变异策略感知的并行模糊测试研究

邹燕燕[1,2], 邹　维[1,2], 尹嘉伟[1,2], 霍　玮[1,2], 杨梅芳[1,2], 孙丹丹[1,2], 史　记[1,2]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 中国科学院大学网络空间安全学院 北京 中国 100049

摘要　模糊测试(fuzzing)具备自动化程度高、可重现性好及易扩展等特点, 是软件漏洞挖掘的有效方法之一。针对其固有的测试盲目性和低效性, 一批先进的灰盒模糊测试方法被提出并应用在 AFL、AFLFast、Vuzzer 等工具中。随着高性能芯片和云计算技术的发展, 模糊测试可以充分利用其中蕴含的丰富并行计算能力、通过多实例并行的手段进一步提高单位时间内的综合测试效率, 典型的代表如 Xu 等人提出的多核并行方法、谷歌的 ClusterFuzz 等。但现有并行模糊测试方法, 由于不同测试实例在测试用例生成过程中缺少有效的控制, 导致生成的畸形样本冗余高、测试综合覆盖率低等问题。针对该问题, 本文提出了一种有效控制多测试实例间模糊测试过程的方案, 该方案以变异策略为基本粒度进行并行化, 定期同步不同测试实例间的有效畸形样本和优化变异策略应用比例, 减少不同测试实例间的测试冗余, 提高测试综合覆盖率。本文实现了一个变异策略感知的并行模糊测试框架, 并选择 AFL 作为基本模糊测试器, 使用 5 款开源软件及 LAVA-M 测试集的实验结果表明, 相同测试时间内本文的方法比 AFL 默认调度方法提高目标覆盖率达 132%、发现异常数量最多提高 50 余倍。

关键词　模糊测试; 漏洞挖掘; 变异策略; 并行化; 覆盖率
中图法分类号　TP309　DOI 号 10.19363/J.cnki.cn10-1380/tn.2020.09.01

Research on Mutator Strategy-aware Parallel Fuzzing

ZOU Yanyan[1,2], ZOU Wei[1,2], YIN Jiawei[1,2], HUO Wei[1,2], YANG Meifang[1,2], SUN Dandan, SHI Ji[1,2]

[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract　Fuzzing has become one of the most effective methods for mining software vulnerabilities due to its high degree of automation, high reproducibility, and good scalability. For its inherent test blindness and inefficiency, a number of advanced grey-box fuzzing approaches have been proposed and applied in AFL, AFLFast, Vuzzer and other tools. As the development of high-performance chips and cloud computing technologies, fuzz testing can make full use of the rich parallel computing capabilities contained therein and further improve the test efficiency through multi-instance parallelism. Typical representatives are Xu's multi-core parallel fuzzing method, Google's ClusterFuzz. However, the existing parallel fuzzing methods have problems such as high repetition rate of the deformed samples generated and low comprehensive test coverage due to the lack of effective control among different instances. Aiming at this problem, we first propose a scheme for effectively controlling the fuzz testing process among multiple instances. It parallelizes the mutation strategies as the basic granularity, regularly synchronizes the effective samples between different instances and optimizes the application ratio of the mutation strategy, reduces the test repeatability between different instances, and improves the coverage rate. We design and implement a parallel fuzzing framework which leverages AFL as the basic fuzzer, and evaluations using 5 popular applications and LAVA-M dataset, showed that, compared to default parallel fuzzing, our framework can improve test coverage rate up to 132%, and the number of crashes triggered increases as high as 50 times.

Key words　fuzz testing; vulnerability mining; mutator strategy; parallelization; coverage

1　引言

模糊测试是一种自动化软件测试方法, 最早由 Miller 等人提出, 用于测试 UNIX 程序健壮性[1]。该方法自动化程度高、易复现、扩展性好, 目前已演进成为一种普遍使用的软件安全漏洞分析技术。

针对某个待分析的目标程序, 模糊测试首先采用一定策略生成大量的非正常的输入样本, 即测试

通讯作者: 邹维, 硕士, 研究员, Email: zouwei@iie.ac.cn。
本课题得到中国科学院网络测评技术重点实验室和网络安全防护技术北京市重点实验室资助; 中国科学院重点实验室基金项目(No. CXJJ-17S049) 资助; 国家重点研发计划项目(No. 2016QY071405)资助。
收稿日期: 2018-08-27; 修改日期: 2019-01-25; 定稿日期: 2020-07-31

用例(又称畸形样本), 然后将它们逐一输入到目标程序中并监测程序运行是否崩溃(或进入非安全状态), 从而发现目标程序中潜在的安全缺陷。模糊测试的缺陷发现能力主要取决于测试用例的质量及测试效率。其中, 测试用例的生成方法可以分为两类: *基于变异* (mutation-based) 和 *基于生成* (generation-based)[2]。基于生成的方法需要测试人员对目标程序的输入进行深入的分析和建模, 测试用例的质量严重依赖于测试人员的经验, 测试成本高昂, 可扩展性面临较大挑战; 而基于变异的方法无需对输入进行手工建模, 仅需选取有限的输入种子, 即可通过一定的变异策略对输入种子进行变异, 从而产生大量的测试用例, 具有良好的通用性和可扩展性, 目前已经成为主流的测试用例生成方法。

模糊测试由于测试随机性、盲目性等特点[3], 存在测试效率低、覆盖率低等问题。为了提高模糊测试的漏洞挖掘能力, 研究人员展开了一系列研究[4-16]。传统黑盒模糊测试方法通过调节种子变异比例[5]、种子的调度算法[6]等技术提高限定时间内发现软件缺陷的数量。近年来广泛使用的基于反馈的模糊测试技术利用插桩[13, 15]、静态分析、动态分析等技术感知程序运行过程中的状态信息, 并用来指导种子选取[7, 9, 15]、变异位置[10, 12, 17]、变异值[12]等, 典型代表工具如 AFL 、Vuzzer[12]、AFLgo[9]等。上述方法致力于提升*单个模糊测试实例*的漏洞挖掘能力, 通过提高测试用例的质量提升模糊测试的效果。

随着软件规模和复杂度的提高, 传统的单模糊测试实例在测试效率及对目标程序的覆盖能力上均难以满足对现有复杂软件的测试需求。因此, 并行模糊测试技术得到广泛应用[18-24], 通过并行运行多个模糊测试实例完成针对相同目标的模糊测试任务。这其中的典型工作有, Xu 等人[23]利用的多核处理器的模糊测试方法, 谷歌提出了包含数百台机器的并行模糊测试系统 ClusterFuzz[24], 并以其为基础搭建了并行模糊测试服务 OSS-Fuzz[18], 微软推出的 Project Springfield[22]利用基于云的并行模糊测试为软件开发者提供安全缺陷检测服务。可见, 并行模糊测试已经成为工业界的主流选择。

并行模糊测试在充分应用传统的单实例模糊测试优化技术的基础上, 利用并行计算能力进一步提升了模糊测试效果。但是, 模糊测试依赖于海量的测试用例, 而现有的并行方法在测试用例的生成过程中欠缺多个模糊测试实例之间的有效协同, 导致不同实例间存在大量的冗余, 严重影响了并行模糊测试的效果。因此, 如何通过多模糊测试实例间的有效协同调度, 降低并行模糊测试实例间的冗余, 提高对目标软件的整体覆盖率是并行模糊测试面临的重要挑战。

针对基于变异的并行模糊测试中实例间的冗余问题, 本文提出了一种变异策略感知的并行模糊测试方法, 包括: 1)以变异策略权重模型为基础的差异化并行调度模型; 2)多模糊测试实例间基于有效样本同步的协同优化方法; 3)使用 5 个开源软件及 LAVA-M 测试集上的实验结果表明, 本文的方法可以将发现异常的数量最多提升 50 余倍, 测试覆盖率最高提高 132%。

2 研究动机

并行模糊测试的核心在于利用多个模糊测试实例同时对多组测试用例进行测试, 希望通过提高单位时间内处理的测试用例数量以提升模糊测试的综合效果。而在已有的并行模糊测试系统当中, 每个模糊测试实例都是相对完整的独立的模糊测试, 它们使用相同的变异策略集合来生成相应的测试用例, 尽管生成测试用例的过程具有一定的随机性, 但完全相同的变异策略集合使得不同的实例间存在非常多的冗余, 这启发我们对不同的模糊测试实例使用不同的变异策略集合。因此, 本节首先通过一组典型的实验数据验证目前并行模糊测试存在巨量冗余问题; 其次, 进一步的说明了如果每个模糊测实例仅使用单一的变异策略, 对该实例的测试效果影响极其微小; 最后, 我们发现不同的变异策略产生的测试用例具有极大的差异性, 即对任意两个变异策略来说, 它们产生的测试用例集合交集非常小。

2.1 并行模糊测试实例间的巨量冗余

本节我们通过实验验证了并行模糊测试不同实例间存在巨量的冗余问题。

我们采用 libxml2 中的 xmllint 程序进行实验, 实验使用了 16 个测试实例并行测试, 测试运行时间为 24 小时, 统计了每个测试实例对目标软件的覆盖率和并行的综合覆盖率, 结果如图 1 所示。

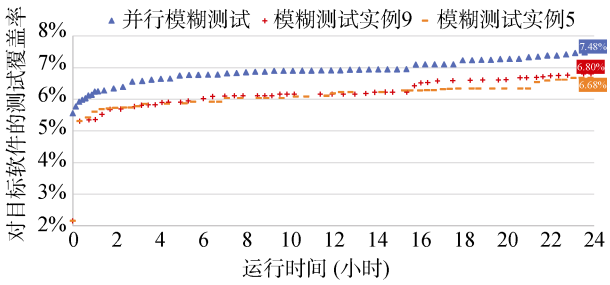


图 1 xmllint 并行模糊测试中不同实例的覆盖率

Figure 1 Coverage comparison between single-instance and overall fuzzing for xmllint

由于 16 个测试实例之间对目标软件的覆盖率的差异较小, 图 1 只列出了测试实例 9、测试实例 5 的覆盖率以及综合覆盖率随时间的变化趋势。可以看出, 单个测试实例的覆盖率分别为 6.8%和 6.68%, 而并行模糊测试的综合覆盖率只有 7.48%。可以看出, 不同的模糊测试实例对目标程序的覆盖大部分是相同的区域, 多个模糊测试实例覆盖汇总后对目标软件的覆盖提升很小, 即不同模糊测试实例间存在海量的冗余覆盖。

2.2 单一变异策略对单模糊测试实例的影响

本节将回答这样一个问题: 如果每个模糊测试实例仅使用单一的变异策略, 是否会影响该实例的测试效果?

我们使用测试用例对目标待测软件的覆盖程度来表示单个模糊测试实例的测试效果。为了保证结论的普适性, 我们使用了三类目标程序(xmllint、bsdtar、tiff2pdf)。对于每个目标程序, 分别使用 AFL 默认变异策略集合(包含 16 种变异策略)以及独立使用每一个单一的变异策略, 测试的初始输入样本分别采用 AFL 的默认 xml、rar、tiff 种子文件, 测试运行时间为 24 小时。对目标程序的最终的边覆盖率如图 2 所示。

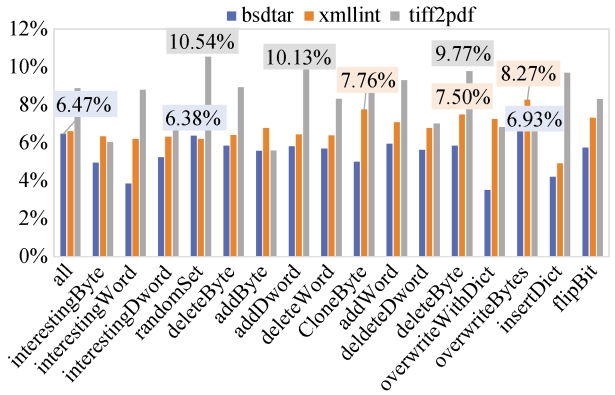


图 2 单一变异策略对目标程序测试覆盖率

Figure 2 Coverage comparison between single mutator strategy and strategies set for three different programs

图 2 中, 横轴表示使用的变异策略名称, 纵轴表示对目标软件的覆盖率。可以看出, 对于 3 个目标程序, AFL 默认组合变异策略的覆盖效果都不是最好的, 对于 xmllint、bsdtar、tiff2pdf, 最好的变异策略相比默认组合变异策略的覆盖率分别提高了 24.92%、7.11%、18.69%。同时, 对于不同变异策略, 也不存在 1 个变异策略同时使这 3 个目标程序达到最好覆盖率。此次实验显示, 对于 xmllint, 测试效果最好的三种变异策略分别是随机字节覆盖、字节复制、随机删除字节, 覆盖率分别达到 8.27%、7.76%、7.50%; 目标程序 bsdtar 测试效果最好的三种变异策略分别是随机字节覆盖、默认组合策略、对随机字节设置随机值, 覆盖率分别达到 6.93%、6.47%、6.38%; 目标程序 tiff2pdf 测试效果最好的三种变异策略分别是对随机设置字节、对随机字算术加运算和删除字节, 覆盖率分别达到 10.54%、10.13%、9.77%。此外, 实验显示对同一目标程序相同测试条件下延长测试时间所得的优势变异策略组合相同。

由此可见, 使用单一变异策略的模糊测试实例的测试效果与默认的使用一组变异策略集合的测试效果相当, 甚至优于默认的组合方案。此外, 不同变异策略对不同类型目标程序的作用效果不同, 即不同目标程序的优势变异策略并不一定相同, 通过精巧的选择不同的变异策略, 可以提高测试实例的测试效果。

2.3 不同变异策略生成测试用例的差异性

本节将回答这样一个问题: 对于不同的变异策略, 它们是否会大概率产生不同的测试用例?

同样使用目标程序(xmllint、bsdtar、tiff2pdf)重复上述实验。对于使用 AFL 默认变异策略组合所生成的测试用例不能够覆盖每一个基本块跳转边, 我们统计了为生成覆盖该跳转边的测试用例所使用的变异策略, 结果如图 3 所示。可以看出, 每种变异策略单独应用时都能够产生与默认执行不同的程序覆盖, 且每种变异策略生成的测试用例都能够产生不同的覆盖。以 xmllint 为例, 仅有 2.93%的基本块跳转边全部可由 16 种变异策略所生成的测试用例覆盖; 97.07%的基本块跳转边至少有 1 种变异策略所生成的测试用例不能覆盖; 有 44%的基本块跳转边仅可由唯一的 1 种变异策略所生成的测试用例覆盖。由此可见, 不同的变异策略总会产生不同的测试用例,

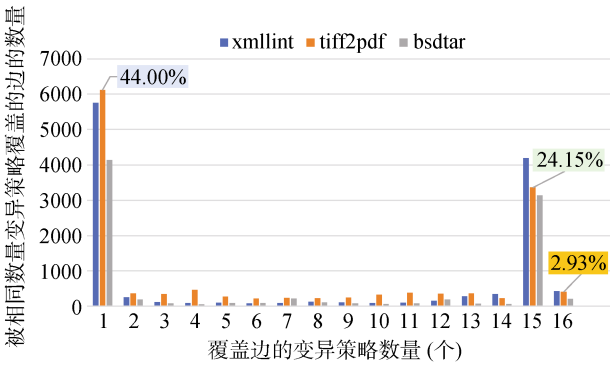


图 3 被相同数量变异策略覆盖的边的数量分布

Figure 3 Distribution of edges covered by different number of mutator strategies

即不同变异策略生成的变异策略存在很大的差异性。在各个采用不同变异策略的测试实例间共享所生成的测试用例, 可以进一步提高测试效果。

3 相关工作和背景知识

3.1 相关工作

基于变异的模糊测试技术通过对输入种子进行随机变异的方式生成测试用例对目标程序进行软件测试。目前大部分的黑盒/灰盒模糊测试工具[4, 8, 15-16, 25-28]都使用基于变异的方法生成测试用例进行测试, 应用广泛的模糊测试器包括 AFL[4] 、Taof[25] 、Sulley[28]等。

近几年随着基于反馈的模糊测试与遗传算法的结合产生了一批更加灵活、通用的工具, 使得模糊测试的过程更加智能和高效[3]。特别地, 基于覆盖率反馈的模糊测试技术取得了很大进展, 如 AFL[4]、LibFuzzer[16]、Honggfuzz[8]等漏洞挖掘工具测试效果显著、应用广泛, 已发掘软件的安全漏洞百余个, 在开源软件的安全测试中发挥了重要作用。

此外, 基于 AFL 改进的模糊测试工具层出不穷。Böhme 等人[7]提出的 AFLFast 利用马尔可夫链模型改善模糊测试中的种子调度, 提高了 AFL 的运行效率和相同时间内异常发现数目。Böhme 等人[9]进一步提出了基于静态分析和动态反馈方法的导向型模糊测试器 AFLgo, 提高对定向代码的覆盖能力。Rawat 等[12]提出的 Vuzzer 以及 Li 等人[10]提出的 Steelix 使用动静态数据流分析, 通过对 cmp 等指令插桩分析对模糊测试过程进行反馈导向, 从而提高生成覆盖更深层次的测试用例概率。Gan 等人[15]提出的 CollAFL 使用更精确的插桩解决 AFL 覆盖率统计中的路径冲突问题, 提高 AFL 的测试覆盖率及漏洞挖掘效果。Chen 等人[17]提出的 Angora 使用字节级污点分析、基于上下文的分支统计及基于梯度下降方法的搜索算法等提高模糊测试过程中的分支覆盖能力。

在结构化类型的测试用例生成方面, Wang 等人[14]提出的 Skyfire 面向高度结构化输入类的目标程序提出一种数据驱动的种子生成方法, 通过从大量的已知样本中学习语法信息, 生成覆盖良好的种子, 然后利用变异方法进行模糊测试用例生成。此外, 随着机器学习的发展和广泛应用, 微软的 Godefroid 等人[29]提出使用机器学习的方法自动生成测试用例, Rajpal 等人[30]使用神经网络从模糊测试过程中学习和预测对输入的变异位置提高测试效果。

在并行模糊测试方面, AFL 支持多实例的并行模糊测试, 且通过实例间的有效样本同步的方法提高测试的效果。Xu 等人[23]提出的多核机器上的并行模糊测试方法通过创建新的操作系统原语提高多核机器上多实例并行模糊测试的效率。谷歌的 ClusterFuzz[24]平台支持数百台机器的并行测试, 每天生成五千多万测试用例。除此之外, 谷歌的 OSS-Fuzz[18]、微软的 Project Spingfield[22]均利用基于云的并行模糊测试为软件开发者提供安全缺陷检测服务。

目前这些研究主要针对模糊测试本身的输入样本、测试过程中的调度反馈、输入样本选取等方式进行优化并取得较好的结果; 且目前的并行模糊测试并未对测试实例间进行有效的协同控制, 因此会由于模糊测试中变异的随机性和盲目性导致其测试深度难以提高。本文通过对并行模糊测试过程中变异策略的并行调度和同步控制方法进行研究, 提高测试覆盖率和发现漏洞的能力。本文的工作与上述先进模糊测试技术和工具之间是正交关系, 可以在利用上述技术的基础上进一步提高并行化应用中的漏洞挖掘效果。

3.2 变异策略分类

目前通用的模糊测试框架中包含变异策略丰富多样, 我们通过对目前广泛使用的先进的模糊测试器进行分析, 对变异策略进行汇总, 并根据模糊测试过程的有序性和随机性对其进行分类, 分为有序变异和随机性变异。表 1 所示为两种类型变异策略的描述及包含的典型变异策略。

表 1 变异策略分类

Table 1 Classification of mutator strategies

名称	描述	典型变异策略
有序变异	对初始数据进行按序变异, 依次对每一位、字节、字等进行变换, 该类变异是某一变异策略在输入数据上的完整应用, 重复执行生成数据相同。	bitflip, bytefilp, arithmetic, interesting value
随机变异	对初始数据随机选取变异位置、随机变异值进行变换, 该类变异产生数据随机、重复运行生成数据不一定相同。	bitflip, bytefilp, arithmetic, interesting value random set, delete, insert, clone, overwrite, splcing

3.2.1 有序变异策略

有序变异是指变异策略作用在输入样本的位置、变异后的值等是有序且确定的, 针对相同的输入多次执行有序变异后得到的输入样本数据集是相同的。例如有序位翻转变异(bitflip), 位翻转变异是针对某一位进行翻转, 由 0 翻转为 1 或者 1 翻转为 0, 因此, 在选定输入的特定位置使用位翻转变异后的结果是确定的。另外, 该类变异方式耗时长, 需要的迭

代次数多。

图 4 所示为有序位翻转的确定性变异过程实例。有序变异过程中的按序位翻转的常见变异规则是从样本第一个位开始进行一定长度(如 1 位、2 位、4 位、8 位、16 位或 32 位)的 0-1 翻转, 一次调度的变异位数是固定的, 单实例上的分时、随机多次调度没有记录当前变异进度, 是对上一次的位翻转过程的重复。即使通过引入全局变量对当前变异位置进行记录, 这种随机调度也无法保证需要长时间、无间断执行的位翻转策略的变异深度。整体上, 局限在这样一个有限变异深度上的大规模并行模糊测试过程必然存在着大量无效重复。

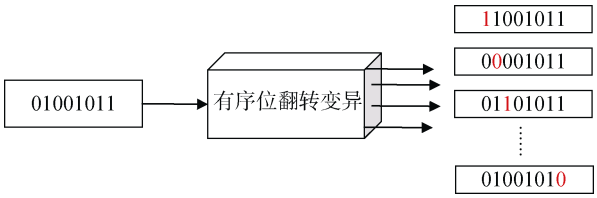


图 4 有序位翻转变异示例

Figure 4 An example of bitflip mutation

针对该类变异策略我们在并行模糊测试中应该对其进行单独处理, 需要控制防止多个实例重复执行相同的变异。另外, 可以采用多个实例同时执行但分区域进行变异的方式来提高有序变异的效率。目前并没有针对该类应用方式进行模糊测试优化技术研究或实现。

3.2.2 随机变异策略

随机性的变异策略对输入测试样本数据的变异是随机的, 变异后的值具有不确定性, 因此多次迭代执行或者并行测试产生的冗余性和重复性是不确定的, 该类变异策略在并行执行过程中可以不考虑其实现中的重复性。但是对于具有随机种子类型的变异策略, 可以通过控制并行测试节点的种子分布来提高变异后数据的分散性。

图 5 所示为随机变异过程中的随机插入变异示例, 对于给定的输入数据, 经过随机插入变异后生成的数据为非固定值, 且每次变异生成的数据及变异的位置都是非固定的。对随机变异而言, 间断性的多次调度和一次调度长时间执行两种方式的测试效果无明显区别。因此, 对于随机变异策略的并行可以采用并行多实例分别调度执行。

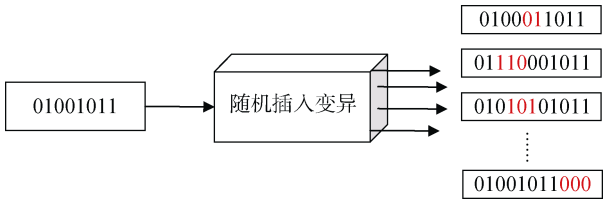


图 5 随机插入变异示例

Figure 5 An example of random insert mutation

此外, 现有的变异策略在其变异过程中会对不同类型的原始数据产生不同的变异效果, 如算数运算类的变异策略对整型数据的变异更容易触发新的数据值。模糊测试中的随机变异策略对输入程序的作用效果与程序自身的输入样本的类型及数据组成格式相关(如图 2 所示)。本文将结合变异策略对目标程序的作用效果进行分析建模, 根据不同类型应用程序的优势变异策略形成并行调度模型, 提高并行模糊测试整体运行效果。目前模糊测试研究中并未开展基于变异策略的并行调度技术研究。

4 变异策略感知的并行模糊测试

为了便于对本文提出的变异策略感知的并行模糊测试方法进行描述, 4.1 节对本文的并行模糊测试系统及变异策略等进行形式化描述; 4.2 节对本文的整体方法进行概述; 4.3 节针对具体的并行化方法进行详细展开描述; 4.4 节介绍并行模糊测试的变异策略感知的整体调度模型。

4.1 形式化描述

首先给出并行模糊测试系统、变异策略集合等概念的定义和形式化描述, 便于后续变异策略感知的并行模糊测试及调度算法的描述。

定义 1. 对于传统的模糊测试器 F, 定义如下:

$$\mathrm{F}=(M,S_{\mathrm{f}})$$

其中, M 表示支持的变异策略集合, S_{f} 表示模糊测试过程中实际使用的变异策略集合, 分别定义如下:

$$M=\{\mathrm{MU}_1,\ \mathrm{MU}_2,\ \cdots,\ \mathrm{MU}_{\mathrm{K}}\}$$

$$S_{\mathrm{f}}=\{\mathrm{MU}_{\mathrm{o}},\ \mathrm{MU}_{\mathrm{p}},\cdots,\ \mathrm{MU}_{\mathrm{q}}\},\ \text{且}S_{\mathrm{f}}\subseteq M$$

其中, MU_i 表示变异策略 i, K 表示模糊测试变异策略的数量, S_{f} 是 M 的子集。

定义 2. 对于并行模糊测试系统 PF, 定义如下:

$$\mathrm{PF}=(\mathrm{M},\ \mathrm{S_C},\ \mathrm{S_{ch}})$$

其中, M 表示其支持的变异策略集合(同定义 1); S_C 表示集群 C 中各个测试实例节点上调度的集合; S_{ch} 表示针对测试实例的每个变异策略选取概率向量。其分别定义如下:

$$S_C=\{S_{f1},\ S_{f2},\cdots,\ S_{fN}\}$$

$$S_{fi}(M,\ S_{ch})=\{MU_o,\ MU_p,\cdots,\ MU_q\},\ 且S_{fi}\subseteq M$$

$$S_{ch}(M,\ i)=(p_{i1},\ p_{i2},\cdots,\ p_{iK})$$

其中, S_C 是 N (测试实例节点数量)个 S_f 组成的集合, 每个 S_{fi} 表示测试实例节点 i 调度使用的变异策略集合, 该集合由总体变异策略集合 M 和每个变异策略的调度概率函数 S_{ch} 计算得到, S_{fi} 是 M 的子集。$S_{ch}(M,\ i)$ 表示测试实例 i 上变异策略的调度概率向量, p_{ij} 表示在测试实例 i 上变异策略 MU_j 被调度的概率。

模糊测试的有效性很大程度上依赖于模糊器所使用的变异策略, 根据 3.2 描述, 变异策略按照变异对输入样本作用的有序性分为有序变异策略和随机变异策略, 具体定义如下。

定义 3. 对于给定的模糊测试器的变异策略集合 M, 其中包含的有序变异策略集合 MD 和随机变异策略集合 MR 分别表示为:

$$MD=\{MDU_1,\ MDU_2,\ \cdots,\ MDU_p\},\ 且MD\subseteq M$$

$$MR=\{MRU_1,\ MRU_2,\ \cdots,\ MRU_q\},\ 且MR\subseteq M$$

其中, p 为有序变异的变异策略数量, 其中 q 为随机变异的变异策略数量。

4.2 方法概述

本文提出一种变异策略感知的并行模糊测试方法, 通过对测试实例所采用的变异策略进行动态调度并进行多实例间有效样本同步, 提高不同测试实例间生成测试用例的差异性, 降低多实例并行测试中的冗余。图 6 所示为变异策略感知的并行模糊测试架构, 包含并行调度和若干测试实例。

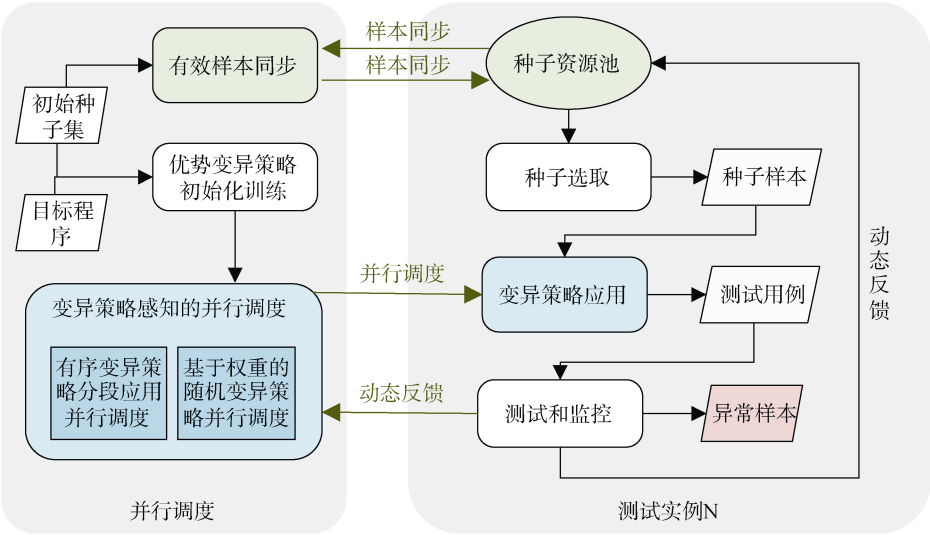


图 6 变异策略感知的并行模糊测试方法

Figure 6 Mutator strategy-aware parallel fuzzing

并行调度负责实现对并行模糊测试过程的整体协调和调度, 主要功能包括优势变异策略的初始化训练、变异策略感知的并行调度以及有效样本的同步。优势变异策略初始化训练负责将各个随机变异策略单独训练执行, 根据变异策略对目标程序的测试效果筛选优势变异策略, 计算并更新各个变异策略的权重值。变异策略感知的并行调度功能主要是利用测试过程中各个测试实例节点的反馈数据实现有序变异策略分段应用并行调度、基于权重的随机变异策略并行调度。有效样本同步通过周期性从各测试实例进行有效样本收集, 并对样本进行去重过滤后分发给各个测试实例, 从而实现各个测试实例执行效果的交叉组合应用。

测试实例是模糊测试器的具体执行实例, 其内部采用基于反馈的灰盒模糊测试器作为引擎执行(本文基于 AFL 实现), 测试过程包括从种子资源池中选取种子样本, 利用变异策略对种子样本进行变异并生成测试用例, 然后对目标程序进行测试和监控, 利用测试和监控的动态数据反馈指导种子资源池的调度和选取。此外, 测试实例将测试过程中的动态数据反馈给并行调度指导变异策略调度。

综上所述, 本文提出的模糊测试并行调度方法根据变异策略种类及应用方式不同, 主要研究内容包含以下三个方面:

(1) 针对有序变异策略对同一输入样本执行的确定性和重复性等问题, 使用有序变异策略分段应

用的方式减少不必要的重复变异, 提高效率;

(2) 针对不同变异策略对特定目标程序的应用效果分析, 对每类变异策略在目标待测程序中应用, 获取测试结果并建立变异策略的权重模型, 并根据模型对每个测试实例分配不同的变异策略集合;

(3) 为了综合应用不同测试实例的有效变异结果, 利用同步方法实现对各个实例间有效用例(能够增加新的覆盖的测试用例)同步, 从而在各实例间实现不同变异策略的交叉组合作用, 进一步提高模糊测试的综合效用。

4.3 变异策略感知的并行模糊测试

基于 4.2 节方法描述可知变异策略感知的并行模糊测试包含有序变异策略并行化、随机变异策略并行化以及有效样本同步三个方面, 本节将分别具体展开描述。

4.3.1 有序变异策略并行化

有序变异策略对不同测试样本使用相同变异策略会生成相同的测试用例, 此外, 由于其变异过程对整个样本进行依次变异, 有序变异的执行周期长、效率低。针对有序变异策略的上述特点, 在并行模糊测试过程中可以采用多种去重复、低冗余的并行模式进行调度。

基于有序策略重复性, 可以利用单测试实例运行的模式进行调度, 在并行模糊测试任务执行过程中采用单个测试实例应用有序变异策略执行, 其他实例只使用随机模式执行, 该方法可以提高运行的效率, 降低无效重复变异。但是由于有序变异策略能够对输入样本进行更充分的全覆盖的变异, 结合本文的基于有效样本同步的变异策略交叉组合应用技术实现多测试实例的变异效果的充分组合。

另外, 针对有序变异策略对输入样本的全覆盖迭代变异特点, 亦可以采用将对样本进行分段、分策略应用的模式实现不同实例的并行调度执行。通过将样本变异空间划分为连续块, 针对单个块空间, 采用单实例连续变异, 确保其变异深度。针对整体变异空间, 采用基于块单元的多实例并行, 提高模糊测试的效率, 发挥并行模糊测试的优势。该方法能够提高有序变异的应用效率, 同时, 利用执行过程中的多节点同步策略能够综合各个节点对样本不同区间段的变异效果, 实现高效测试效果。

定义 4. 对于给定的初始样本 T, 其长度为 L_{T}, 其整个样本变异区间为整数区间 $[0, L_{\mathrm{T}}-1]$; 对于样本 T 中的第 i 块样本空间 $Block_i$, 其中该块的长度 ℓ_i, 所有块的长度的和等于测试用例样本长度, 即 $\sum_{i=1}^{M}\ell_i = L_{\mathrm{T}}$。

具体来说, 利用定义 3 对输入样本空间划分为多个互补的块空间(第 i 块空间表示为 $Block_i$), 划分的块数量等于集群 C 中测试实例的数量 N, 每一个块对应一个测试实例, 测试过程中每个实例对其分配的块空间运用所有的有序变异策略进行样本变异和用例生成。例如, 对于测试实例 i 使用的有序变异策略 MD_i=MD。对于各个测试实例平均划分的调度方法每个块的长度相同, 块地址长度用 ℓ 表示, $\ell = L_{\mathrm{T}}/N$, 该节点数据变异在样本 T 中的区间表示为 $[i*\ell, \min(i*\ell+\ell, L_{\mathrm{T}})]$, 其中该区间在并行调度中进行区间分配。因此, 对于测试实例 Node_i 其调度的组合表示为 $(\mathrm{Node}_i, Block_i)$。

4.3.2 随机变异策略并行化

针对随机变异策略, 每个测试实例中随机策略的变异结果具有不确定性, 且不同变异策略变异效果与输入特征、程序特点等具有一定的相关性。因此, 随机变异策略的并行调度方法需结合程序运行特点和不同策略在程序测试中的运用表现进行分配调度, 各个测试实例需要分配的变异策略及组合方式将根据具体测试需求和具体调度算法区别。

本文针对随机变异策略的并行调度将从各实例变异策略分离应用、基于权重的变异策略并行调度及变异策略交叉组合应用角度进行综合调度, 实现各类变异策略在不同目标程序上的深度应用, 并最大化发挥各变异策略自身的优势。

变异策略分离应用。针对随机变异策略引入并行技术, 通过多实例上变异策略的分离应用提高每个变异策略的变异深度, 从而实现对目标程序在指定输入样本的深度测试。对于单个测试实例的随机变异策略集合为 MR_i, 且满足

$$MR_i \subset MR$$

$$MR_i \neq \phi$$

首先, 根据模糊测试器包含的随机变异策略 MR 和现有集群资源 C, 利用平均划分的方法对变异策略进行分配, 实现不同测试实例中的随机变异策略初始化调度。测试实例的随机变异策略集合 MR_i 满足:

$$MR_i = \begin{cases} \bigcup\limits_{j=i\%M} MRU_j, & M \leqslant N \\ \bigcup\limits_{k=M/N} \bigcup\limits_{k=0} \bigcup\limits_{j=i+N*k} MRU_j, & M > N \end{cases}$$

基于权重的变异策略调度。采用基于权重的方

法对变异策略的调度优先级及应用并行度计算, 对于指定输入类型的目标程序衡量每种变异策略分离应用的作用效果形成调度因子, 从而实现随机变异策略不同权重和规模的运用, 让更多的计算资源应用在更有效变异策略上进行测试用例生成, 优化并行模糊测试执行过程的资源分配, 提高模糊测试漏洞挖掘效果和对目标程序的测试深度。

定义 5. 随机变异策略的并行调度依据不同变异策略对目标程序的作用效果进行概率选取。在测试节点 Node_i 上变异策略 MRU_j 在模糊测试执行过程中被选取应用的概率表示为 p_{ij}, 且满足: $0 \leqslant p_{ij} \leqslant 1$ 和 $MRU_j \in MR_i$。

基于权重的变异策略调度利用测试过程中的数据反馈计算各个变异策略在目标应用中的作用效果, 从而更新测试过程中各个变异策略的使用频率及组合频率。本文在权重的计算过程中考虑的因素主要有测试过程触发的异常数目、测试对目标程序的覆盖率两个方面。具体来说, 不同变异策略的权重满足:

$$p_{ij} = \frac{B_j}{\sum_{k=0}^{k<|MR_i|} B_k} \cdot \alpha + \frac{C_j}{\sum_{k=0}^{k<|MR_i|} C_k} \cdot \beta$$

$$\alpha+\beta=1$$

$$0 p_{ij} \leqslant 1 \alpha,\ \beta p_{ij} \leqslant 11$$

其中 B_j 表示变异策略 MRU_j 触发的异常数目, C_j 表示变异策略 MRU_j 测试对目标程序的测试覆盖率, α 表示触发异常对调度的影响因子, β 表示覆盖率对变异策略调度的影响因子。

单实例变异策略交叉组合应用。在模糊测试应用过程中单个样本的测试用例生成可以采用多个变异策略的组合应用生成, 从而实现测试用例生成的复杂性及对目标程序的覆盖能力。

定义 6. 测试实例 Node_i 的变异策略组合 MR_i 的概率分布数组表示为 $\mathrm{P}=(p_{ij})$, 且满足

$$0 \leqslant \mathrm{p}_{ij} \leqslant 1$$

$$\sum_{j=1}^{|MR_i|} \mathrm{p}_{ij} = 1$$

$$1 \leqslant |MR_i| \leqslant |MR|$$

变异策略多实例并行应用的初始化过程中各个变异策略单独执行, 且每种策略权重相同。在测试执行过程中通过分析执行数据更新各个权重的基础上, 并行调度引擎更新各个实例变异策略选取的组合和权重。对于测试实例 Node_i 的随机变异策略组合使用 Top K 优势变异策略($\mathrm{C}_{\mathrm{top}}$ 表示前 K 个优势变异策略的最低阈值)组合的方式进行, 在各个测试节点增加前 K 变异策略的方式调度, 具体表示满足:

$$MR_i = MR_{\mathrm{i}} \cup \left(\bigcup_{\mathrm{C_k} > \mathrm{C_{top}}} MRU_{\mathrm{k}} \right)$$

4.3.3 多实例间有效样本同步

变异策略感知的并行模糊测试执行中, 每个测试实例运行单个或多个组合变异策略, 这可以一定程度上提高变异策略在目标程序上生成的测试用例覆盖更广目标代码的概率。由于不同变异策略的分布式应用使得不同测试实例上测试覆盖目标程序的区域不同, 为了提高整个并行测试任务的综合测试深度和漏洞挖掘效果, 在测试过程中使用有效样本同步的方法对变异策略进行交叉组合应用。图 7 所示为变异策略交叉组合应用的示例。在两个测试实例中的模糊测试器使用的变异策略分别是位翻转变异和随机插入变异, 两者分别测试得到的测试样本分别为样本 1 和样本 2。使用有效样本同步后, 两个测试实例的输入样本集合得到扩展, 然后测试实例继续进行测试后可以得到样本 3 和样本 4。对比不使用同步交叉的方式, 在实例 1 上只使用位翻转变异从样本 1 不可能变异得到样本 3, 在实例 2 上的随机插入方法也无法从样本 2 变异出样本 4。

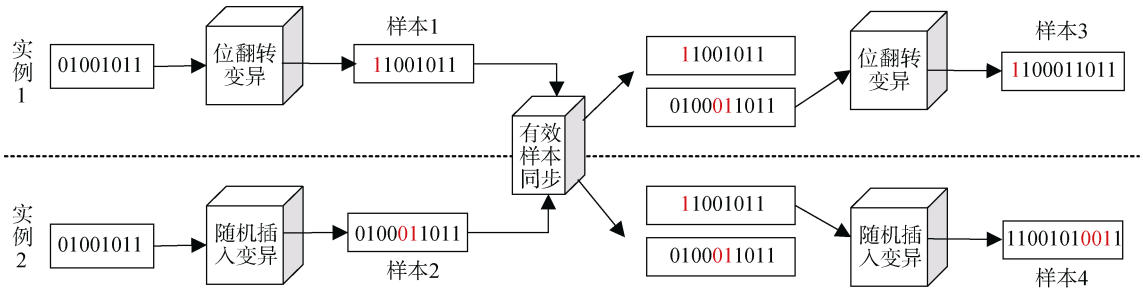


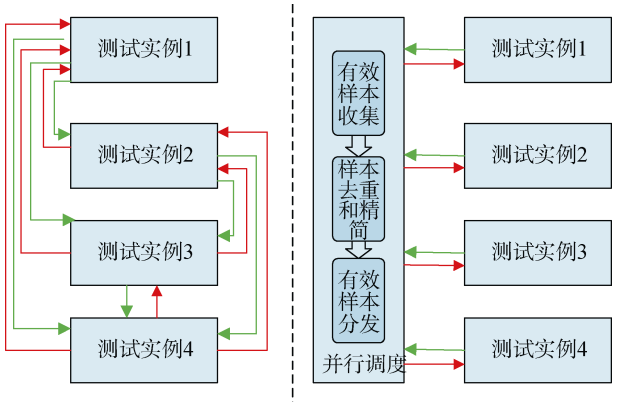
图 7 变异策略交叉组合应用示例

Figure 7 An example of cross mutation

由此可见, 在使用变异策略并行调度的基础上使用有效样本同步方式的变异策略交叉组合应用可以实现多种变异策略的测试能力的深度融合, 从而生成单个变异策略变异能力之外的测试用例, 能够在原有变异策略的基础上提高测试能力。

本文在多实例间有效样本同步实现变异策略交叉应用过程中采用基于时间片触发的有效样本同步方式来实现, 基本原理如图 8 (b)所示, 其中绿色箭头表示样本收集, 红色箭头表示样本分发。

多实例间有效样本同步主要由并行调度节点负责管理和实现。调度节点通过数据收集接口周期性地从各实例收集有效用例; 然后, 利用样本去重与精简功能对所有实例汇总的测试样本进行精简去重, 实现对冗余用例或者无新覆盖能力的用例的筛减, 最终保留所有能够对目标程序最大覆盖的最小样本集合。最后, 通过有效样本分发接口实现对同步且精简后测试用例向各实例的分发, 并由各个测试实例完成对所分发样本的后续交叉组合变异, 从而实现实例间的变异策略交叉组合应用效果。



(a) AFL样本同步方法 (b) 本文的样本同步方法

图 8 多实例间有效样本同步方法

Figure 8 Methods for effective samples synchronization between different instances

传统的样本同步方法各个测试实例间依次进行样本传递和分发, 图 8 (a)所示为 AFL 并行测试的样本同步方法, 通过各个实例间相互同步所有的有效样本。由图 8 的可以看出, 在各实例的平均样本数量为 N 的情况下, AFL 完成样本同步的开销为 $O(N^2)$, 而本文同步开销为 $O(N)$。因此, 本文使用的多实例间有效样本同步方法能够有效减少样本在各个测试实例间的传递次数, 并通过调度节点的样本精简方法减少实例间传递的样本数量, 能够实现测试用例同步效率的提升。

4.4 并行模糊测试调度模型

变异策略感知的并行模糊测试技术通过并行调度实现对各个测试实例的调度, 利用测试过程中的执行反馈信息动态调整变异策略的分配及在各实例中变异策略的组合方式, 从而实现对模糊测试过程中变异策略的动态调度和调整。并行模糊测试执行过程中并行调度方法如算法 1 所示。

算法 1: 变异策略并行的调度算法

1. INPUTS: Target P、Input Seeds S、Cluster C
2. $N = len(C)$, P=(0)
3. //初始化变异策略调度方案
4. FOR i IN $range(1, N)$:
5. //初始化测试节点使用的有序变异策略
6. $MD_i = InitializeMD(Node_i)$
7. //初始化测试节点使用的随机变异策略
8. $MR_i = InitializeMR(Node_i)$
9. FOR j IN $range(1, |MR_i|)$:
10. $p_{ij} = 1/|MR_i|$
11. $Schedule(Node_i, MD_i, MR_i, P)$
12. WHILE Timer & Running :
13. //动态收集测试过程中的运行数据
14. $GuiderInfo = CollectInfo(C)$
15. //动态更新各测试节点变异策略组合
16. FOR i IN $range(1, N)$:
17. MR_i = UpdateMR($Node_i$, GuiderInfo)
18. //动态更新测试节点中变异策略权重
19. FOR i IN $range(1, N)$:
20. FOR j IN $range(1, |MR_i|)$:
21. $p_{ij} = Compute(MRU_j, GuiderInfo)$
22. $Schedule(Node_i, MD_i, MR_i, P)$
23. $SyncSeeds(C)$

首先, 并行调度根据测试实例的数量、模糊测试器的有序变异及随机变异的变异策略种类对调度算法分别调用 *InitializeMD* 和 *InitializeMR* 进行初始化, 并调用 *Schedule* 函数实现对 $Node_i$ 节点上模糊测试实例的启动执行。

初始调度运行一定时间间隔后使用 *CollectInfo* 对测试过程的覆盖率、测试异常数量等数据进行收

集和分析处理; *UpdateMR* 根据收集的运行信息实现对各个测试实例上选取的随机变异策略组合进行更新; *Compute* 依次对 MR_i 中的所有变异策略进行权重计算和更新, 从而实现对各个测试节点上的模糊测试实例使用的变异策略组合及权重分配进行调整; *Schedule* 实现对 $Node_i$ 实例上模糊测试执行过程的动态更新。

最后, 针对并行测试实例间变异策略交叉应用方法, 调用 SyncSeeds 实现各个实例的有效样本同步和分发, 从而综合各个实例测试结果实现变异策略的深度交叉应用。

5 系统实现

本文基于 VARAS(Vulnerability Analysis and Risk Assessment System)实现了该并行模糊测试系统, VARAS 是由中国科学院信息工程研究所自主研制和打造的面向网络空间安全漏洞分析、渗透测试和风险评估的大规模协同化综合信息处理系统。

本文实现的系统支持对多实例并行的模糊测试的调度, 系统架构采用主从模式实现, 由并行调度节点作为主控端和多个测试实例节点作为从控端组成。系统的调度功能包括并行调度节点的调度引擎以及测试节点的执行引擎, 分别负责多测试实例调度及单实例内部的模糊测试执行过程调度。此外, 并行调度节点包含的调度模型是支持调度引擎实现的算法, 包括根据目标程序类型或输入数据的特点实现的多种不同的并行调度算法。并行模糊测试系统是一个针对模糊测试并行化应用的通用调度系统, 支持本文提出的基于变异策略的模糊测试并行调度, 同时, 支持用户提出的其他调度算法的集成应用。该系统的并行调度节点和测试实例节点都由三部分组成: 引擎、模型和数据交互接口, 系统整体架构如图 9 所示。

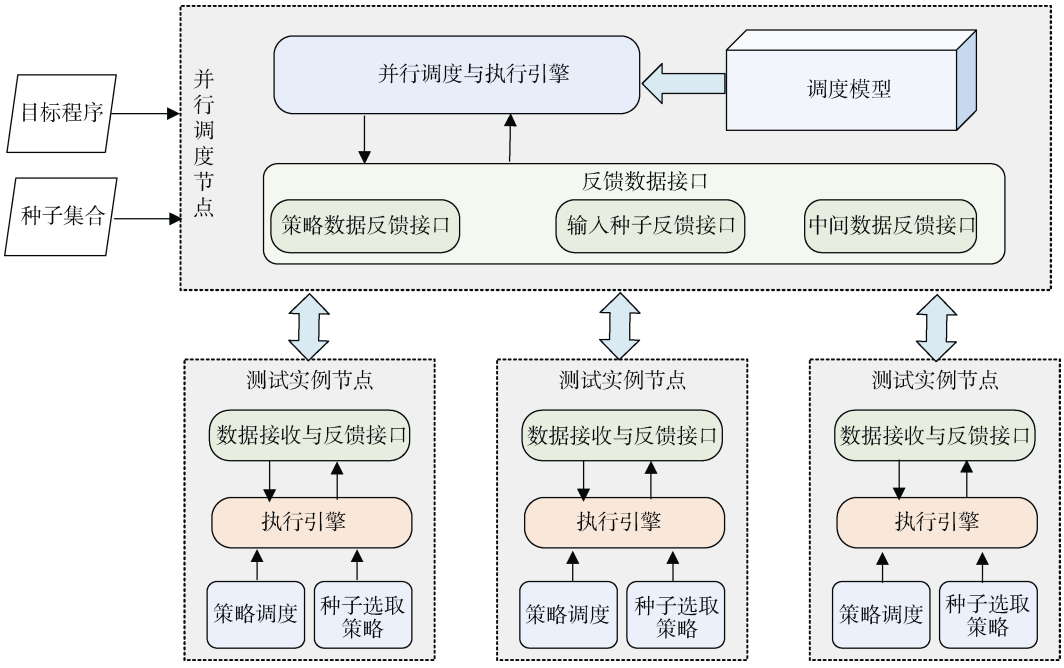


图 9 并行模糊测试系统整体架构

Figure 9 Architecture overview of parallel fuzzing system

并行调度节点: 并行调度节点包括并行调度与执行引擎、调度模型及反馈数据接口三个部分组成, 其中反馈数据接口包含策略数据反馈接口、输入种子反馈接口、中间数据反馈接口等组成, 支持用户的自定义扩展。

并行调度与执行引擎主要负责实现对并行模糊测试执行过程的动态调度和管理, 利用调度模型的调度算法及执行反馈数据实现对执行过程的动态调整与优化。调度模型是实现并行模糊测试调度的调度算法库, 包含基于变异策略的并行调度算法模型, 且支持用户扩展该模型实现满足用户调度需求或测试需求的模型, 该模型将作为并行调度的指导策略实现对整个过程的运行指导及实现。反馈数据接口是实现并行调度节点与测试节点通信与交互的接口, 实现对不同类型数据的接收、发送和统计处理等, 实现控制节点调度指令的传达及测试结果的接收和反馈。反馈数据接口的实现与调度模型依赖的数据形式相关, 支持用户的扩展。

测试实例节点: 测试实例节点是模糊测试的中测试实例的执行载体, 包含数据接收与反馈接口、测试执行引擎、策略调度及种子选取策略等不同功能模块组成。测试节点利用与主控节点的交互及模糊测试执行引擎的调度控制实现单实例测试执行。

执行引擎是具备测试策略选择等功能的模糊测试器执行引擎, 该部分支持用户开发或者定制化后的模糊测试器, 通过配置文件或参数形式指定测试执行模式, 实现低冗余测试执行。策略调度及种子选取策略是实现并行测试的单实例调度执行模型, 支持单实例的测试调度, 支持用户的自定义扩展, 本文描述的是策略调度方法, 种子选取方法不在本文描述范围之内。接收与反馈接口是实现模糊测试器与并行调度节点交互的通信接口, 通过接收控制指令实现对测试引擎的调度, 通过执行数据交互实现对测试过程和执行结果的反馈, 从而实现测试过程的动态反馈和优化。

本文基于该并行模糊测试系统架构, 在并行调度节点实现了变异策略感知的并行调度模型(如算法 1 所述)。在测试实例节点上基于 AFL 模糊测试器实现了支持变异策略调度控制的模糊测试执行, 并实现与并行调度节点的数据交互和反馈功能, 为变异策略感知的并行调度提供支持。

6 实验

模糊测试漏洞挖掘的效果的主要衡量指标包括测试对目标程序的覆盖率及限定时间内挖掘漏洞的数量两个方面, 本文从这两个方面对所提出的变异策略并行的模糊测试调度方法进行实验验证。

6.1 测试环境和测试集

我们利用目前应用广泛的模糊测试器 AFL 作为变异策略并行调度的模糊测试中测试实例的模糊测试工具, 该工具本身在单实例应用中效果卓越, 是目前主流和广泛应用的模糊测试工具。本文选取实际广泛应用的开源目标软件作为测试集验证方法的有效性, 包括 libxml2、libtiff、libarchive 程序。此外, 为了验证本文的并行调度方法的漏洞挖掘能力提升效果, 选取含有已知注入漏洞的测试集 LAVA-M[31](包含 who、base64、md5sum、uniq 四个测试程序)进行并行模糊测试漏洞挖掘能力验证。本文实验所用测试集及工具如表 2 所示。

LAVA-M 测试集: LAVA-M 包含 4 个注入漏洞的 Linux Utilities 程序, 即 base64、md5sum、uniq 和 who 程序。该测试集是由 LAVA 通过向源代码自动化注入难以触发的漏洞后编译生成的可执行程序, 一般用于验证模糊测试器、符号执行等工具的漏洞挖掘能力。LAVA 的作者已经验证在 FUZZER(基于反馈的模糊测试器)和 SES(基于约束求解等的符号执行方法)测试方法下不能够完全发掘程序内注入的漏洞。最近的一些模糊测试器(如 Vuzzer、Steelix 等)相关研究工作也通过该测试集对其效果进行验证, 本文利用该测试集对基于变异策略的并行调度方法的漏洞挖掘能力进行验证。

表 2 测试集

Table 2 Test suits

类别	名称	可执行程序	版本
模糊测试器	AFL	afl-fuzz	2.52b
测试目标	libarchive	bsdtar	3.3.2
	libtiff	tiff2pdf	4.0.9
	libxml2	xmllint	2.9.7
	OpenJPEG	opj_decompress	2.3.0
	ImageMagick	convert	7.0.8.1
	LAVA-M	base64, md5sum, who, uniq	coreutils-8.24

真实应用程序测试集: 本文选取 5 款广泛应用的开源软件作为测试目标进行测试, 分别是 libxml2、libtiff、libarchive、OpenJPEG、ImageMagick, 选取的版本为实验时最新版本(如表 2 所示)。本文选取的 5 款软件处理不同类型的输入格式, 包括结构化数据格式 xml、图片格式 tiff、压缩文件格式 tar 等。我们利用不同类型目标程序来验证该调度方法在不同类型输入程序的应用具有通用性, 此外, 实验选取的目标软件在 OSS-Fuzz 项目中得到持续的漏洞挖掘测试, 我们在最新版本的程序上进行测试以验证本文的并行调度算法对目前先进模糊测试器并行的测试覆盖和漏洞挖掘效果的提升能力。

模糊测试器: 本文采用的模糊测试器是 AFL, 为了实现基于变异策略的并行调度方法, 我们在现有代码基础上进行变异策略选择的扩展修改, 实现对变异策略并行化调度支持。本修改在保持 AFL 原有先进性基础上实现对基于变异策略选择的动态并行调度的支持。

测试环境: 本文的并行调度实验在基于 OpenStack[32]的漏洞分析平台 VARAS 支持下展开, 本文每个模糊测试的实例执行在一个虚拟机节点上, 每台虚拟机配置为 Ubuntu16.04 64 位 2 核 CPU 4G 内存。每组并行调度实验的并行规模为 17。本文实验中选取的并行规模是基于 AFL 模糊测试器的变异策略种类的 1 倍进行设定, 在实际应用中用户可基

于硬件资源和需求扩展或调整规模。

6.2 实验结果

实验分为两组对本文提出的调度方法的应用效果进行验证: 一组是对真实应用程序的不同调度方法实验对比, 用于验证本文提出的基于变异策略调度在不同组合调度模式下的测试效果, 包括目标程序的测试覆盖率及漏洞挖掘能力, 该组实验结果见 6.2.1 节描述; 另一组是对已知注入漏洞的 LAVA-M 测试程序进行默认调度和本文最优调度算法的比较, 验证基于变异策略的并行调度方法在并行模糊测试中的漏洞挖掘能力提升效果, 本组实验结果见 6.2.2 节描述。

6.2.1 真实应用程序测试结果

实验方法: 我们选取 libxml2 中的 xmllint, libtiff 中的 tiff2pdf, libarchive 中的 bsdtar, ImageMagick 中的 convert, OpenJPEG 中的 opj_decompress 五个目标程序进行并行调度实验验证。实验中调度算法分为 4 种, 分别是默认调度(Default)、单变异策略并行的调度(One-Mutator)、基于权重的多变异策略组合并行调度(Weighted-Mutator)、基于同步的多变异策略组合并行调度(Sync-Mutator)。其中 Default 使用默认的运行方式在每个实例上使用相同命令执行模糊测试; One-Mutator 在每个实例中运行不同的单一变异策略进行模糊测试; Weighted-Mutator 使用权重方式动态调度每个测试实例使用的变异策略组合, 增加优势策略的调度权重, 但并未使用实例间变异策略交叉组合应用; Sync-Mutator 在 Weighted-Mutator 的基础之上使用基于时间片的有效样本同步方式进行实例间变异策略组合应用。实验中的不同调度算法在相同运行环境下执行, 默认运行时间 24 小时。本文对测试执行过程中目标程序的测试覆盖率及发现异常数量等进行对比分析。本组实验中各个调度算法随时间变化呈现相同的增长趋势, 前期增长较快, 后期趋于平缓, 这是由模糊测试本身的特点决定的。

为了验证本文调度方法的应用效果, 以下将分别从并行测试的综合覆盖率、实例内的测试冗余率以及漏洞挖掘能力等方面进行分析。

测试覆盖率: 本节将使用实验数据说明本文提出的 Sync-Mutator, 相比于默认策略, 收敛后的覆盖率更高, 而且达到相同覆盖率所需时间更短。图 10~图 13 分别是针对 libtiff 库中的 tiff2pdf、ImageMagick 库中的 convert 和 OpenJPEG 库中的 opj_decompress、libxml2 库中的 xmllint 在使用 AFL 模糊测试器进行不同并行调度测试后数据。由图可以看出, 在 24 小时测试过程中默认调度算法的测试覆盖率增长缓慢、覆盖率低。One-Mutator 直接使用单一变异策略并行的调度算法覆盖率比默认调度算法 Default 高, Weighted-Mutator 基于权重的方法覆盖率略优于 One-Mutator 算法, Sync-Mutator 使用有效样本同步的并行调度的测试覆盖率最高。三组实验中的 Sync-Mutator 比 Default 的综合测试覆盖率分别提高 35%、132%、41%和 51.7%。

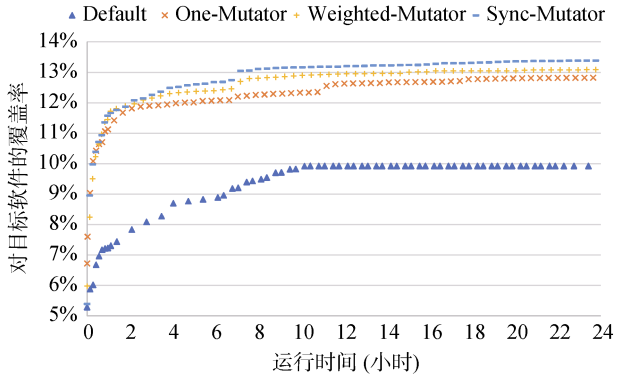


图 10 tiff2pdf 不同并行调度策略覆盖率对比

Figure 10 Coverage comparison of tiff2pdf between different parallel schedulers

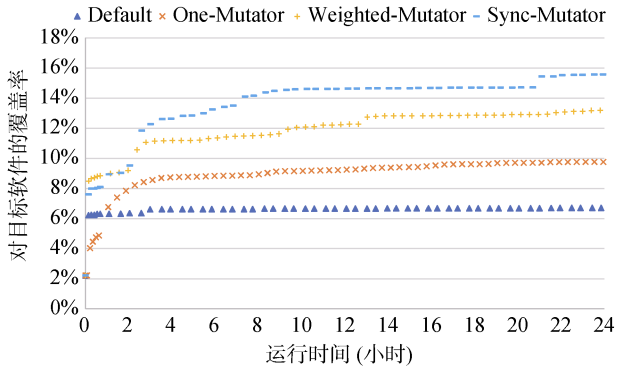


图 11 convert 不同并行调度策略覆盖率对比

Figure 11 Coverage comparison of convert between different parallel schedulers

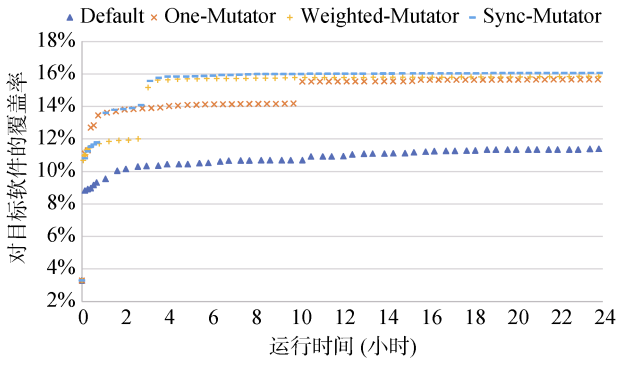


图 12 opj_decompress 不同并行调度策略覆盖率对比

Figure 12 Coverage comparison of opj_decompress between different parallel schedulers

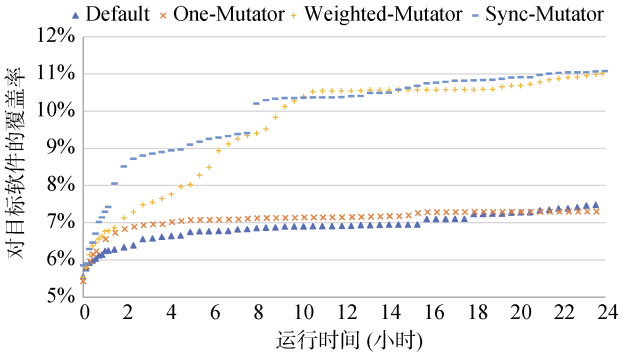


图 13 xmllint 不同并行调度策略覆盖率对比

Figure 13 Coverage comparison of xmllint between different parallel schedulers

图 14 是 libarchive 库中的 bsdtar 的不同并行调度算法测试结果对比, Sync-Mutator 调度算法相对于 Default 调度算法对 bsdtar 的测试覆盖率提高 45.1%。此外, 由于bsdtar中默认调度算法下的测试覆盖率收敛速度慢, 本文延长了 Default 调度的测试时间, 其在运行 40 小时后趋于收敛, 收敛后的覆盖率为 8.17%, 本文提出的Sync-Mutator在24小时时的收敛覆盖率仍比其高出 18.23%。可见, 在本文的并行调度方法收敛快于默认方法时, 收敛后的覆盖率仍然高于默认的调度方法。

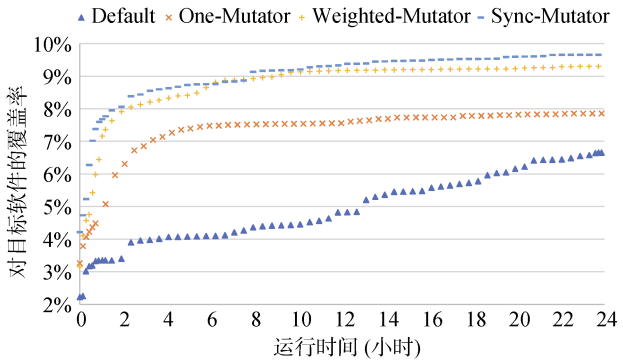


图 14 bsdtar 不同并行调度策略覆盖率对比

Figure 14 Coverage comparison of bsdtar between different parallel schedulers

实例内测试冗余: 通过上述测试覆盖率测试效果显示, 本文调度方法的并行模糊测试的综合覆盖能力大幅提升, 即降低了实例间的测试冗余率。此外, 在实例内测试冗余的衡量上, 我们定义单个实例内的有效测试用例比例(实例内测试冗余=有效测试用例数量/所有测试用例数量)来表示实例内部的冗余率, 有效测试用例比例越高, 表示实例内的冗余率越小。我们针对 5 款测试软件采用 Default 和 Sync-Mutator 调度方法的实例内平均有效测试用例比例来衡量。表 3 所示为不同目标软件实例内每 10000 个测试用例中的有效测试用例数量比较, 可以看出, 本文的 Sync-Mutator 平均有效测试用例比例均高于默认的调度方法, 且综合 5 个目标软件的平均有效测试用例比例提高 5 倍。

表 3 不同并行调度策略下实例内每 10000 个测试用例中平均的有效测试用例数量比较

Table 3 Comparison of effective samples in 10000 testcases between different parallel schedulers

目标程序	Default	Sync-Mutator
tiff2pdf	1.12	90.04
xmllint	0.58	17.99
bsdtar	0.27	3.89
convert	1.29	58.70
opj_decompress	43.89	110.39
平均	9.43	56.20

本文的方法在调度过程中每个测试实例内使用的变异策略更具有针对性, 可以提高不同变异策略的应用深度, 另外, 通过同步有效测试用例的方法达到交叉变异的效果, 可以提升实例内部的测试广度。因此, Sync-Mutator 提高了实例内有效测试用例的比例, 减少了实例内部的测试冗余。

漏洞挖掘能力: 表 4 所示是 tiff2pdf、convert、opj_decompress 程序在不同并行调度方法下进行并行化模糊测试过程中发现异常数目对比。可以看出, 对于三个目标程序来说, Sync-Mutator 并行调度算法发现的异常数目均明显高于其他三个调度算法, 使用异常调用栈自动化分析方法对异常样本进行去重后 Sync-Mutator 算法的不同异常数目最多。

表 4 不同并行调度运行异常数目

Table 4 Crashes with different parallel schedulers

目标程序	tiff2pdf		convert		opj_decompress	
并行调度算法	异常数目	不同异常数目	异常数目	不同异常数目	异常数目	不同异常数目
Default	29	**1**	0	**0**	0	**0**
One-Mutator	700	**3**	6	**1**	29	**1**
Weighted-Mutator	689	**3**	12	**1**	5	**1**
Sync-Mutator	678	**4**	13	**1**	32	**4**

综合上述 5 个真实应用程序的测试效果来看, 本文提出的基于变异策略的调度方法能够提高模糊测试运行过程中对目标程序的测试覆盖程度及漏洞挖掘效果, 结果表明结合变异策略的并行调度、优势变异策略增加使用权重及实例间有效样本同步的 Sync-Mutator 调度算法达到的测试效果最优。

6.2.2 LAVA-M 测试集测试结果

由于类似的并行模糊测试系统 ClusterFuzz 并没有开源代码或者数据可以对比, 本组采用本文的两种并行调度方法对 17 个测试实例并行测试比较。一组采用 AFL 默认并行(Default-AFL)的执行方法进行调度, 一组使用变异策略并行的调度方法(Sync-Mutator)进行调度。执行过程中每个测试程序的输入种子样本相同, 测试命令相同。其中 md5sum 使用“-c”参数, base64 使用“-d”参数运行, who 和 uniq 使用默认参数执行。每组测试运行时间是 24 小时, 测试完成后统计的每组测试发掘 LAVA-M 中的漏洞的分析结果如表 5 所示。

表 5 LAVA-M 测试集中发掘漏洞数量

Table 5 Detected bugs on LAVA-M dataset

程序	总计	Default-AFL	Sync-Mutator
base64	44	0	40
md5sum	57	1	5
uniq	28	0	5
who	2136	1	54
总计	2265	2	104

表 5 所示是两组不同调度方法在执行 24 小时在 4 个测试程序中发掘的注入漏洞的数量统计。第一列是对应目标程序的名称, 第二列是该目标程序中包含的所有漏洞的数量, 第三列是采用 17 个默认执行命令的实例并行测试发现漏洞数量, 第四列是采用本文的变异策略并行的调度方法发掘漏洞的数目。

通过表 5 可以看出, 本文采用的并行调度算法在 4 个测试程序中均发现了注入的漏洞, 其中发现 uniq 中漏洞 5 个、base64 中漏洞 40 个、md5sum 中漏洞 5 个、who 中漏洞 54 个, 对用的漏洞编号如表 6 所示。采用默认的调度算法只在其中两个测试程序中发现漏洞, 其中 md5sum 程序中漏洞 1 个、who 程序中漏洞 1 个。对于 LAVA-M 中的 4 个测试程序的漏洞挖掘效果都明显优于默认的调度方法。综合看来, 采用变异策略并行的调度算法在实际漏洞挖掘中能够明显提高模糊测试器的漏洞挖掘效率。

表 6 LAVA-M 测试集中发现漏洞的编号

Table 6 IDs of bugs detected on the LAVA-M dataset

程序	漏洞编号(Bugs ID)
uniq	321, 222, 346, 468, 472
base64	558, 790, 780, 573, 842, 276, 560, 835, 788, 1, 235, 778, 284, 805, 582, 782, 583, 584, 817, 556, 831, 562, 278, 832, 274, 521, 572, 566, 386, 786, 576, 804, 222, 253, 784, 841, 843, 255, 792, 774
md5sum	554, 270, 272, 1, 2
who	512, 1280, 4194, 1, 4240, 4354, 1282, 4327, 576, 4230, 562, 4342, 580, 531, 4096, 503, 4250, 4352, 522, 10, 577, 1290, 596, 548, 355, 336, 327, 559, 346, 4111, 4347, 558, 4186, 4115, 4252, 535, 454, 3835, 3850, 492, 4051, 1299, 4167, 4176, 4355, 334, 3918, 4052, 3782, 4110, 3863, 4163, 4094, 582

7 总结

随着模糊测试技术的不断发展以及云计算等技术的普及, 模糊测试的应用逐渐走向并行化、规模化。然而, 由于模糊测试的随机性、盲目性等特点, 导致缺少协同控制的并行模糊测试存在高冗余、低覆盖率等问题。本文提出一种变异策略感知的并行模糊测试方法, 在建立权重模型的基础上结合不同变异策略作用效果的差异指导不同测试实例进行变异策略选取, 并利用实例间同步方法实现变异策略与种子的交叉组合应用, 减少不同实例间测试的冗余性, 提高综合覆盖率。本文基于 VARAS 平台实现了一个并行模糊测试系统, 并在真实应用程序和 LAVA-M 测试集上对本文提出的调度方法进行了实验验证, 结果表明, 本文的并行模糊测试方法在测试目标覆盖率和漏洞挖掘效果两个方面均能够大幅提高模糊测试的应用效果。

我们将利用本文实现的并行模糊测试系统对模糊测试中的初始种子并行及同步方法等进行研究, 提高模糊测试在多实例并行模式下的测试效率。

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] Sutton. M., A. Greene, .P. Amini. Fuzzing: brute force vulnerability discovery[M]. Pearson Education, 2007.

[3] Li J, Zhao B D, Zhang C. Fuzzing: A Survey[J]. *Cybersecurity*, 2018, 1: 6.

[4] Lcamtuf. american fuzzy lop - Lcamtuf - coredump.cx. 2013; Available from: http://lcamtuf.coredump.cx/afl/.

[5] Woo M, Cha S K, Gottlieb S, et al. Scheduling Black-box Muta-

tional Fuzzing[C]. *the 2013 ACM SIGSAC conference on Computer & communications security*, 2013: 265-261.

[6] Rebert. A., S.K. Cha, T. Avgerinos, et al. Optimizing seed selection for fuzzing[C]. *USENIX*, 2014:124-131.

[7] Böhme M, Pham V T, Roychoudhury A. Coverage-based Greybox Fuzzing as Markov Chain[C]. *the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 124-131.

[8] Swiecki, R. Honggfuzz. 2016; Available from: https://github.com/google/honggfuzz.

[9] Böhme M, Pham V T, Nguyen M D, et al. Directed Greybox Fuzzing[C]. *the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 12-17.

[10] Li Y K, Chen B H, Chandramohan M, et al. Steelix: Program-state Based Binary Fuzzing[C]. *the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017: 125-131.

[11] Petsios T, Zhao J, Keromytis A D, et al. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities[C]. *the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 26-32.

[12] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C]. Proceedings 2017 Network and Distributed System Security Symposium, 2017: 36-43.

[13] Schumilo. S., C. Aschermann, R. Gawlik, et al. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels[C]. *26th {USENIX} Security Symposium*. 2017: 36-41.

[14] Wang. J., B. Chen, L. Wei, et al. Skyfire: Data-driven seed generation for fuzzing[C]. *Security and Privacy (SP)*, 2017: 15-32.

[15] Gan. S., C. Zhang, X. Qin, et al. CollAFL: Path Sensitive Fuzzing. in Security and Privacy (SP)[C]. *2018 IEEE Symposium*. 2018:365-372.

[16] Serebryany, K. Libfuzzer: A library for coverage-guided fuzz testing (within llvm). Available from: https://llvm.org/docs/LibFuzzer.html.

[17] Chen. P., H. Chen. Angora: Efficient Fuzzing by Principled Search[C]. *Security and Privacy (SP)*, 2018:258-261.

[18] OSS-Fuzz - continuous fuzzing of open source software. Available from: https://github.com/google/oss-fuzz.

[19] Godefroid. P., M.Y. Levin, D.A. Molnar. Automated whitebox fuzz testing. *NDSS*. 2008:258-261.

[20] Bucur S, Ureche V, Zamfir C, et al. Parallel Symbolic Execution for Automated Real-world Software Testing[C]. Proceedings of the sixth conference on Computer systems – EuroSys, 11, 2011: 654-662.

[21] Bounimova. E., P. Godefroid, D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production[C]. *the 2013 International Conference on Software Engineering*. 2013: 45-51.

[22] Linn, A. Microsoft previews Project Springfield, a cloud-based bug detector. 2016; Available from: https://blogs.microsoft.com/ai/ microsoft-previews-project-springfield-cloud-based-bug-detector/.

[23] Xu W, Kashyap S, Min C, et al. Designing New Operating Primitives to Improve Fuzzing Performance[C]. *the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 236-241.

[24] Arya. A., C. Neckar, C.S. Team. Fuzzing for Security[EB/OL]. https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[25] Taof-The art of fuzzing. Available from: https://sourceforge.net/projects/taof/.

[26] Hocevar, S., zzuf—multi-purpose fuzzer. 2011.

[27] Helin, A. Radamsa fuzzer. 2013; Available from: https://github.com/aoh/radamsa.

[28] Amini, P. Sulley: Pure Python fully automated and unattended fuzzing framework[EB/OL]. 2010; code.google.com/p/sulley.

[29] Godefroid. P., H. Peleg, .R. Singh. Learn&fuzz: Machine learning for input fuzzing[C]. *the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 2017: 269-271.

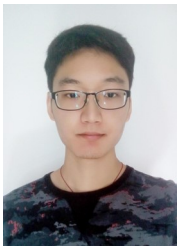
[30] Rajpal, Mohit, Blum, et al. Not all bytes are equal: Neural byte sieve for fuzzing[EB/OL]. arXiv:1711.04596.

[31] Dolan-Gavitt. B., P. Hulin, E. Kirda, et al. Lava: Large-scale automated vulnerability addition[C]. *Security and Privacy (SP)*, 2016:369-372.

[32] OpenStack is Open source software for creating private and public clouds.; Available from: https://www.openstack.org/.

邹燕燕 于 2014 年在中国科学技术大学计算机系统结构专业获得硕士学位。现在中国科学院大学网络空间安全专业攻读博士学位。研究领域为软件安全分析。研究兴趣包括：漏洞挖掘、模糊测试、程序分析。
Email: zouyanyan@iie.ac.cn

邹维 于 1988 年在中国科学院计算技术研究所计算机软件专业获得硕士学位。现任中国科学院信息工程研究所副所长。研究领域为包括软件安全分析理论与技术等。研究兴趣包括: 规模化软件漏洞挖掘技术。
Email: zouwei@iie.ac.cn



尹嘉伟 于 2017 年在吉林大学软件工程专业获得学士学位。现在中国科学院大学网络空间安全专业攻读硕士学位。研究领域为系统安全。研究兴趣包括: 系统结构安全。Email: yinjiawei@iie.ac.cn

霍玮 于2010年在中国科学院计算技术研究所获得博士学位。现任中国科学院信息工程研究所副研究员。主要研究方向为软件漏洞挖掘和安全评测、基于大数据的软件安全分析、智能终端系统及应用安全分析等。Email: huowei@iie. ac.cn

杨梅芳 于2018年在中国科学院大学信息安全专业获得硕士学位。现任中国科学院信息工程研究所研究实习员。研究领域为软件安全。研究兴趣包括: 二进制漏洞挖掘。Email: yangmeifang@iie.ac.cn

孙丹丹 于 2013 年在上海大学计算机软件与理论专业获得硕士学位。现任中国科学院信息工程研究所工程师。研究领域为云计算、软件安全。研究兴趣包括: 云计算安全、软件安全、漏洞挖掘。Email: sundandan@iie.ac.cn

史记 于 2016 年在中国人民公安大学公安技术专业获得硕士学位。现在中国科学院大学网络空间安全专业攻读博士学位。研究领域为软件安全测评、软件漏洞挖掘。研究兴趣包括: 软件模糊测试。Email: shiji@iie.ac.cn