修改，达到变异的效果，而 interest 变异则是直接将字段替换为有趣的内容。AFL 定义了三种有趣的值集合，分别为 INTERESTING_8、INTERESTING_16 和 INTERESTING_32，

分别针对不同长度的变异字段，具体如表 2.5 所示。

4）dictionary

dictionary 变异的特点是由用户提供程序特有的字典文件，保存在 extras 数组中。AFL 在确定性变异和非确定性变异阶段，将 extras 数组的内容替换或者插入到待变异的种子中。dictionary 优点是能够针对不同程序提供针对性的字典文件，更快地变异符合程序约束地种子，但是前提是该程序已有字典或存在提取字典的程序。

5）Havoc

表 2.6 havoc 变异算法

Table 2.6 havoc mutation algorithm

算法 2 havoc 变异算法

输入 种子以及种子的能量：q,stage_max

输出 无

```
1:  procedure havoc_mutation(q,stage_max)
2:    for stage_cur in stage_max do
3:      use_stacking=random(1,1<<HAVOC_STACK_POW2)
4:      for i in use_stacking do
5:        switch random(15) do
6:          case 1:
7:            flip_byte(q)
8:          Break
9:          case 2:
10:           interesting_set(q)
11:         break
12:       case ...
13:         ...
14:     end switch
15:   end for
16:   run_target(q)
```

haovc 变异由 15 种确定性变异操作组成。在执行 havoc 变异时，主要使用随机性算法进行操作和位置的选择。如算法 2 所示，AFL 会产生一个随机数

-use_stacking，该随机数的范围是 1-HAVOC_STACK_POW2 的平方。与种子能量不同，use_stacking 决定了非确定性组合变异操作的次数。haovc 变异的优点是效率较高，缺点是多位置随机变异可能产生大量错误结构的种子。

6）splice

splice 变异核心操作是将当前种子与其他种子进行拼接，生成新的种子。新种子并不会直接运行或者保存，而是对新种子进行 havoc 变异。Splice 变异能够对种子进行结构化的破坏和重组，使用 splice 变异可以生成小范围变异策略难以产生的新覆盖。

表 2.7 AFL 能量分配算法

Table 2.7 AFL energy allocation algorithm

算法 3 AFL 能量分配算法

输入 需要分配能量的种子：q

输出 种子的能量大小：stage_max

```
1:  procedure energy_scheduling(q)
2:      stage_max=q->perf_score / total_seed / 100
3:      if  splice_cycle then
4:          if deter_done then
5:              stage_max=stage_max*1024
6:          else
7:              stage_max=stage_max*256
8:          end if
9:      else
10:         stage_max=stage_max*32
11:     end if
12:     if stage_max< HAVOC_MIN then
13:         stage_max= HAVOC_MIN
14:     end if
```

2.1.4 能量分配

能量分配阶段是模糊测试过程中十分重要的环节。当确定性变异执行完后，AFL 会重复执行非确定性变异，能量分配的大小（AFL 记作 stage_max）决定每个种子非确定性变异的次数。AFL 根据种子的执行时间、种子大小、覆盖的路径数量、种子深度对种子进行评分，种子评分的结果（AFL 记作 perf_score）越

高，种子分配的能量越大，同时种子是否执行过确定性变异，当前是否是 splice 变异都会影响种子能量分配的结果，能量分配的具体策略如表 2.7 所示。

通过算法 2 描述可以看出，能量分配的初始值来自于种子评分的结果，AFL 根据以下几个原则计算种子的分数：

1）种子的执行时间越长，评分越低。

2）种子的长度越长，评分越低。

3）种子覆盖的路径越少，评分越低。

4）种子的深度越低，评分越低。

表 2.8 AFL 种子评分策略-根据种子执行时间

Table 2.8 AFL Seed Scoring Strategy - Based on Seed Execution Time

算法 4 AFL 种子评分策略-根据种子执行时间

输入 需要评分的种子：q

输出 种子的评分结果：q->perf_score

```
1:      procedure calculate_score_time(q,avg_exec_us)
2:          if q->exec_us * 0.1 > avg_exec_us than
3:              q->perf_score=10
4:          else if q->exec_us * 0.25 > avg_exec_us than
5:              q->perf_score=25
6:          else if q->exec_us * 0.5 > avg_exec_us than
7:              q->perf_score=50
8:          else if q->exec_us * 0.75 > avg_exec_us than
9:              q->perf_score=75
10          else if q->exec_us * 4 < avg_exec_us than
11:             q->perf_score=300
12:         else if q->exec_us * 3 < avg_exec_us than
13:             q->perf_score=200
14:         else if q->exec_us * 2 < avg_exec_us than
15:             q->perf_score=150
16:         end if
```

AFL 默认将种子的初始评分（perf_score）设置为 100，然后进行调整。表 2.8、表 2.9 和表 2.10 的算法具体介绍 AFL 针对种子的评分进行的几种不同方面的调整。

表 2.9 AFL 种子评分策略-根据种子路径覆盖情况

Table 2.9 AFL Seed Scoring Strategy - Based on Seed Path Coverage

算法 5 AFL 种子评分策略-根据种子路径覆盖情况

输入 需要评分的种子：q

输出 种子的评分结果：q->perf_score

```
1:    procedure calculate_score_bitmap(q,avg_bitmap_size)
2:      if q->bitmap_size * 0.3 > avg_bitmap_size than
3:        q->perf_score=q->perf_score*3
4:      else if q->bitmap_size * 0.5 > avg_bitmap_size than
5:        q->perf_score=q->perf_score*2
6:      else if q->bitmap_size * 0.75 < avg_bitmap_size than
7:        q->perf_score=q->perf_score*1.5
8:      else if q->bitmap_size * 3 < avg_bitmap_size than
9:        q->perf_score=q->perf_score*0.25
10:     else if q->bitmap_size * 2 < avg_bitmap_size than
11:       q->perf_score=q->perf_score*0.5
12:     else if q->bitmap_size * 1.5 < avg_bitmap_size than
13:       q->perf_score=q->perf_score*0.75
14:     end if
```

表 2.10 AFL 种子评分策略-根据种子深度

Table 2.10 AFL seed scoring strategy - based on seed depth

算法 6 AFL 种子评分策略-根据种子深度

输入 需要评分的种子：q

输出 种子的评分结果：q->perf_score

```
1:    procedure calculate_score_depth(q)
```

续表 2.10 AFL 种子评分策略-根据种子深度

算法 6 AFL 种子评分策略-根据种子深度

输入 需要评分的种子：q

输出 种子的评分结果：q->perf_score

```
2:    if q->depth >=0 and q->depth <=3 than
3:        break
4:    else if q->depth >=4 and q->depth <=7 than
5:        q->perf_score=q->perf_score*2
6:    else if q->depth >=8 and q->depth <=13 than
7:        q->perf_score=q->perf_score*3
8:    else if q->depth >=14 and q->depth <=25 than
9:        q->perf_score=q->perf_score*4
10:   else
11:       q->perf_score=q->perf_score*5
12:   end if
```

2.1.5 种子筛选

AFL 保存了所有路径不同的种子，因此保存的种子过多，存在大量的冗余。为了测试的高效性，AFL 无法对每一个种子进行变异，因此 AFL 设计了种子筛选策略以选择更优的种子集合。

表 2.11 top_rated 集合更新算法

Table 2.11 top_rated set update algorithm

算法 7 top_rated 集合更新算法

输入 top_rated 集合、新种子 q

输出 更新后的 top_rated 集合

```
1:    procedure update_bitmap_score(q,top_rated)
2:    q_factor=q->exec_us * q->len
3:    for i in MAP_SIZE do
4:        if top_rated[i] != 0 then
```

续表 2.11 top_rated 集合更新算法

算法 7 top_rated 集合更新算法

输入 top_rated 集合、新种子 q

输出 更新后的 top_rated 集合

```
5:              temp_factor= top_rated[i]->exec_us * top_rated[i]->len then
6:              if q_factor > temp_factor then
7:                  continue
8:              else
9:                  free(top_rated[i])
10:             end if
11:             top_rated[i]=q
12:             bitmap_score_changed=1
13:         end if
14:     end for
```

表 2.12 AFL 种子筛选算法

Table 2.12 AFL seed screening algorithm

算法 8 AFL 种子筛选算法

输入 所有种子队列：queue

输出 标记为 favored 的种子队列:queue

```
1:      procedure cull_queue(queue)
2:          q=queue
3:          bitmap_containted=[]
4:          while q != None do
5:              q->favored=0
6:              q=q->next
7:          end while
8:          for i in MAP_SIZE do
9:              if top_rated[i]!=0 and bitmap_containted[i]==0 then
10:                 top_rated[i]->favored=1
```

续表 2.12　AFL 种子筛选算法

算法 8 AFL 种子筛选算法
输入 所有种子队列：queue
输出 标记为 favored 的种子队列:queue
11:　　update_bitmap_containted(bitmap_containted)
12:　　end if
13:　end for

AFL 维持一个全局的种子集合-top_rated，该集合容量为 65536，与记录路径的集合 bitmap 相同。当产生一个新种子时，AFL 根据表 2.11 所示的算法对 top_rated 集合进行更新，top_rated 集合更新的核心策略是计算种子执行时间和种子长度（代码行：2,5）。top_rated 集合更新策略的依据是种子执行时间越短，在一定时间能够执行更多次变异，种子长度较小，更容易对种子进行变异。top_rated 每个元素都是一个指针，如果元素不为空，则该元素记录着每条已经访问的分支覆盖对应执行速度较快且种子较小的种子，集合中索引为 i 的位置，表示访问了 bitmap 对应索引为 i 的分支覆盖并且执行速度快且种子较小的种子。在种子筛选阶段，AFL 从 top_rated 集合中选择更少且包括已有分支的集合，对应的种子标记为 favored。

2.2　程序静态分析

程序静态分析是指在不运行程序的前提下，对程序的语义信息、语法信息、数据流、控制流等进行分析。静态分析相较于动态分析，其优点是能够提取更多的程序信息，获取动态执行难以执行到的程序信息。其缺点是，很多程序语句存在间接跳转，其依赖于动态值，所有静态分析难以判断程序实际执行的路径。常用的静态分析包括：模型检测、符号执行、基于规则的检查和基于机器学习的漏洞检测。

2.2.1　模型检测

模型检测[31]是一种自动验证有限状态是否满足规范的方法，其原理是通过显示状态搜索或者隐式不动点进行计算，其广泛应用于软件的可信性验证。由于模型检测是基于对状态空间的搜索，所以其存在状态爆炸的问题。针对状态爆炸

问题，模型检测技术有四种策略进行优化，分别为：1）随机化和启发式搜索[32]、2）状态空间压缩[33][34][35]、3）存储空间压缩[36][37][38][39]以及分布式和并行计算[40][41][42]。

2.2.2 符号执行

符号执行技术是一种重要的程序分析技术，其通过将程序输入变量看作符号，程序计算的输出看作输入程序输入的函数，然后对函数进行求解。早期的符号执行由于难以解决路径穷举遍历，无法保证分析结果的完备性。近年来，符号执行借助智能搜索策略[43]、内存模型优化、Z3 等 SMT 求解技术[44]的改进，大大提高了约束求解的能力。

2.2.3 基于规则的检查

不同的程序中，往往包含一些特殊的规则。如先打开文件，再读取内容；读取文件后必须将文件进行关闭等。这些规则往往基于开发者的经验，然后将其转换为计算机能够识别的内部表示，进而对静态程序进行分析，判断其是否符合规则。

2.2.4 基于机器学习的漏洞检测

基于机器学习的漏洞检测技术分为三类：1）监督学习、2）无监督学习以及3）强化学习。监督学习通过对标记的样本进行建模，学习漏洞的特征得到漏洞检测模型，然后根据该模型对程序进行检测。无监督学习的特点是训练的样本并没有标记，学习的目的是识别给定数据集的结构特征。强化学习的特点是在训练过程中存在互动，通过奖励和惩罚机制使得学习模型达到某个预期。

2.3 LLVM 编译插桩与复杂约束拆分

2.3.1 LLVM 编译插桩

LLVM[45]是构架编译器的框架结构，能够对程序语言的编译器优化、链接优化、在线编译优化、代码生成。编译器通常包括前端和后端，前端包括：词法分析、语法分析、语义分析和中间代码生成，后端包括：生成机器码。GCC 编译器其前端与后端耦合性太高，难以支持新的编程语言，相比之下 LLVM 由于其解耦的前后端结构，现在被作为实现各种静态和运行时编译语言的通用基础结

构。

AFL 支持 GCC 和 LLVM 对程序进行编译插桩。与 GCC 编译插桩不同的是，LLVM 能够在编译程序将代码转化为中间语言，从而便于对程序进行优化，LLVM 插桩后的程序执行效率一般优于 GCC 插桩后的程序，并且使用 LLVM 插桩，还可以获取程序更多的信息，如分支转移信息、函数调用序列等，便于对程序做出更多的插桩改进策略。

2.3.1 复杂约束拆分

约束拆分是将程序较为复杂的匹配语句拆分为多个简单的匹配。laf-intel 是基于 LLVM 的约束拆分工具。对于有源码的程序，laf-intel 在编译程序的同时，搜索程序中包括复杂约束的代码，然后修改程序原有逻辑-将长约束拆分为多个单字符或数值匹配。经过拆分后的程序，模糊测试在求解原复杂约束时，变异出单个匹配内容后能够得到反馈。

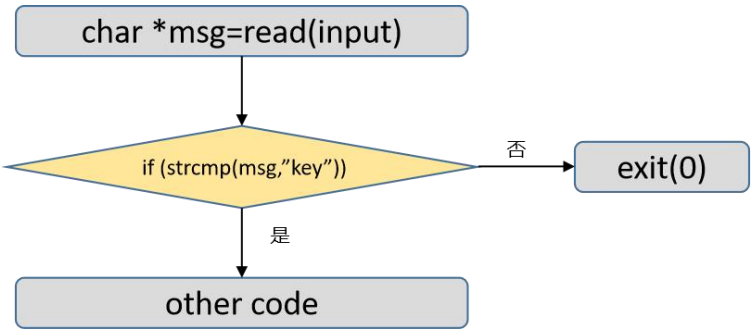

图 2.2 strcmp 长字符串约束示例

Figure 2.2 strcmp example of long string constraints

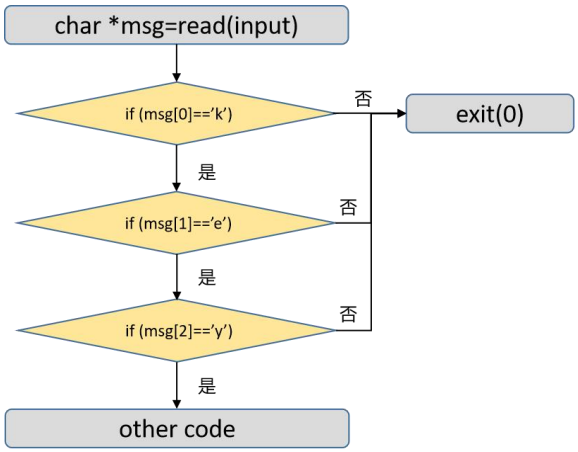

图 2.3 laf-intel 拆分 strcmp 为单个字符匹配示例

Figure 2.3 laf-intel split strcmp into single character matching example

如图 2.2 所示为典型的 strcmp 约束语句，只有生成内容完全匹配的字符串才能走到对应的分支，但是现有的粗粒度变异策略几乎不可能变异出满足要求的长字符串。

表 2.13 laf-intel 约束拆分算法

Table 2.13 Laf-intel constrained splitting algorithm

算法 9 约束拆分算法

输入 待拆分基本块 basic_block

输出 拆分后基本块集合 basic_block_list

```
1:    procedure laf_constraint_split(basic_block)
2：     for instrtion in basic_block do
3:       if  type(instrtion) is strcmp then
4:         remove(instrtion)
5:         string=get_string(instrtion)
6:         for i in len(string) do
7:           insert_cmp_instrtion(string[i])
8:         end for
9:       elif  type(instrtion) is compare then
10:         remove(instrtion)
11:         num=get_num(instrtion)
12:         for i in len(num) do
13:           insert_compare_instrtion(num[i])
14：        end for
15:      elif  type(instrtion) is switch then
16:         remove(instrtion)
17:         do_similarity_with_strcmp_or_compare(instrsion)
18:      end if
19:    end for
```

laf-intel 通过表 2.13 所示的约束拆分算法，可以对这样的长约束语句进行拆分，最终得到图 2. 3 所示的程序。模糊测试对拆分约束后的程序进行插桩时，每个单字符匹配基本块都会进行插桩。当模糊测试对该程序进行测试时，变异出单

个字符匹配后，匹配单个字符的种子就会被保存，该种子参与后续模糊测试时仍然通过变异，生成结果匹配下一个字符要求的新种子，循环迭代直到满足所有字符要求。

从拆分前和拆分后的程序对比可知，laf-intel 在拆分长约束后得到的新程序具有更多的指令和基本块，新增加的指令和基本块不仅影响程序执行的速度，也会影响模糊测试的插桩、路径覆盖统计、种子评分等各个阶段，导致程序在拆分复杂约束后并没有提高模糊测试的总体效率。

2.4 聚类算法

聚类算法是一种运用广泛的数据分析计算方法，常用作将离散的点进行分类、聚合，将相似的对象归为一类，不相似的对象归为不同类。与分类算法不同，聚类是一种无监督学习，能够发现数据结构之间隐藏的关系，在图像分割、用户画像、固有结构认知等方面广泛运用。

由于应用场景不同，聚类算法分为多种不同类别的算法，主要包括以下几种类型：1）基于划分的聚类算法，2）基于层次的聚类算法，3）基于密度的聚类算法，4）基于网格的聚类算法。划分式聚类算法需要事先指定簇类的数目或者聚类中心，然后反复进行迭代，直到簇内点足够近，簇间的点足够远。常用的基于分为的聚类算法有 k-means、kernel k-means 等。基于层次的聚类算法分为自上而下和自下而上，自下而上聚类将每个数据点看作单一的簇，然后连续合并两个簇，直到所有的簇合并成一个包含所有点的簇。基于密度的聚类根据样本的密度分别进行聚类，根据样本之间的可连接性不断扩展进行聚类。常用的基于密度的聚类算法有 DBSCAN[46]、MeanShift[47]等。基于网格的聚类算法将离散点看作独立且有限的单元，所有单元形成一个网格结构，所有点的聚类操作在网格上进行。

2.5 本章小结

本章节介绍本文所涉及的相关理论和技术。2.1 节主要介绍 AFL 的工作流程和技术细节。2.2 节介绍静态分析技术的特点和现有技术。2.3 节介绍 LLVM 技术和约束拆分。2.4 介绍聚类算法的分类和原理。

第 3 章　基于关键位置的模糊测试改进

3.1　整体架构

针对 1.3 节提出的现有模糊测试的问题和不足，结合第 2 章提到的相关技术-静态分析、约束拆分对现有模糊测试技术进行改进。本文提出了一种基于关键位置的模糊测试改进技术，并实现了模糊测试改进后的原型工具-AgileFuzz。

AgileFuzz 的整体工作流程如图 3.1 所示，其中绿色标记的模块为 AgileFuzz 新增或者优化的模块，浅灰色的部分是 AFL 原有的模块。AgileFuzz 在 AFL 的基础上，针对编译插桩、静态分析、种子评分、种子选择、种子变异等方面进行了优化。AgileFuzz 通过 3.2 节介绍的约束拆分的覆盖反馈策略拆分原程序复杂约束基本块，降低了程序关键位置的长度；然后通过 3.3 节介绍的种子筛选优化策略优先选中访问了拆分后新增基本块的种子；再利用 3.4 节介绍的基于聚类的关键位置确定算法快速确定种子变异过程中包括的关键位置；3.5 节针对“关键位置”的种子变异优化策略针对关键位置进行细粒度变异；3.6 节新覆盖与静态分析结合的种子优化评分策略对种子进行评分，使得更有价值的种子进行更多次变异。

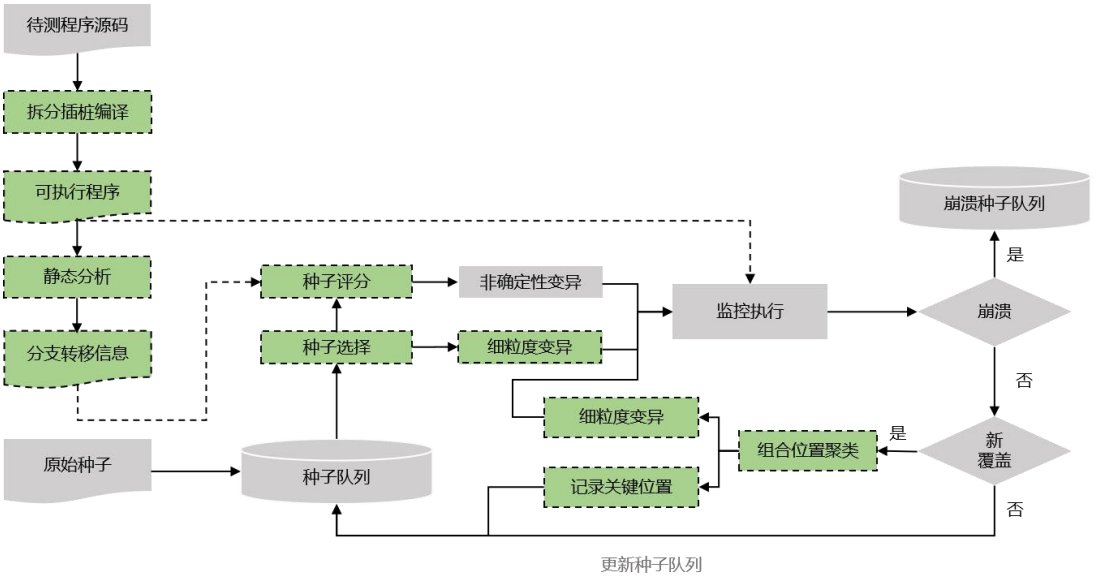

图 3.1　AgileFuzz 工作流程图

Figure 3.1　AFL workflow chart

3.2 约束拆分的覆盖反馈策略

传统的灰盒模糊测试工具难以解决长字符串匹配或长整数匹配的条件约束，其本质上的原因是，产生接近结果的种子并不会得到新的覆盖，只有变异出完全符合条件的种子才能得到新路径反馈。如果经过变异，生成满足单个字符的种子也可以得到新路径反馈，模糊测试工具将会提高变异效率，解决先前无法求解的问题。尽管有一些程序，如：xmllint，字符串匹配使用的是单字符匹配，但是较多的程序仍然使用 strcmp 函数进行匹配。所以，为了解决这种复杂求解问题，需要将长约束求解转换为短约束求解。2.2.1 节讨论的 laf-intel 在将长约束拆分的同时，带来了很多的问题：如严重的 hash 冲突、过多的 favored 种子导致的能量分配分散等。这些问题导致使用 laf-intel 后，严重影响了模糊测试的效率。

针对 laf-intel 的问题，我们提出了针对模糊测试特性的优化插桩拆分工具 lag-intel。lag-intel 拆分插桩工具包括两个算法，约束拆分算法和针对约束拆分的插桩算法。两个算法具体描述如表 3.1 和表 3.2 所示，lag-intel 约束拆分算法与 laf-intel 的约束拆分算法类似，不同的是 lag-intel 将程序原基本块和拆分新增的基本块标记（对应代码行：2,6,12,18），以便在后续插桩操作中进行区分。lag-intel 插桩算法基于 AFL 的插桩进行修改，不同之处在于 lag-intel 将程序员基本块和拆分后新增基本块进行区分（对应代码行：2,4），以执行不同的插桩逻辑（对应代码行：3,5）。总的来说，lag-intel 主要的特点如下：

1）程序原基本块与拆分后基本块使用不同的 bitmap。lag-intel 工具在拆分基本块后，会对新增的基本块进行标记。具体来说，在编译插桩时，lag-intel 使用两个不同的 bitmap 分别记录程序原分支和拆分得到的新分支的路径覆盖，使得路径统计 bitmap 不会受到新增基本块的影响。

2）程序原基本块与拆分后的基本块使用不同的路径统计方式。AFL 为了记录分支执行的次数，使用字节统计单个分支转移。但是拆分后的基本块不需要统计其执行次数，只需要反馈其是否访问。所以，拆分后的基本块使用比特位保存分支是否执行这一状态。

3）拆分后基本块的约束区分。lag-intel 针对 strcmp、长整数等进行拆分，如果是访问了 strcmp 的基本块，那么满足条件的约束值必然是有效字符类型的 ASCII 码值；而如果是长整数匹配语句，则约束值可能是 0-255 之间的任意值。为了提高变异的效率，我们将拆分基本块使用的 bitmap 分割为两个部分，strcmp

匹配拆分出的基本块只用高地址 bitmap（大小为原 bitmap 一半）。长整数匹配拆分和 switch 匹配拆分新增的基本块的路径统计在低地址 bitmap。AgileFuzz 使用 has_new_lag 字段记录每个种子是否访问新增约束基本块，值为 1 表示只访问了 strcmp 匹配约束基本块，值为 2 表示访问了数值匹配约束基本块，值为 0 表示没有访问任何新增约束基本块。

表 3.1 lag-intel 约束拆分算法

Table 3.1 lag-intel constrained splitting algorithm

算法 10 lag-intel 约束拆分算法

输入 待插桩基本块 basic_block

输出 拆分后基本块集合 basic_block_list

```
1:  procedure laf_constraint_split(basic_block)
2:    for instrtion in basic_block do
3:      if  type(instrtion) is strcmp then
4:        remove(instrtion)
5:        mark_extra_strcmp(basic_block)
6:        string=get_string(instrtion)
7:        for i in len(string) do
8:          insert_cmp_instrtion(string[i])
9:      elif  type(instrtion) is compare then
10:       remove(instrtion)
11:       mark_extra_compare(basic_block)
12:       num=get_num(instrtion)
13:       for i in len(num) do
14:         insert_compare_instrtion(num[i])
15:     elif  type(instrtion) is switch then
16:       remove(instrtion)
17:       mark_extra_switch(basic_block)
18:       do_similarity_with_strcmp_or_compare(instrsion)
19:   end for
```

表 3.2 lag-intel 针对约束拆分的插桩算法

Table 3.2 lag-intel's instrumentation algorithm for constraint splitting

算法 11 lag-intel 针对约束拆分的插桩算法

输入 待插桩基本块 basic_block

输出 插桩后的基本块

```
1:    procedure lag_instrumentation(basic_block)
2:    if get_mark(basic_block)==normal then
3:        instrumentation_normal(random(),shared_memory_normal,basic_block)
4:      elif get_mark(basic_block)==extra then
5:        instrumentation_extra(random(),shared_memory_extra,basic_block)
6：     end if
```

3.3 种子筛选优化策略

AFL 会保存路径不同的种子，而在种子筛选时则会选择包括所有分支覆盖的最小种子集合，并且满足一定的条件，主要有以下原则：

1）对于每一个分支，记录运行时间短且种子大小较小的种子，保存在 top_rated 集合。

2）从 top_rated 集合中选择包括所有分支覆盖的最小种子集合。

由于我们统计了种子的正常程序分支覆盖和拆分后的程序覆盖，两种分支覆盖探索程序分支具有不同的作用，访问拆分后基本块的种子通过针对性细粒度变异容易产生新覆盖，并且产生新覆盖效率较高，而只访问正常程序分支的种子需要通过盲目的非确定性变异，产生新覆盖效率较低。所以针对这些特性，我们设计了以下改进后的种子筛选策略：

1）如果种子 A 没有被选中过，并且标记为 has_new_lag，则直接选择种子。

2）对种子集合中的所有种子执行步骤 1），直到没有这种种子，然后统计现有种子的分支覆盖。

3）重新遍历种子集合，从标记为 has_new_cov 种子中选择包括所有分支（除了步骤 2 统计的分支）的最小种子集合。

3.4 基于聚类的关键位置确定算法

定义二：我们定义如果种子 A 只变异了某个连续的位置，产生了新覆盖，这样的连续位置叫作种子的**关键位置**。

针对非确定性变异组合位置的关键位置确定策略。模糊测试其中一个重要的变异策略是非确定性变异，这是一种针对多个随机位置的组合变异策略。如果种子经过非确定性变异后产生了新的程序覆盖，可能是因为组合位置中单个变异位置变异而导致产生新覆盖，这样的位置同样也是关键位置。AFL 为了减小种子的大小，通过不断的裁剪种子并判断路径是否发生变化确定种子的最小有效内容。我们采取类似的方式确定非确定性变异过程中存在的关键位置，策略是只保留变异组合位置中的单个位置变化结果，并判断经过单个位置变异后是否存在同样的新覆盖，如果存在新覆盖，那么这样的单个位置关键位置。

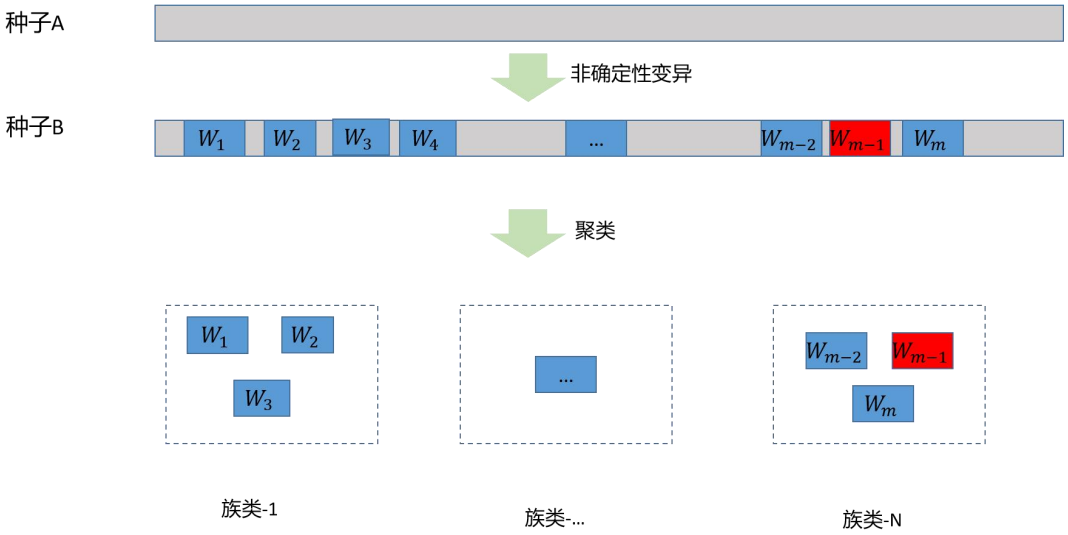

图 3.2 组合变异位置进行聚类的过程

Figure 3.2 Clustering combined variant positions

基于聚类的快速关键位置确定算法。我们的确定关键位置策略是将种子的组合位置分为几个独立的变异位置，并进行单一变异效果保留，生成新种子。需要解决的问题就是对组合位置进行分块。为了完成对种子组合变异的分块，我们使用聚类算法对离散的组合变异位置进行聚类。在文件结构中，字段是连续的一段内容，所以在一个文件中，距离越近的位置越可能属于同一字段。所以我们可以以变异位置之间的距离作为聚类的计算指标对组合位置进行聚类，距离越近的位置越可能属于同一个类。如图 3.2 所示，文件 A 的组合变异 M 个位置（其中包

括蓝色标记和红色位置标记的位置，而且红色位置为关键变异位置）得到的种子 B 访问了新的覆盖，对种子 B 变异的位置进行聚类操-将离散的变异位置聚类为 N 个族。根据不同的族的位置，生成 N 个新的种子，如图 3.3 所示，每个新种子只有对应字段的内容与原文件不同。如果种子 Ni 访问了新的覆盖，那么 Ni 种子对应的变异字段就是关键的变异字段。

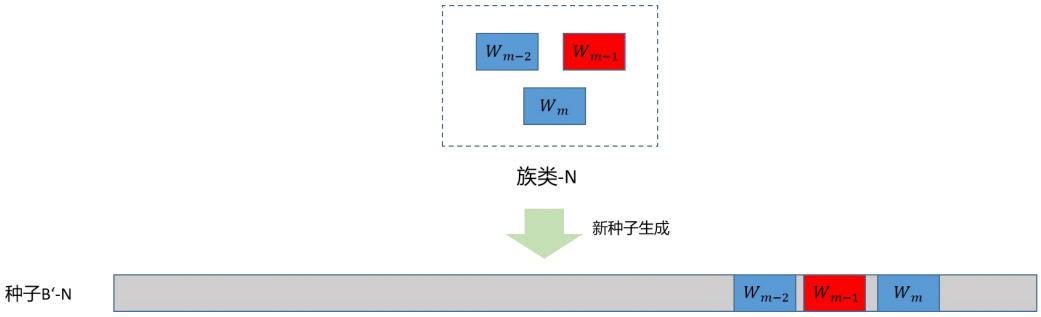

图 3.3 根据聚类结果生成新的种子

Figure 3.3 Generate new seeds based on clustering results

Meanshift 算法完成变异位置聚类。在实验中，我们选择了 Meanshift 算法。使用该算法的原因有两点：1）该算法是基于密度的聚类算法，能够将距离相近的变异位置组合成一个族，2）聚类之前不需要指定类的数量，种子的组合变异位置无法提前获知可能有多少族。Meanshift 算法的核心思想是通过不断移动样本点，使其向密度更大的区域移动，直到满足某一条件，此时该点就是样本点的收敛点，收敛到同一点则被认为属于同一族类。对于变异位置，其偏移向量计算方式如公式 1 所示，其中表示所有变异位置点中到点距离小于的所有点。

$$M_l = \frac{1}{K}\sum_{p_i \epsilon P_k}(p_i - p) \quad \text{…（公式 1）}$$

聚类算法具体步骤如下：

1）随机选择未被分类的变异位置，作为中心点-P。

2）找出离中心点 P 距离 L 范围的所有变异点，这些点记作集合 M。

3）计算中心点 P 与集合 M 中所有点的向量，将这些向量相加得到偏移向量 M。

4）将中心点 P 加上向量 M 得到新的中心点。

5）重复 2、3、4，直到偏移向量满足设定的阈值，保存此时的中心点。

6）重复 1、2、3、4、5 直到所有点都完成分类。

7）根据每个类，对每个点的访问频率，取访问频率最大的那个类，作为当

前点集的所属类。

表 3.3 聚类算法确定关键位置

Table 3.3 Clustering Algorithms to Determine Key Locations

算法 12 聚类算法确定关键位置

输入 非确定性变异后产生新覆盖的种子和变异前的种子:q_son,q

输出 保存关键位置的新种子:q_key_list

```
1：    procedure deter_keyloc_base_cluster(q,q_son)
1：      diff_points=get_diff_point(q，q_son)
2：      cluster_points=cluster(diff_ponts)
3:       q_key_list=[]
3：      for cluster_point in cluster_points:
4：        q_single_cluster=generate_seed_by_cluster(cluster_point):
5：        if(is_new_path(q_single_cluster)):
                q_single_cluster.mark_key(cluster_point)
6：             q_key_list.add(q_single_cluster)
10：     end if
11：   end for
```

通过表 3.3 所示的聚类算法确定关键位置，AgileFuzz 得到了产生新覆盖并且只变异了连续字段（关键位置）的种子，但是聚类算法的分类结果（关键位置）仍然包含了一些无关的位置，比如得到的关键位置长度为 5，但是只有一个位置变异是有效的。为了进一步缩小关键字段的长度，对于判定为关键位置聚类结果，AgileFuzz 以字节为最小替换单位，生成新的种子（新生成的种子与原种子只有一个字节内容不相同）。具体流程如下：如果种子 Ni 访问了新覆盖，并不会直接将族中的变异位置记录为关键字段，而是将族中所有的字节看作独立的变异，然后生成与原种子相比只改变了单个字节的新种子，然后以相同的方式执行程序，判断能否产生新覆盖。如果依然能够产生新覆盖，则记录下关键位置长度为 1 的新种子。

3.5 基于关键位置的细粒度变异策略

AFL 的种子变异包括种子变异策略（先执行至少一次确定性变异，再执行非确定性变异）和种子变异方式（翻转、算术运算等）。为了提高变异的效率，AgileFuzz 针对变异策略和变异方式进行了优化。

3.5.1 种子变异策略优化

AFL 原有的变异策略是对选中且未执行过确定性变异的种子执行一次确定性变异，然后计算种子评分，根据评估结果执行相应次数的非确定性变异。为了提高变异效率，AFL 还有跳过确定性变异阶段的 AFL-d 模式。

AgileFuzz 针对这一策略进行了改进，其改进策略是首先执行非确定性变异，跳过效率较低的确定性变异。AgileFuzz 在对种子执行非确定性变异时，会判断新产生的种子是否访问了新覆盖，如果发现新的程序路径而没有发现新的分支覆盖，则直接保存该种子；如果发现了新的分支覆盖，则不会直接保存种子，而是利用关键位置确定算法分析种子变异过程中的关键位置，对于存在关键位置的种子，我们会保留只变异关键位置的种子，并设置种子的 has_new_cov 字段为 1。所有操作完成后，再次执行原有经过非确定性组合变异得到的种子，如果该种子仍然能够发现新路径，则保留通过非确定性变异产生的种子。

3.5.2 种子变异方式优化

AFL 所采用的变异方式是位翻转等粗粒度变异，这种方式虽然变异效率较高，但是无法变异出字段（字节）所有可能的结果，求解复杂约束时需要消耗较长的时间。

AgileFuzz 针对这一策略进行了改进，对于存在关键位置的种子，AgileFuzz 会执行细粒度的变异策略。下面我们以具体示例介绍种子针对“关键位置”的细粒度变异策略。假设种子 A 通过非确定性变异产生新覆盖，通过分析得到了种子 A 的关键位置以及通过变异该关键位置得到的种子 B。如果关键位置的字段长度为 1，即通过变异单个字节访问了程序新覆盖，将种子 A 的关键位置（字节）变异成 0-255 所有可能的结果并执行程序；对于种子 B，将关键位置对应的偏移记录为种子 B 的关键位置，当种子 B 被选择时，变异关键字段后继两个字节，并且对于种子 B 的 has_new_lag 字段，判断其访问了哪些类型的约束基本块，如果只访问了 strcmp 拆分的基本块，那么只将种子的对应位置变异为有意义的字

符值（字母、数字、特殊字符等）。如果关键位置的字段长度大于 1，则对变异种子 A 的关键位置执行 AFL 原有的确定性变异操作，种子 B 不记录关键位置信息。

3.6 新覆盖与静态分析结合的种子评分优化策略

在 AFL 的种子评分中，会根据种子的总分支覆盖数量、种子执行时间、种子大小等进行评分，这种评分方式忽略了静态分析的程序语义信息和种子发现的新覆盖分支信息，进而导致能量分配不合理。

3.6.1 静态分析提取分支转移信息

图 3.4 为经典的二进制程序基本块分支转移示意图，branch1 分支转移的目标基本块作为新的分支转移的起始基本块，存在两条分支转移 branch1-1 和 branch1-2（真实程序中至多存在两条，至少存在零条）。考虑到汇编指令的结构特性，虽然基本块是连续指令的集合，但是 branch1-1 或 branch1-2 可能并不同时存在或都不存在。如图 3. 5 所示，这种情况发生在某基本块的子基本块的父基本块超过一个时，其子基本块即使只有一个，但是该基本块仍然被认为是一个独立的基本块。

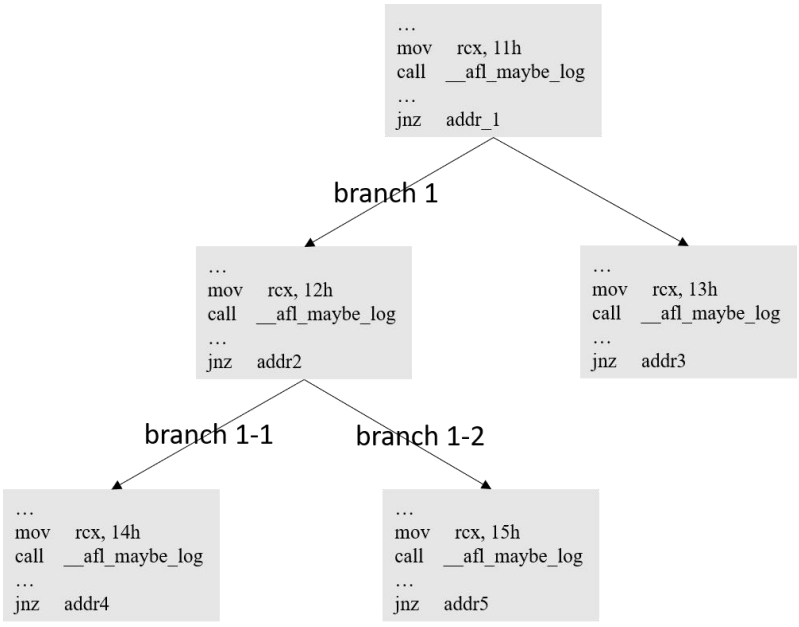

图 3.4 典型分支转移示例-1

Figure 3.4 typical branch transfer example-1

假设种子 A 在一次执行过程中，程序访问了 branch1 分支，并且只访问了 branch1-1 或者 branch1-2 其中一条分支，那么对种子 A 进行持续性变异，新生成的种子可能访问种子 A 未访问的 branch1 子分支。如果 branch1 的子分支只有一个或者零个，那么访问 branch1 的种子 A，并不能访问到其潜在未访问的分支。所以，程序的静态分支转移信息对模糊测试发现新覆盖具有重要的作用。

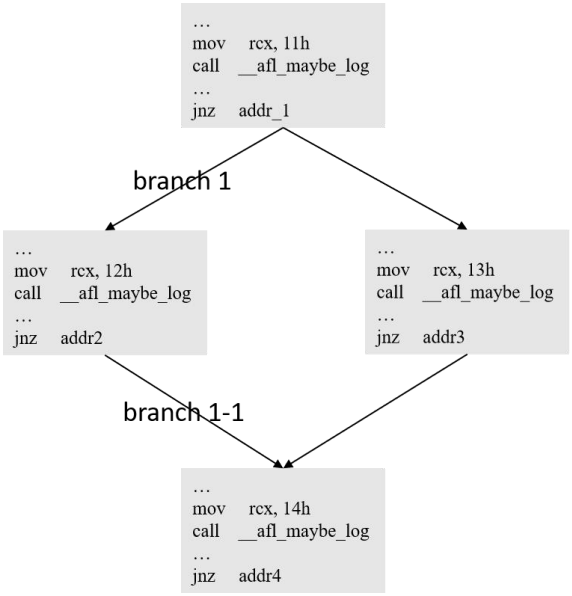

图 3.5 典型分支转移示例-2

Figure 3.5 typical branch transfer example-2

为了种子评分更加合理，AgileFuzz 会在程序插桩编译后，提取静态分支转移信息。表 3.4 所示为 AgileFuzz 静态分支提取算法，该算法的核心操作是遍历所有基本块，获取基本块插桩的随机数并计算分支转移编号，然后记录父分支转移与子分支转移的编号关系，最终得到分支转移关系序列集合<branch_id,son1_branch_id,son2_branch_id>。

表 3.4 静态分支信息提取算法

Table 3.4 Static branch information extraction algorithm

算法 13 静态分支信息提取算法
输入 待提取分支信息的函数：function
输出 提取的分支转移信息:branch_result
1: procedure stasic_analyse(function)

续表 3.4 静态分支信息提取算法

算法 13 静态分支信息提取算法

输入 待提取分支信息的函数：function

输出 提取的分支转移信息:branch_result

```
2:    branch_ids={}
3:    for basic_block in function do
4:        random_num=get_instrumentation_num(basic_block)
5:        branch_ids[basic_block.addr]=random_num
6:    end for
7:    branch_result={}
8:    for basic_block in function do
9:        son_branch1=branch_ids[basic_block.son1_addr]
10:       son_branch2=branch_ids[basic_block.son2_addr]
11:       branch_result[branch_ids[basic_block.addr]]=[son_branch1,son_branch2]
12:   end for
```

3.6.2 新增分支覆盖信息

AFL 等模糊测试工具仅根据是否访问新分支覆盖判断种子是否保留，AgileFuzz 为了更好地对种子进行评分，在保存新种子的同时，将新种子的分支信息与全局种子信息进行比较，得到新访问的分支覆盖编号，并保存。AgileFuzz 使用 65536 大小的集合保存种子访问的分支编号，但是种子新增分支数量一般较少，为了高效地记录和使用新增分支编号，AgileFuzz 使用链表进行保存。

3.6.3 种子评分优化

根据 3.6.1 节和 3.6.2 节获取的静态分支转移信息和种子新增覆盖信息，AgileFuzz 对种子评分阶段进行了优化，具体优化策略如下：

1）对于待测程序，通过静态分析获取每个基本块转移的子分支转移信息，保存在文件中，在启动模糊测试测试时，该文件作为分支附加信息提供。

2）对于每个新加入队列的种子，比较其分支覆盖与总分支覆盖的差异，得到新发现的分支覆盖数量和具体分支编号。

3）在种子评分阶段，计算种子分支覆盖中是新覆盖且对应分支有两个子分支的数量，记作 value_branch_num。

种子分数越高，对应的能量越大，被执行的次数越多。分数与能量的具体转换方式我们采用 AFL 的算法（见 2.1.4 节中算法 3）。种子 A 的评分使用公式 2 进行计算，其中 score_A_old 是使用 AFL 原有的种子分数计算方式（见 2.1.4 节算法 4、5 和 6）得到的种子初始分数，我们在 score_A_old 的基础上根据新覆盖数量进行调整得到种子的分数 score_A。种子的新覆盖数量可能差异数百倍，如果新覆盖数量与种子分数是正比关系，则新覆盖数量对种子评分产生数百倍的增益，导致过多的能量集中在少数评分较高的种子，降低变异的效率，所以我们选择 log 函数。在实验中，我们选择了多个常数为底的 log 函数进行实验比较，发现使用 ln 函数计算种子能量总体实验效果较好。根据公式可知，当种子的新覆盖数量为 0 时，种子分数为初始分数。

$$\text{score\_A} = \text{score\_A\_old} * (\ln(\text{value\_branch\_num} + 1) + 1) \qquad \text{（公式 2）}$$

3.7 本章小结

本章节主要介绍基于关键位置的模糊测试改进整体架构和技术细节。首先在 3.1 节中介绍 AgileFuzz 的整体架构，其中包括核心的几个模块：拆分插桩编译、静态分析、种子评分、种子选择和种子变异。3.2 节介绍约束拆分的覆盖反馈策略，在 AFL 插桩的基础上进行判断基本块的复杂约束并拆分。3.3 节介绍种子筛选策略优化，结合插桩后覆盖反馈对种子进行筛选。3.4 节介绍基于聚类的关键位置确定算法，通过执行后路径反馈快速确定关键位置。3.5 节针对“关键位置”的模糊测试优化策略，针对关键位置进行持续性细粒度变异。3.6 节提出了基于新覆盖和静态分析的种子评分策略。

第 4 章 模糊测试工具实现

4.1 整体方案实现

本节针对第 3 章提出的模糊测试改进方法，在 AFL v2.52b 的基础上完成了改进后的模糊测试工具 AgileFuzz。如图 3.1 所示，绿色标记且使用虚线框的模块为 AgileFuzz 改进或新增的模块，主要包括：代码编译插桩、静态分析、种子变异、种子筛选、种子评分和能量分配。

4.2 拆分插桩编译

表 4.1 lag-intel 针对拆分后新增基本块的插桩算法

Table 4.1 lag-intel's instrumentation algorithm for new basic blocks after splitting

算法 14	lag-intel 针对拆分后新增基本块的插桩算法
输入	待插桩基本块 basic_block
输出	插桩后的基本块

```
1:  procedure instrumentation_extra(random,shared_memory,basic_block)
2:    prev_loc=get_afl_prev_loc()
3:    insert_num(random)
4:    if get_mark(basic_block)==extra_strcmp then
5:      branch=prev_loc^random|8000H
6:    else
7:      branch=prev_loc^random&7FFFH
8：   end if
9:    shared_memory[branch/8]=shared_memory[branch/8]|(branch%8)
```

拆分插桩编译模块使用 LLVM 编译器框架，在 AFL 原有插桩的逻辑基础上进行修改,分为约束拆分模块和插桩模块。约束拆分模块针对字符串匹配、长整数匹配和 switch 匹配进行拆分，分别实现对应的拆分处理代码（分别为 split-compares-pass.so.cc、split-switches-pass.so.cc 以及 compare-transform-pass.cc）。插桩代码基于 AFL 的 afl-llvm-pass.so.cc 文件进行修改，对于需要插桩的基本块，AgileFuzz 分为原程序基本块和拆分后新增的基本块，两种基本块实现不同的插

桩逻辑。

针对拆分后新增的基本块的算法如表 4.1 所示，对于 strcmp 和非 strcmp 约束采用不同的插桩逻辑，即记录在共享内存的不同位置（代码行：4,5,6,7）。与 AFL 原有插桩逻辑不同，AgileFuzz 只记录是否访问该路径（代码行：8），而不记录路径访问的次数，但是用于保存拆分新增基本块的共享内存与保存正常程序路径的共享内存大小一致，因此 AgileFuzz 可以保存 8*65536 条拆分后新增的分支，极大地缓解拆分新增基本块过多带来的 hash 冲突严重的问题。

4.3 静态信息提取

静态信息提取模块需要分析并提取插桩后的二进制程序基本块转移的编号与子分支转移编号之间的对应关系。由于需要分析的插桩后程序是二进制程序，所以我们使用 ida-pro[56]工具进行分析。ida-pro 是一款功能非常强大的静态反编译软件，不仅可以对二进制程序进行反汇编操作，还提供了丰富 python 接口，如：idaapi、idautils 以及 idc 等，这些接口可以使得使用者对反编译后的二进制程序进行进一步分析，我们的静态分析也是基于此。

4.4 聚类提取关键位置

聚类的主要功能是将非确定性变异的多个离散的位置聚类为几个字段。我们所使用的聚类算法是 Meanshift 算法，该算法逻辑并不复杂，通过计算每个位置的密度并迭代偏移位置实现聚类操作。我们使用 C 语言完成该聚类算法，并与 AFL 原有代码相结合，当获取变异位置后，调用聚类算法接口完成聚类，然后对聚类后的字段进行测试，仍然能够产生新覆盖的位置被标记为关键位置。

4.5 模糊测试优化

AgileFuzz 的改进涉及模糊测试多个重要的阶段，包括编译插桩、静态分析、种子变异、种子筛选、种子评分。我们在 AFL 逻辑的基础上，对这些模块进行优化。约束拆分插桩模块在 AFL 原有的 llvm 插桩基础上，对复杂约束进行拆分，通过新增共享内存的方式避免拆分后基本块对模糊测试的不利影响。通过 Meanshift 聚类算法提取关键位置后，对模糊测试多个模块进行对应优化，并结合静态分析的分支信息，对种子评分模块进行优化。

4.6 本章小结

本章主要介绍 AgileFuzz 实现过程。4.1 节介绍整体方案实现，4.2 节介绍拆分插桩编译模块如何对基本块进行拆分，4.3 节介绍静态信息提取模块如何提取分支转移信息，4.4 节介绍聚类提取关键位置的实现，4.5 节介绍模糊测试整体优化的实现。

第 5 章　分析与评估

为了说明我们改进的模糊测试工具 AgileFuzz 具有更高效的漏洞挖掘能力，我们从针对以下几个问题进行回答，从多个方面证明 AgileFuzz 的优势。

问题一：本文提出的关键位置进行变异，是否是有效的，通过变异关键位置所产生的新覆盖占比？

问题二：本文提出的优化约束拆分插桩工具，对模糊测试的影响？与原有的拆分工具 laf-intel 相比，效率如何？

问题三：AgileFuzz 如何针对具体程序进行高效的测试，为什么效果会比其他模糊测试好？

问题四：AgileFuzz 在通用程序集测试中，与其它模糊测试工具比较，分支覆盖率是否较好？

问题五：AgileFuzz 与其它模糊测试工具相比，漏洞挖掘能力如何？

问题六：AgileFuzz 能否挖掘到真实的未知漏洞？

5.1　实验配置

为了实验的公平性，本文所有实验均在同一台服务器中进行，保证了实验的硬件环境一致，服务器配置详细信息如表 5.1 所示。

表 5.1　实验环境信息

Table 5.1　Experimental environment information

参数名	参数值
机器型号	HW2488V5
系统	CentOS Stream release 8
核心数	64
CPU 型号	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz
内存	128G

实验对比的模糊测试优化工具的选择。由于 AgileFuzz 是基于 AFL 框架优化的模糊测试工具，所以对比模糊测试工具选择了 AFL、AFL-d 和同样基于 AFL 的模糊测试优化工具 EcoFuzz、MOPT。EcoFuzz 和 MOPT 是近年来安全顶会中

较为优秀的模糊测试优化工具。

对比实验的待测程序选择。本文实验参考了近年来顶会中模糊测试改进相关的论文，从中选择了常被用作模糊测试测评的程序，具体信息在实验中详细介绍。

5.2 针对关键位置变异的有效性

AgileFuzz 对模糊测试的变异策略改进最大的体现在于，变异过程中会对种子的部分关键位置进行细粒度的变异，即在变异过程中以字节为最小变异单位，对关键位置以及后继字段的最小变异单位进行所有可能结果的遍历（0-255）。这与多数基于 AFL 的模糊测试改进存在较大区别，这些工具多以变异范围较大、粒度较粗的组合变异为主要变异方式，优化方式更侧重于种子选择、能量分配模块。

本文为了验证针对关键位置变异的有效性进行了覆盖率分析实验。为了实验结果的客观性，进行了三组重复性实验，每组实验进行 24 小时，最终统计了 objdump[48]、pdftotext[49]、xmllint[50]、harfbuzz[51]等程序在测试中数据的平均值-发现的总程序分支数量以及只通过变异单个字节产生的总分支数量。如表 5.2 所示，在测试的 4 个程序中，通过变异单个字节发现的新覆盖数量超过非确定性变异产生的新覆盖数量，这说明针对单个字节进行细粒度变异是可行的，并且在发现新覆盖方面起到了重要的作用。

表 5.2 单字节变异分支覆盖统计

Table 5.2 Single-byte mutation branch coverage statistics

程序	总分支数量	单字节变异产生的分支数量	单字节有效变异覆盖比例
objdump	6815	4011	0.58
pdftotext	9496	6238	0.65
xmllint	8794	6933	0.78
harfbuzz	6356	3702	0.58

5.3 约束拆分插桩工具 lag-intel 的有效性

laf-intel 是一个基于 LLVM 的约束拆分编译工具，针对有源代码的程序在对其进行编译期间，将 strcmp、长整数匹配等复杂约束判断拆分为单个字符和数值

匹配。因为 laf-inel 将程序拆分后与 AFL 结合使用时带来的 hash 冲突，造成模糊测试总体效率没有得到明显提升，我们提出改进后的约束拆分插桩工具 lag-intel。

表 5.3 拆分工具对模糊测试的影响

Table 5.3 The impact of Split tool on fuzzing

程序	模糊测试工具	插桩工具	拆分覆盖率	真实覆盖率	差异
nm	AgileFuzz	lag-intel	13.70%	13.62%	0
	AgileFuzz	laf-intel	48.39%	12.63%	3.83
	AFL	laf-intel	28.77%	6.90%	4.16
	AFL	LLVM	-	8.06%	-
	AFL-d	laf-intel	39.49%	10.44%	3.78
	AFL-d	LLVM	-	10.43%	-
	EcoFuzz	laf-intel	42.87%	11.62%	3.68
	EcoFuzz	LLVM	-	11.31%	-
pdftotext	AgileFuzz	lag-intel	13.29%	13.24%	0
	AgileFuzz	laf-intel	44.56%	13.11%	3.39
	AFL	laf-intel	31.02%	9.02%	3.68
	AFL	LLVM	-	8.35%	-
	AFL-d	laf-intel	33.87%	10.03%	3.37
	AFL-d	LLVM	-	10.35%	-
	EcoFuzz	laf-intel	29.42%	8.49%	3.46
	EcoFuzz	LLVM	-	9.82%	-
tcpdump	AgileFuzz	lag-intel	20.83%	20.74%	0
	AgileFuzz	laf-intel	87.37%	17.62%	4.96
	AFL	laf-intel	76.23%	13.40%	5.68
	AFL	LLVM	-	12.16%	-
	AFL-d	laf-intel	86.81%	17.44%	4.97
	AFL-d	LLVM	-	17.25%	-
	EcoFuzz	laf-intel	87.64%	17.76%	4.9
	EcoFuzz	LLVM	-	18.11%	-

为了验证我们改进后的 lag-intel 的效果，我们将使用了 lag-intel 的 AgileFuzz、使用了 laf-intel 的 AgileFuzz、使用了 laf-intel 拆分插桩和 AFL 原始的 LLVM 插桩的 AFL、AFL-d 和 EcoFuzz 进行对比，并分别统计使用拆分插桩工具后统计的覆盖率以及对应的种子在未进行拆分的程序中的覆盖率。实验统计数据如表 5.3 所示，可以得到以下结论：

1）使用 laf-intel 拆分工具和未使用拆分工具的程序，进行模糊测试时统计的真实的分支覆盖数量（程序原分支覆盖）和加上拆分后基本块的统计数量出现了 3 倍以上的差异，导致了巨大的 hash 冲突。尤其是 tcpdump[52]程序，bitmap 覆盖已经达到了 80%以上，严重影响模糊测试的正常路径反馈工作，而 lag-intel 拆分工具并不会带来 hash 冲突严重的问题。

2）使用 laf-intel 的其它模糊测试工具在 24 小时的实验中与使用 AFL 原始的 llvm 插桩程序的覆盖统计几乎没有差别，在一些程序中效果甚至有所下降。结果说明直接使用 laf-intel 工具并不能有效提高这些模糊测试工具的效率。

3）使用 lag-intel 的 AgileFuzz 的程序覆盖率高于使用 laf-intel 的 AgileFuzz 的程序覆盖率，说明 lag-intel 能够降低拆分约束带来的对模糊测试的影响。

实验结果说明，laf-intel 虽然能够将复杂约束求解成功转化为单字符或数值约束匹配，但是使用该工具后总体实验效果并不好。结合 AFL 以及约束拆分特点分析可知，主要有以下几点原因：1）laf-intel 将原程序简单的语句转化为多条单个匹配，造成了更多的性能开销，2）拆分后的程序基本块数量较原先增加数倍，使得作为路径统计的 bitmap 的 hash 冲突十分严重，影响新的路径统计，3）基本块的增加会导致模糊测试工具标记为 favored 的种子增加，在一定时间内程序原有分支所得到的能量就会减少。

5.4 AgileFuzz 在具体程序中的路径覆盖高效性

在这一部分，我们结合具体程序说明 AgileFuzz 为何能更快地发现新覆盖。结合其它模糊测试工具常用的程序测试集，我们选择了 binutils 和 libxml2 程序进行实验，并针对实验现象进行分析，详细介绍 AgileFuzz 针对程序特性进行的测试优化以及最终得到的较好的实验结果。

5.4.1 xmllint 程序长约束求解高效性

我们首先以 libxml2 软件的 xmllint 程序为例，将我们的模糊测试工具 AgileFuzz 与 AFL 2.52b、关闭确定性变异的 AFL 2.52b -d 以及关闭确定性变异的模糊测试改进工具 EcoFuzz 进行 24 小时实验比较，覆盖率对比结果如图 5.1 所示，图中可以看出 AgileFuzz 在相同的时间内发现了更多的路径。我们将对这一实验结果进行详细的分析，以展示 AgileFuzz 的优势。

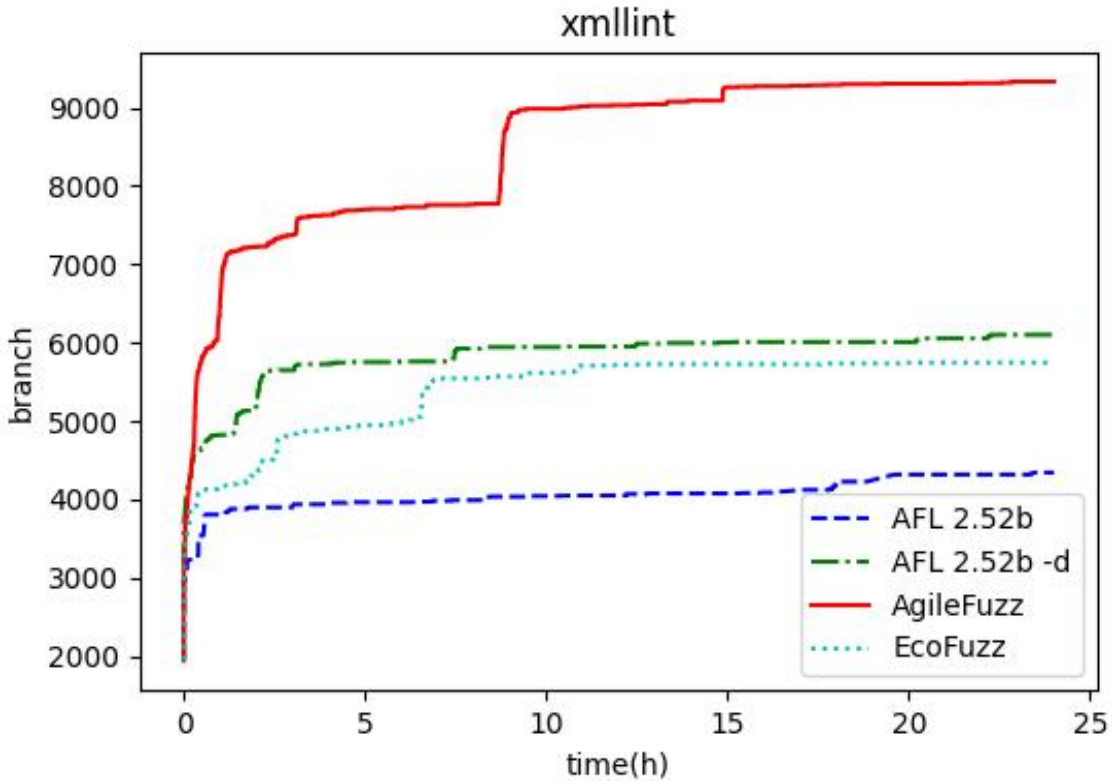

图 5.1 xmllint 程序覆盖率对比

Figure 5.1 xmllint program coverage comparison

为了分析不同模糊测试产生覆盖率差异的原因，我们使用 afl-cov[53]分析四个模糊测试工具所发现的程序覆盖的差异。分析结果显示，AgileFuzz 不仅发现了 EcoFuzz 等工具所发现的全部程序覆盖，还发现了更多的难以发现的程序代码。其中 hash.c 文件的代码，AgileFuzz 覆盖率为 56.3%，而 EcoFuzz、AFL 等工具的覆盖率为 0%，这是导致结果中程序覆盖率出现较大差异的主要原因。

我们进一步分析产生现象的原因，通过程序分析发现如图 5.2 所示的调用序列以及条件判断。这里简单介绍一下 CMP9 函数，这是 xmllint 实现的定长字符串匹配宏定义，与 c 语言自带的 strcmp 函数不同的是，"<"和"<！"虽然都没有完

全满足匹配条件，但是调用 cmp9 触发的程序覆盖是不同的，而调用 strcmp 触发的程序覆盖是相同的。

EcoFuzz 等工具通过 24 小时的变异无法产生满足条件约束的字符串-"<!ATTLIST"，这是非确定性变异的随机性和组合性所造成的。当 EcoFuzz 得到包含"<"的种子后，保存该种子到种子队列中。当从种子队列选择该种子后，非确定性变异会随机选择变异位置，选择"<"的后继位置的可能性很低，并且组合变异可能会导致"<"或者其它已经满足约束条件的字段被修改为错误的内容，从而导致无法产生有效变异。未关闭确定性变异的 AFL 2.52b，虽然会对种子所有位置进行变异，但是由于其变异效率较低并且变异粒度不够细，同样没有产生满足约束条件的种子。

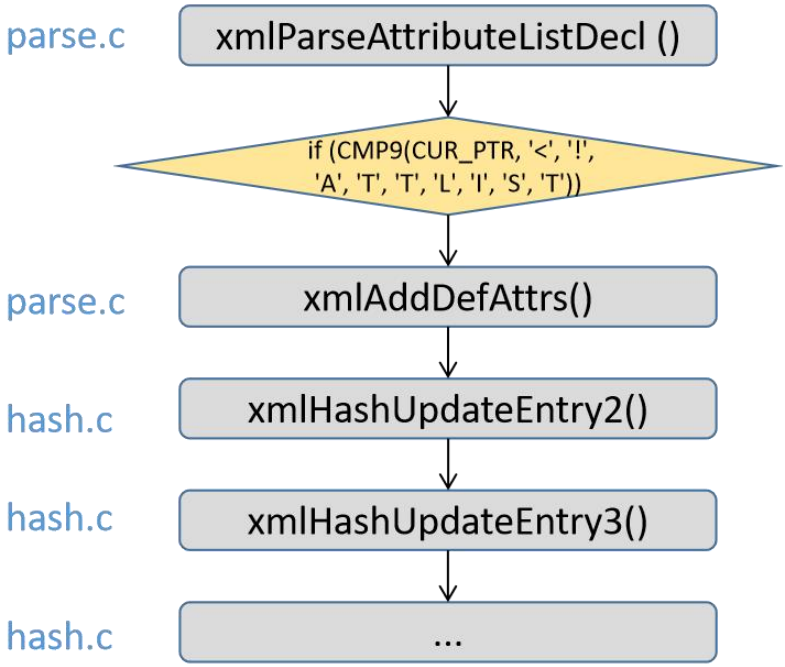

图 5.2 xmllint 程序 hash.c 调用依赖

Figure 5.2 xmllint program coverage comparison

AgileFuzz 在通过非确定性变异产生了包含字符串"<**"种子时，触发了 cmp9 函数新的覆盖，此时 AgileFuzz 利用聚类算法确定单一有效变异位置，确定新覆盖是由于变异了"<"字符所在的位置.保存的新种子记录了关键位置，能够对关键位置的后继位置进行持续性的细粒度变异，从而保证了产生"<!"字符串，迭代下去则保证能够产生"<!ATTLIST"字符串，从而满足条件约束。图 5.3 为 AgileFuzz 通过确定性变异产生满足 cmp9 函数条件约束的种子的产生序列，红色标记的字

段是种子记录的关键位置，对于 id 为 001889 种子，AgileFuzz 会对"<"的后继字符进行细粒度确定性变异，从而能够迅速产生"<!**"字符串。

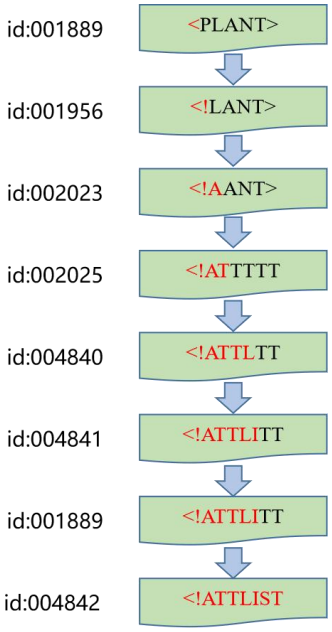

图 5.3 种子变异过程

Figure 5.3 Seed mutation process

5.4.2 binutils 程序约束值多解高效性

AgileFuzz 不仅擅长求解长约束求解类型的问题，针对单字段多值求解问题同样具有高效性。如图 5.4 所示，为典型的 switch 类型分支语句，其程序特点是，当关键字段等于不同的内容时，会满足不同的分支条件，进而访问不同的程序代码。

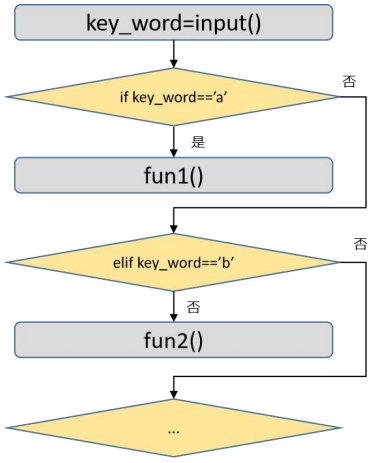

图 5.4 单字段多值求解示例

Figure 5.4 Single Field Multivalue Solver Example

为了进一步介绍 AgileFuzz 的高效性，我们设置了另一组对比实验。实验选取 AgileFuzz 与 AFL、AFL-d、EcoFuzz 模糊测试工具进行对比，实验对象为 binutils 软件的 nm 程序。如图 5.5 所示，在相同时间和环境条件下，AgileFuzz 程序覆盖率相较于其他工具分别高出 67%、29%和 19%。

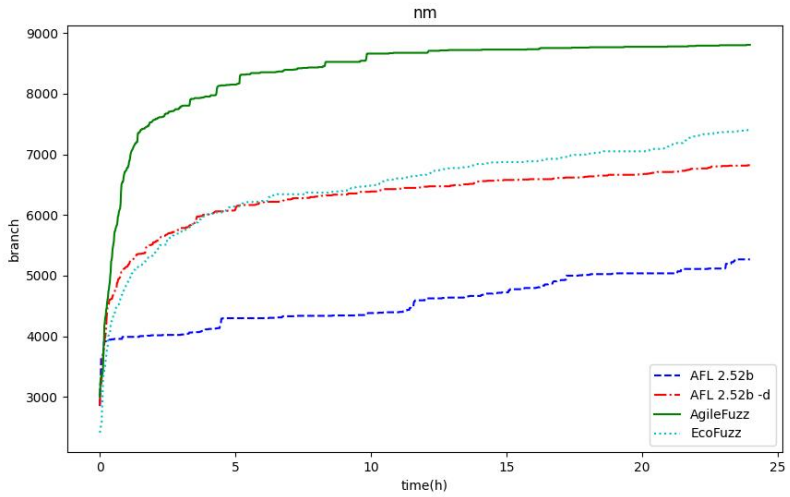

图 5.5 nm 程序覆盖率对比

Figure 5.5 nm program coverage comparison

通过分析 binutils 程序的源码，并利用 afl-cov 工具对实验过程中访问的程序代码进行分析，我们发现程序覆盖率差异是因为：binutils 源码中的 cp-demangle.c 文件中存在很多类似“d_check_char (di, '_')”、“if (peek2 == 'T' || peek2 == 't')”、“if (! d_check_char (di, 'Z'))”、“d_peek_next_char (di) == '_'”这样的语句。这种语句有以下几个特点：

1）约束类型为单字符约束求解，并且字符内容直接来自于程序输入文件。

2）输入文件的某些关键位置内容被多个不同约束条件进行判断，其内容不同将会触发不同的分支。

3）当前字符满足约束条件后，下一个字符也作为约束求解的值，如：“d_peek_next_char (di)”语句则是获取当前字符的下一个字符进行匹配。

综上所示，AgileFuzz 针对这类程序特性能够较快地变异出所有可能的字段，因为 AgileFuzz 能够对产生新覆盖的位置的相同位置或相近位置进行持续性的细粒度变异，保证了变异位置精确以及变异方式相较于其他模糊测试工具，极大地降低了变异的盲目性和随机性。

5.5 AgileFuzz 探索多个程序路径的高效性

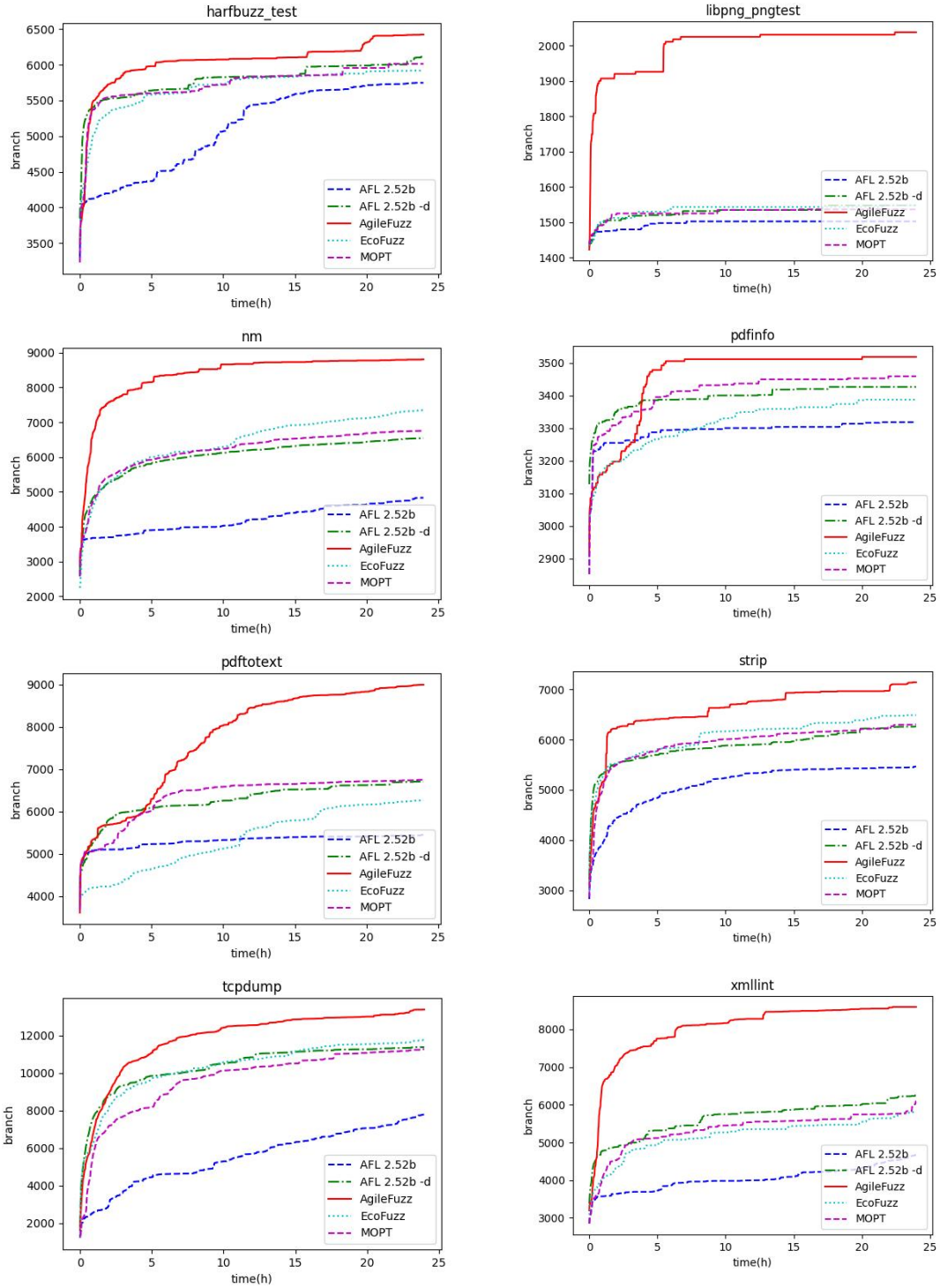

图 5.6 程序覆盖率对比图

Figure 5.6 Program coverage comparison chart

通过上节分析可知，由于 hash 冲突增加以及保留为 favored 的种子多导致能量分配过于分散等原因，其它模糊测试改进结合 laf-intel 并没有带来覆盖率的有

效提升，所以我们对使用了 laf-intel 拆分插桩工具的 AgileFuzz 和使用 AFL 原始的 llvm 插桩工具的 AFL、AFL-d、MOPT、EcoFuzz 进行对比实验。实验选择了常被用作模糊测试测评的 binutils、harfbuzz、libxml2 等程序，在相同环境下进行了 3 组重复性实验，每组实验进行 24 小时，最终取三组实验的平均覆盖率作为对比结果。覆盖率对比图如图 5.6 所示，从结果中可以看出 AgileFuzz 在所有程序中都能发现最多的程序分支覆盖。

5.6 AgileFuzz 发现程序漏洞的高效性

模糊测试工具的评测指标最重要的一项是漏洞挖掘的能力高低。漏洞挖掘数量对比实验相较于覆盖率评测，具有极大的偶然性。为了更准确、客观的评价 AgileFuzz 的漏洞挖掘能力，我们设置了较为客观的漏洞挖掘对比实验。

5.6.1 漏洞挖掘对比实验

为了验证 AgileFuzz 发现漏洞的高效性，我们设置了漏洞挖掘对比实验。实验测试对象选择了常用作模糊测试测评的软件，对应软件选择了早期版本以便于对比漏洞数量，软件具体信息如表 5.4 所示。

表 5.4 漏洞对比实验所选取的软件信息

Table 5.4 Software information selected for vulnerability comparison experiment

软件	版本号	发布时间
binutils	2.26.1	2016
tcpdump	3.9.4	2005
xpdf	3.04	2014
libxml2	2.6.32	2008
libpng	1.0.9	2009
harfbuzz	2.0.0	2018

为了实验的客观性和非偶然性，我们在相同的环境内进行了三组重复性实验，每组实验进行 24 小时，最终统计三次实验中各个模糊测试工具发现漏洞数量的平均值，具体数据信息如表 5.5 所示。

从图中数据可以看出，AgileFuzz 在相同时间内发现了更多的程序漏洞，并且每个程序的测试结果都是 AgileFuzz 发现的漏洞数量最多。这是因为 AgileFuzz

能够针对所有程序的长约束语句进行拆分，相较于其他模糊测试的改进，AgileFuzz 提升程序覆盖率的能力更强，更容易访问程序存在漏洞的代码。实验结果还有一个有趣的现象：即只有 AgileFuzz 在 readelf 和 xmllint 程序中发现了程序漏洞（图中用红色标记）。

表 5.5 漏洞挖掘对比实验结果

Table 5.5 Vulnerability mining comparison experiment results

软件	模糊测试工具				
	AgileFuzz	EcoFuzz	MOPT	AFL	AFL -d
nm	**32**	11	16	1	18
readelf	**3**	0	0	0	0
xmllint	**1**	0	0	0	0
tcpdump	**48**	28	27	17	30
pdfinfo	**3**	2	5	2	2
pdftotext	**55**	50	30	4	30
pngtest	**0**	0	0	0	0
harfbuzz_test	**0**	0	0	0	0
汇总	**143**	93	80	25	80

5.6.2 xmllint 程序漏洞挖掘结果分析

AgileFuzz 在漏洞挖掘对比实验中取得了较好的结果。我们针对 xmllint 程序漏洞挖掘的结果进行分析，即为何只有 AgileFuzz 发现该漏洞。通过手动分析该漏洞，得到如下图 5.7 所示的程序异常调用栈，绿色标记的函数为关键函数。通过对比其他模糊测试工具，判断出 AgileFuzz 能够发现该漏洞的关键原因是访问了 xmlParseInternalSubset 函数，而其它模糊测试工具并没有访问该函数。

我们针对程序崩溃的关键调用序列，并结合 libxml2 的源码，进一步分析 libxml2 的 parse.c 文件，得到图 5.8 所示的约束，显然只有变异出符合条件的字段-!DOCTYPE，才能访问 xmlParseInternalSubset 函数。AgileFuzz 能够针对这种复杂约束进行快速的求解，而 EcoFuzz 等模糊测试工具由于变异的盲目性，难以变异出符合条件的种子，故而无法发现对应的漏洞。

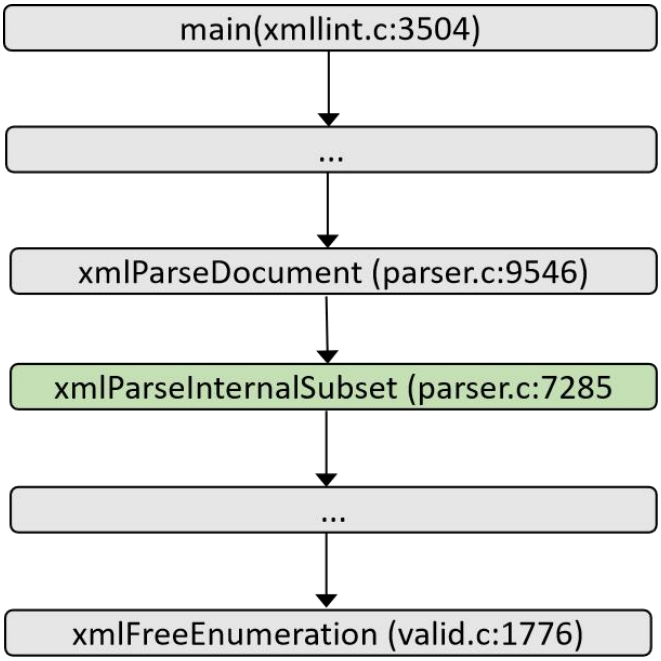

图 5.7 漏洞崩溃调用栈

Figure 5.7 Vulnerability crash call stack

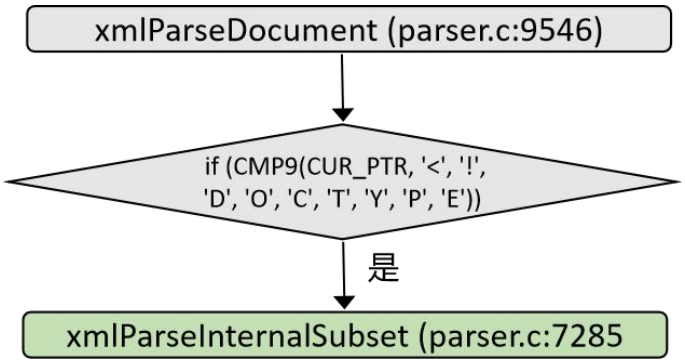

图 5.8 漏洞崩溃关键约束

Figure 5.8 Vulnerability crash critical constraints

5.7 真实世界程序中的漏洞

为了验证 AgileFuzz 对软件未知漏洞挖掘的能力，我们对 binutils、harfbuzz、fontforge[54]、ffjpeg[55]等软件等程序的最新版本进行了模糊测试。在漏洞挖掘中，我们发现了这些软件大量的未知程序崩溃，并将漏洞分析结果提交给作者。其中三个漏洞获得了 CNNVD 编号，分别为：CNNVD-202201-384、CNNVD-202201-388 和 CNNVD-202111-1731。通过对程序漏洞进行分析并对比程序漏洞提交记录，整理了如表 5.6 漏洞统计表。该研究成果表明我们的系统

AgileFuzz 具备在实践中发现漏洞的能力。

表 5.6 真实世界程序中发现的漏洞

Table 5.6 Vulnerabilities found in real-world programs

软件	版本	漏洞类型	状态
fontforge	2021.5.9	内存泄漏	等待 CVE 审核
harfbuzz	2.8.0	内存泄漏	等待 CVE 审核
objdump	2.37	SEGV	已存在该 Bug 报告
cxxfilt	2.37	栈溢出	已存在该 Bug 报告
htmlcss	2021.5.9	内存泄漏	等待 CVE 审核
packJPG	2021.4.6	堆溢出	**CNNVD-202201-384**
ffjpeg	2021.4.4	使用未初始化的值	**CNNVD-202201-388**
Xpdf	4.03	栈溢出	**CNNVD-202111-1731**

5.8 实验结果的有效性分析

本小节主要讨论本章对 AgileFuzz 评估的有效性，主要从以下几个方面进行分析：1）实验结果是否具有偶然性，2）实验对象、环境、测试用例等是否刻意选择。

5.8.1 实验结果的非偶然性

本章所有对比实验均进行了三次相同的重复性实验，并且选择了各个种类的多个程序进行覆盖率对比、漏洞挖掘数量对比的实验，实验结果客观并不存在偶然性。并且 5.4 节针对 AgileFuzz 覆盖率较高的现象进行了详细的解释，AgileFuzz 针对程序的特性，优化模糊测试流程，进而取得了较好的实验结果。

5.8.2 实验配置的客观性

实验对象的选择来自于其他模糊测试论文，并尽可能选择输入格式为多种类型的程序。在进行实验时，保证所有实验都在同一实验环境进行，并且保证服务器运行状态相近。实验的程序原始语料库来自于 AFL 提供的语料库和程序自带的测试用例，保证测试结果的客观性。

5.9 本章小结

本章主要是从多个方面对 AgileFuzz 进行评估。5.1 节介绍实验的环境配置。5.2 节证明了针对关键位置进行持续变异是有效的。5.3 节针对本文提出的约束拆分插桩工具进行评估，与原有的工具对比，证明了 lag-intel 的高效性。5.4 节通过实际程序分析，详细解释了 AgileFuzz 路径覆盖的高效性。5.5 节从覆盖率角度，与现有模糊测试改进工具进行对比。5.6 节从漏洞挖掘数量角度，与现有模糊测试改进工具进行对比。5.7 节介绍了 AgileFuzz 所发现的漏洞，5.8 节分析实验的有效性。通过上述实验分析，证明了 AgileFuzz 的先进性-能够更快地发现程序漏洞。

第 6 章 总结与展望

6.1 总结

软件安全关于国计民生，涉及国家安全、个人工作、生活、娱乐等方方面面。近年来，软件安全漏洞频出，对国家、企业、个人造成了巨大的危害。我们在享受软件带来的便利性的同时，也遭受软件带来的挑战。为了解决软件漏洞的危害影响，相关的安全研究人员以提高软件安全性为目的，提出了各种有效的程序安全分析方法，其中模糊测试技术由于其高效率、易拓展、易部署的优点，受到了安全人员的广泛关注，并且模糊测试在实际工作中发现了程序较多的安全问题，具有较高的实用性。但是现有模糊测试存在盲目性、随机性的问题，导致其消耗了大量的时间在无效的操作上，因此本文致力于研究如何提高模糊测试的效率：

1. 本文首先介绍了现有的软件安全研究方法及其原理，并分析各安全研究方法的优点和不足，阐述了模糊测试在软件安全研究方面的优势。然后针对模糊测试的分类和技术细节进行了详细的阐述，包括：插桩变异、路径反馈、种子变异、种子筛选等。在介绍模糊测试相关技术的同时也介绍了现有模糊测试存在的盲目性和随机性的问题。最终本文针对模糊测试的问题提出了针对关键位置优化的模糊测试改进技术。

2. 现有的模糊测试技术难以求解复杂条件约束，同时确定性变异和非确定性变异存在各种的问题，导致变异的效率较低，并且在种子评分阶段，忽略了种子分支转移信息和新覆盖对种子变异后发现新分支能力的影响。针对这些问题，本文提出的优化策略通过约束拆分插桩解决模糊测试难以求解长约束匹配的难题，并对种子的关键位置进行细粒度变异以提高变异的效率，然后利用新覆盖和静态分支转移信息对模糊测试的种子进行了优化。结合这些改进，本文实现了模糊测试改进原型工具-AgileFuzz。

3. 本文通过大量的实验验证了 AgileFuzz 的高效性，并结合具体程序解释了为何 AgileFuzz 发现程序分支覆盖和漏洞数量较高。AgileFuzz 在实验过程中发现 8 个程序未修复的漏洞，包括 6 个未知漏洞，其中三个未知漏洞获得了 CNNVD 编号。

6.2 展望

本文提出的基于关键位置的模糊测试改进技术在实验过程中，取得了不错的实验结果，结合具体程序分析并做了多组验证实验，证明了本文提出方法的有效性，同时本文所提方法仍然存在一些改进的空间，未来我们将会针对存在的问题进一步研究，主要的问题包括：

1. 本文提出的约束拆分编译插桩策略虽然针对有源码的程序能够有效的进行拆分，但是在一些场景下无法有效使用。比如：如果待测程序非开源，那么很多源代码的细节难以获取，导致无法有效识别 strcmp 等约束语句。

2. 本文提出的针对静态分支转移信息对种子进行评分的策略，能够有效提高模糊测试的效率，但是部分程序可能存在较多的隐式函数调用，这些隐式调用难以提取分支转移信息，如果能够结合指向分析技术，就能够进一步优化种子评分策略。

3. 本文提出的针对关键位置变异的细粒度变异策略，能够有效提高变异效率，但是对于一些无法将关键位置缩减为长度为 1 的程序，细粒度变异无法起到效果。我们考虑针对程序长约束进行进一步研究，完善对程序所有长约束的拆分。

参考文献

[1] 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

[2] 陈梁. “震网” 病毒敲响自动化系统安全警钟[J]. 工业控制计算机, 2010, 10.

[3] 柒月. 2015 年度国际网络安全重大事故攻击, 漏洞无处不在, 数据安全危在旦夕[J]. 信息安全与通信保密, 2016 (2): 38-39.

[4] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6519 [DB/OL], 2020 .

[5] 任玉柱,张有为,艾成炜.污点分析技术研究综述[J].计算机应用,2019,39(08):2302-2309.

[6] Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework[C]//Proceedings of the 2007 international symposium on Software testing and analysis. 2007: 196-206.

[7] Baldoni R, Coppa E, D’elia D C, et al. A survey of symbolic execution techniques[J]. ACM Computing Surveys (CSUR), 2018, 51(3): 1-39.

[8] Zhang T, Jiang Y, Guo R, et al. A Survey of Hybrid Fuzzing based on Symbolic Execution[C]//Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies. 2020: 192-196.

[9] 向灵孜. 源代码审计综述[J]. 保密科学技术, 2015 (12): 36-41.

[10] 任泽众,郑晗,张嘉元,王文杰,冯涛,王鹤,张玉清.模糊测试技术综述[J].计算机研究与发展,2021,58(05):944-963.

[11] Technical “ whitepaper ” for afl-fuzz[Z]. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[12] 张婉莹. 白盒模糊测试技术的研究与改进[D]. 南京邮电大学, 2019.

[13] Chen P, Chen H. Angora: Efficient fuzzing by principled search[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 711-725.

[14] 李红辉, 齐佳, 刘峰, 等. 模糊测试技术研究[J]. 中国科学: 信息科学, 2014, 44(10): 1305-1322.

[15] Eddington M. Peach fuzzing platform[J]. Peach Fuzzer, 2011, 34.

[16] Lee S, Yoon C, Lee C, et al. DELTA: A Security Assessment Framework for Software-Defined Networks[C]// The Network and Distributed System Security Symposium (NDSS). 2017.

[17] Klees G, Ruef A, Cooper B, et al. Evaluating fuzz testing[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 2123-2138.

[18] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. in IEEE Transactions on Software Engineering, vol. , no. 01, pp. 1-1, 5555

[19] C. Lv, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, “MOPT: Optimize Mutation Scheduling for Fuzzers,” in USENIX Security ’19, 2019, pp. 1949–1966.

[20] Yue T, Wang P, Tang Y, et al. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit[C]//29th {USENIX} Security Symposium ({USENIX} Security 20). 2020: 2307-2324.

[21] Lemieux C, Sen K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage[C]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018: 475-485.

[22] Chen P, Chen H. Angora: Efficient fuzzing by principled search[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 711-725.

[23] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C]// The Network and Distributed System Security Symposium (NDSS). 2016, 16(2016): 1-16.

[24] Wang J, Chen B, Wei L, et al. Skyfire: Data-driven seed generation for fuzzing[C]//2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017: 579-594.

[25] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2329-2344

[26] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing [C]// The Network and Distributed System Security Symposium (NDSS). 2017, 17: 1-14.

[27] Zong P, Lv T, Wang D, et al. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning[C]//29th {USENIX} Security Symposium ({USENIX} Security 20). 2020: 2255-2269.

[28] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain[J]. IEEE Transactions on Software Engineering, 2017, 45(5): 489-506.

[29] Chen H, Xue Y, Li Y, et al. Hawkeye: Towards a desired directed grey-box fuzzer[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 2095-2108.

[30] https://gitlab.com/laf-intel/laf-llvm-pass,[DB/OL], 2021.

[31] 侯刚, 周宽久, 勇嘉伟, 等. 模型检测中状态爆炸问题研究综述[D]. , 2013.

[32] Groce A, Visser W. Heuristics for model checking Java programs[J]. International Journal on Software Tools for Technology Transfer, 2004, 6(4): 260-276.

[33] Dong Y, Ramakrishnan C R. An optimizing compiler for efficient model checking[M]//Formal Methods for Protocol Engineering and Distributed Systems. Springer, Boston, MA, 1999: 241-256.

[34] Kurshan R, Levin V, Yenigün H. Compressing transitions for model checking[C]//International conference on computer aided verification. Springer, Berlin, Heidelberg, 2002: 569-582.

[35] Özdemir K, Ural H. Protocol validation by simultaneous reachability analysis[J]. Computer Communications, 1997, 20(9): 772-788.

[36] Grégoire J C. State space compression in SPIN with GETSs[C]//Proc. Second SPIN Workshop, Rutgers Univ. 1996.

[37] Visser W, Barringer H. Memory efficient state storage in SPIN[C]//Proceedings of the 2nd SPIN Workshop. Providence, RI: American Mathematical Society, 1996, 21.

[38] Larsen K G, Larsson F, Pettersson P, et al. Efficient verification of real-time systems: Compact data structure and state-space reduction[C]//Proceedings Real-Time Systems Symposium. IEEE, 1997: 14-24.

[39] Parreaux B. Difference compression in spin[C]//SPIN. 1998, 56.

[40] Barnat J, Brim L, Stříbrná J. Distributed LTL model-checking in SPIN[C]//International SPIN Workshop on Model Checking of Software. Springer, Berlin, Heidelberg, 2001: 200-216.

[41] Barnat J, Brim L, Cerná I. Property driven distribution of nested DFS[C]//University of Southampton, UK. 2002.

[42] Barnat J, Brim L, Chaloupka J. Parallel breadth-first search LTL model-checking[C]//18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings. IEEE, 2003: 106-115..

[43] Cadar C, Dunbar D, Engler D R. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs[C]//OSDI. 2008, 8: 209-224.

[44] Moura L, Bjørner N. Z3: An efficient SMT solver[C]//International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008: 337-340.

[45] LATTNER C, ADVE V. Llvm: A compilation framework for lifelong program analysis & transformation[C]//International Symposium on Code Generation and Optimization, 2004.CGO 2004. IEEE, 2004: 75-86

[46] Ester M, Kriegel H P, Sander J, et al. A density-based algorithm for discovering clusters in large spatial databases with noise[C]//kdd. 1996, 96(34): 226-231.

[47] Derpanis K G. Mean shift clustering[J]. Lecture Notes, 2005: 32.

[48] binutils, 2021, [online] Available: https://sourceware.org/binutils.

[49] Xpdf, 2021, [online] Available: http://www.xpdfreader.com.

[50] libxml2, 2008, [online] Available: http://xmlsoft.org.

[51] harfbuzz, 2021, [online] Available: https://github.com/harfbuzz/harfbuzz.

[52] tcpdump, 2005, [online] Available: https://www.tcpdump.org.

[53] afl-cov, 2018, [online] Available: https://github.com/mrash/afl-cov.

[54] fontforge 2021, [online] Available: https://github.com/fontforge/fontforge.

[55] ffjpeg, 2021, [online] Available: https://github.com/rockcarry/ffjpeg.

[56] EAGLE C. The ida pro book[M]. no starch press, 2011.

作者简历及攻读学位期间发表的学术论文与研究成果

作者简历

王化磊，安徽省蚌埠市人，中国科学院软件研究所硕士研究生。研究生专业是软件工程（学术），主要研究方向是软件与系统安全，在校期间发表论文一篇，参与多项国家科研项目。

教育经历

1. 2015 年 9 月-2019 年 6 月，工学学士，中国民航大学，计算机学院，计算机科学与技术专业。

2. 2019 年 9 月-2022 年 6 月，工学硕士，中国科学院软件研究所，软件工程专业。

已发表（或正式接受）的学术论文

1. 程亮，王化磊，张阳, 等. 基于聚类和新覆盖信息的模糊测试改进 [J]. 计算机系统应用, 2022.

参加的研究项目及获奖情况

1. 国家重点研发计划，软件与系统漏洞分析与可利用判定技术研究；
2. 国家自然科学基金，统一智能内核模糊测试技术研究；
3. 国家重点研发计划，xxxx 技术研究；
4. 北京网络数据研究所（横向），保密 xxxx 采购项目；

致　谢

不知不觉间，研究生就要毕业了，回首整个研究生阶段，自己成长了许多。三年的时间，每天都过得很充实，相较于本科阶段，研究生期间科研项目更加丰富且具有挑战。研究生期间遇到了很多挫折，在很长一段时间里，对自己科研和能力充满了怀疑，担心未来，不知道该做些什么。幸运的是，在老师，家人，朋友的关心和鼓励下，我走过了最困难的时期，未来更加广阔的世界在等着我。我能够顺利完成研究生的学业，离不开帮助我的每一个人。

首先特别感谢研究生期间的三位老师：张阳老师，程亮老师以及孙晓山老师。作为本科是计算机科学与技术的学生，我很感谢三位老师将我带领到软件与系统安全领域。刚入学时，专业技能非常差，很多相关方向的知识都没有掌握，是孙老师孜孜不倦地教导我，让我参与到组内的多个项目中，自此我开始了研究生的科研工作，并在这个过程中学习到了很多科研知识和做事的方法。同时，非常感谢程老师和张阳老师，他们丰富的学识修养、耐心的教导和关心，让我度过了一个又一个研究生最困难的时期。在老师们的帮助下，我才得以完成研究生毕业论文。我十分感谢国科大和软件研究所的所有老师对我的耐心教导以及对我的巨大帮助，在此表示感谢。

感谢王文硕师兄，傅钰师兄，感谢我的师弟范俊杰和周庚，感谢我的舍友黄智榕和蒋晓斌，感谢我的同门佟思明，感谢实验室同学王梓博、王舰等。他们或在生活为我提供帮助，或在我失落时为我提供鼓励，或在科研中为我提供经验和教训，我感谢研究生期间遇到的每位同学。

特别感谢我的女朋友王珊珊，从高中相识，一路相伴到现在。从准备考研到研究生期间进行科研以及编写论文，整个过程都离不开她的鼓励和支持。从大学开始异地，到研究生依然异地，能够陪伴的时间很少，而且由于研究生期间的科研项目繁多且困难，很多时候不能及时回复消息。能够顺利的度过这段时间，离不开她的包容和支持。当自己一度怀疑自己的时候，她会鼓励我，告诉我你也可以；当自己觉得对现状满意，开始洋洋得意之时，她会告诫我，自己仍然存在很多的不足。

感谢我的家人，尤其是已经去世的奶奶，是他们对我学习和生活的支持，让我勇敢地向前，无所畏惧。

我还需要感谢本科时期的一位老师-惠康华老师。在研究生统一初试结束之后，是惠康华老师耐心地帮助我准备研究生复试，一遍又一遍地陪我修改研究生复试答辩 PPT。我能够成功进入中科院软件研究所学习，离不开惠老师以及本科阶段所有老师的耐心教导，在这里感谢本科阶段的每位老师和同学。

感谢国科大和软件研究所提供的学习环境和硬件条件。

感谢强大的祖国为我提供良好的教育条件和幸福的生活环境，只有祖国强大，才能实现个人的理想和价值，祝愿我的祖国繁荣昌盛。

感谢所有曾经给予我帮助的人，在未来的日子中，我会更加努力，积极向上，做好本职工作，尽最大的努力回馈父母和师长，报效祖国，实现个人价值。