

基于分布散度的自适应模糊测试优化方法

许航¹, 计江安¹, 马哲宇², 张超²

(1. 网络空间安全教育部重点实验室, 河南 郑州 450001; 2. 清华大学网络科学与网络空间研究院, 北京 100080)

摘要: 为了提高覆盖引导的模糊测试的性能, 提出一种利用分布散度和深度强化学习模型自适应模糊测试的优化方法。基于过程间控制流图构建过程间比较流图, 以刻画被测程序分支判断变量对应的空间随机场, 并使用蒙特卡洛方法提取模糊测试变异策略产生的随机场分布特征; 构建深度图卷积神经网络, 以提取过程间比较流图的特征嵌入, 并将该神经网络作为深度强化学习的深度Q网络; 基于双重深度Q网络模型建立在线深度强化学习模型, 从而训练智能代理以优化模糊测试变异策略。该深度强化学习模型利用种子文件及相关区块对应的随机场分布特征定义状态, 将种子文件重点变异区块的选择定义为动作, 并将动作前后随机场近似分布的分布散度定义为奖励。针对该模糊测试优化方法实现了原型系统, 并对其开展了多轮次的持续24 h的评估。实验结果表明, 在测试集FuzzBench上, 该原型系统代码覆盖速度和总体覆盖明显优于基线模糊测试器AFL++与Havoc_{MAB}, 并在多数基准测试目标上优于CmpLog; 在测试集Magma上, 该原型系统在基准测试目标openssl、libxml和sqlite3上具备更强的漏洞触发能力。

关键词: 模糊测试; 深度强化学习; 分布散度; 分支判断变量

中图分类号: TP311.5

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2024079

Self-adaptive fuzzing optimization method based on distribution divergence

XU Hang¹, JI Jiangan¹, MA Zheyu², ZHANG Chao²

1. Key Laboratory of Cyberspace Security, Ministry of Education, Zhengzhou 450001, China

2. Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100080, China

Abstract: To improve the performance of coverage-guided fuzzing, a method for self-adaptive optimization of fuzzing using distribution divergence and a deep reinforcement learning model was proposed. An interprocedural comparison flow graph was first constructed based on the interprocedural control flow graph to characterize the spatial random field corresponding to the branch condition variables of the program under test, and the distribution features of the random field generated by a fuzzing mutation strategy were extracted using the Monte Carlo method.

收稿日期: 2024-07-11; 修回日期: 2024-11-21

通信作者: 张超, chaoz@tsinghua.edu.cn

基金项目: 国家重点研发计划 (2021YFB2701000)

Foundation Item: The National Key Research and Development Program of China (2021YFB2701000)

引用格式: 许航, 计江安, 马哲宇, 等. 基于分布散度的自适应模糊测试优化方法[J]. 网络与信息安全学报, 2024, 10(6): 37-58.

Citation Format: XU H, JI J A, MA Z Y, et al. Self-adaptive fuzzing optimization method based on distribution divergence[J]. Chinese Journal of Network and Information Security, 2024, 10(6): 37-58.

Then, a deep graph convolutional neural network was constructed to extract the feature embeddings of the interprocedural comparison flow graph, and this neural network was used as the deep Q-network for deep reinforcement learning. Finally, an online deep reinforcement learning model was established based on the dual deep Q-network model, and an intelligent agent was trained to optimize the fuzzing mutation strategy. This deep reinforcement learning model defined its state using the random field distribution features corresponding to the seed file and the associated blocks. The selection for the focused mutation block of a seed file was defined as an action, and the distribution divergence of the approximate distributions of the random fields before and after the action was defined as the reward. A prototype was implemented for this fuzzing optimization method, and multiple rounds of up to 24 hours of evaluation were carried out on this prototype. The experimental results show that on the benchmark Fuzz-Bench, the code coverage speed and overall coverage achieved by the prototype are significantly better than those of the baseline fuzzer AFL++ and HavocMAB, and better results are achieved on most benchmarks compared to CmpLog. On the benchmark Magma, stronger vulnerability triggering capability is demonstrated by the prototype on the benchmarks openssl, libxml, and sqlite3.

Keywords: fuzzing, deep reinforcement learning, distribution divergence, branch condition variables

0 引言

模糊测试被广泛应用于软件漏洞检测, 其利用动态生成的测试用例探索被测程序的输入空间, 以触发程序异常行为或崩溃。然而, 被测程序的输入空间非常广阔, 需要缩小输入空间以提高测试效率。目前, 覆盖引导的模糊测试 (coverage guided fuzzing, CGF) 是成熟的优化方案之一, 其根据代码覆盖的相关反馈选择种子, 并使用预定义的变异算子, 在特定变异策略 (如 Havoc) 的指导下生成更多测试用例。

为了提高覆盖引导的模糊测试的性能, 研究人员在种子调度和种子变异两个方面进行了一系列探索。前者研究如何保留有可能提高覆盖率的测试用例作为种子, 后者则研究如何基于特定种子进行高效变异。

在种子调度方面, AFLFast^[1]、VUzzer^[2]、CollAFL^[3]、FairFuzz^[4]和 EcoFuzz^[5]等研究使用稀有路径、相邻未覆盖边和稀有边等控制流反馈指标来挑选有望推进模糊测试进度的种子, 但没有利用数据流反馈。DDFuzz^[6]虽然能够基于数据流图边的覆盖情况挑选种子以在数据流方面引导模糊测试, 但是其收集的反馈粒度较粗, 无法捕捉到细粒度的数据流反馈 (如变量的值和分布), 而这些反馈可能有利于制定更高效的模糊测试策略。

在种子变异方面, 一个关键的挑战是自动化地构建模糊测试输入空间到符合被测程序格式要

求的样本空间的映射。目前, 一些方法使用程序分支判断变量的单个运行时的值来建立这种映射。REDQUEEN^[7]利用比较指令日志 (CMPLog) 识别输入中的幻数, 但无法处理复杂的数据流反馈。WEIZZ^[8]是一种面向二进制形式输入的被测程序的模糊测试方法, 其根据比较指令日志自动生成和修改测试用例。但是, 现有的方法主要是在输入和比较指令之间建立基本的关联关系。这些方法并没有利用细粒度数据流反馈, 且没有提出鲁棒的模型, 以构建模糊测试策略及其对被测程序影响之间的映射。此外, 使用人工定义的模型建立模糊测试反馈到模糊测试策略之间的映射受到特征维度和模型复杂度的限制, 从而导致这些模型的鲁棒性和适应性不如智能模型。

本文提出了一种基于深度强化学习 (deep reinforcement learning, DRL) 的解决方案, 利用分支判断变量分布特征动态调整模糊测试变异策略。为了实现该方案, 以下关键挑战需要被克服。

第一, 程序通常包含大量具有空间相关性的分支判断变量, 描述它们的整体概率分布特征具有挑战性。为了应对这一挑战, 本文引入了过程间比较流图 (interprocedural cmp flow graph, ICMPG) 以描述被测程序分支判断变量的空间分布特征。此外, 本文还利用蒙特卡洛方法获取分支判断变量的近似空间分布。

第二, 传统模糊测试方法无法根据变异策略对被测程序数据流分布的影响自适应地调整变异

策略。因此，本文设计了一个专用的神经网络模型以分析分支判断变量的空间分布特征。该神经网络模型可以预测不同变异策略在提高分支判断变量分布的分布散度的效果，并从一组预定义的子策略中自适应地选择潜力最大的子策略。

第三，平衡人工智能模型训练开销和模糊测试吞吐量是一项挑战。对此，本文应用深度强化学习模型在线选择最有效的种子变异策略并动态控制开销。本文对各个变异策略影响下的分支判断变量空间分布特征进行概率采样，以确定深度强化学习模型的状态。该方法避免了对深度学习模型进行预训练的额外开销，并克服了预训练模型泛化能力差的限制。

本文主要的贡献如下。

1) 提出了一种基于深度强化学习调整模糊测试策略的新方法，可以分析分支判断变量分布特征并动态调整模糊测试变异策略。

2) 实现了原型系统，并通过实验证明了在线深度强化学习模型在改善分支判断变量分布的分布散度方面的有效性。

3) 利用测试集 FuzzBench^[9]评估了原型系统在对 24 个真实世界测试目标的覆盖性能。实验结果表明，与基线方法 AFL++ 相比，在 24 h 的模糊测试中，原型系统的分支覆盖在平均速度上提高了 2 倍以上，并明显优于相关工作 Havoc_{MAB} 和 CmpLog。

4) 使用测试集 Magma^[10]中的 9 个真实世界目标对原型系统进行了评估。实验结果表明，相较于 AFL++、Havoc_{MAB} 和 CmpLog，本文原型系统在一些测试目标上具有更强的漏洞触发能力。

1 研究背景

本文的研究内容是深度强化学习辅助的模糊测试变异策略自适应调整方法，具体使用了图神经网络 (graph neural network, GNN) 作为深度强化学习的神经网络，因此对覆盖引导的模糊测试、图神经网络和深度强化学习的研究背景进行简要说明。

1.1 覆盖引导的模糊测试

覆盖引导的模糊测试收集被测程序在运行时与覆盖相关的信息作为反馈，以评估模糊测试的测试用例对被测程序的影响。根据该影响效果，

覆盖引导的模糊测试评估模糊测试策略 (包括种子调度策略、种子变异策略等) 的效果，并动态调整策略，以更快地探索被测程序的代码空间。

为了提高评估的全面性，可以使用更多类型的反馈，包括控制流反馈 (如代码覆盖) 和数据流反馈 (如变量运行时值)。同时，为了提高评估的准确性，在反馈和影响效果之间建立更复杂的映射关系也是有益的。目前，研究人员使用了多臂老虎机模型^[11]、马尔可夫链模型^[1]、基于图理论的模型^[12]和基于熵理论的模型^[13]构建这种映射关系。与简单模型相比，这些模型可以提供更鲁棒的映射，从而帮助模糊测试更准确地评估变异策略或种子调度策略的效果。

通过使用对被测程序适合度更高的变异算子或更高效的变异策略，模糊测试的效率亦可以被提高。在变异算子方面，除使用人工定义的变异算子 (例如 AFL^[14]中使用的比特翻转、算术运算和随机替换等) 外，基于人工智能的变异算子也是目前流行的解决方案 (例如使用深度学习训练生成符合特定输入格式的神经网络)。基于人工智能的方法也被用于变异策略自适应方面的工作。在此方面，强化学习为变异策略自适应调整提供了一条前景广阔的研究方向，其无须大量的训练数据集，且具备适应不同模糊测试环境的潜力。

1.2 图神经网络

图神经网络^[15]是一种以图类型数据作为输入的神经网络，通常被用于提取图级别、节点级别或边级别的特征嵌入，从而应对不同的任务。根据任务需求，可以为图的节点和边添加特征，并且可以基于频谱方法、空间方法或注意力机制定义消息传递过程。通常，在第 l 层使用 $M^{(l)}$ 作为消息传递函数的图神经网络消息传递过程可以用式(1)表示。

$$h_i^{(l+1)} = \sum_{j \in A_i} M^{(l)}(h_i^{(l)}, h_j^{(l)}, e_{ij}) \quad (1)$$

其中， $h_i^{(l)}$ 是图神经网络第 l 层输入的图的第 i 个节点的嵌入， A_i 是第 i 个节点的相邻节点索引的集合， e_{ij} 是第 i 个节点和第 j 个节点之间的边。

此外，与卷积神经网络 (convolutional neural network, CNN) 不同，图神经网络通常单独定义池化层。池化层被用于去除不被关注的节点

及边,以降低神经网络的计算复杂度并提高鲁棒性。对于图分类或图回归等图级别的任务,图神经网络利用读出层得出整个图的特征嵌入。该嵌入被传递至输出层,并生成整个神经网络最终的输出嵌入。

由于程序的控制流关系和数据流关系都可以使用图数据结构来表征,因此使用图神经网络提取程序(或程序行为)的特征是当前热门的研究方向。为了使用图神经网络提取程序(或程序行为)的特征嵌入,必须定义图的拓扑结构以及图节点或边的特征。

1.3 深度强化学习

强化学习(reinforcement learning, RL)通过与环境交互训练智能代理,并通过该代理做出决策以优化任务的目标奖励。根据学习方法的不同,强化学习可分为离线式和在线式。前者使用预先准备好的训练数据进行训练并可用于非实时任务,后者使用实时数据进行训练并可用于实时任务^[16]。离线式强化学习适用于具有丰富数据的确定环境,其训练成本在决策时可以被忽略。对于类型不确定(或差异较大)的环境,使用在线强化学习可以训练适应不同环境的智能代理,从而克服固定参数模型泛化性差的缺点。此外,根据学习和使用过程中是否使用相同的策略,强化学习被分为同策略式(on-policy)和异策略式

(off-policy)。这两种类型中,异策略式强化学习方法具有更强的鲁棒性和通用性。

深度强化学习采用深度神经网络(deep neural network, DNN)拟合强化学习中的值或策略函数,以增强传统强化学习的学习能力和泛化能力。深度学习的目的是学习一个能根据数据特征做出预测或决策的神经网络模型,而深度强化学习的目的则是训练一个能根据环境状态做出决策以尽可能获得最大奖励的智能代理。Mnih等^[17]提出的双重深度Q网络(deep Q-network, DQN)是一种典型的在线异策略式深度强化学习方法,其在DQN^[18]的基础上进行了改进,额外使用了目标Q网络来计算在某个状态实施各个动作的动作价值,缓解了传统DQN使用相同Q网络判断策略和评估目标Q值引起的不稳定问题。

2 方法

图1展示了本文提出的基于分布散度的自适应模糊测试优化方法的概览,主要分为空间随机场生成、神经网络模型和深度强化学习模型3个部分。

首先,本方法根据被测程序的控制流图和数据流图构建过程间比较流程图(ICMPG),并利用蒙特卡洛方法估计受模糊测试策略影响的分支判断变量的概率分布,从而构建空间随机

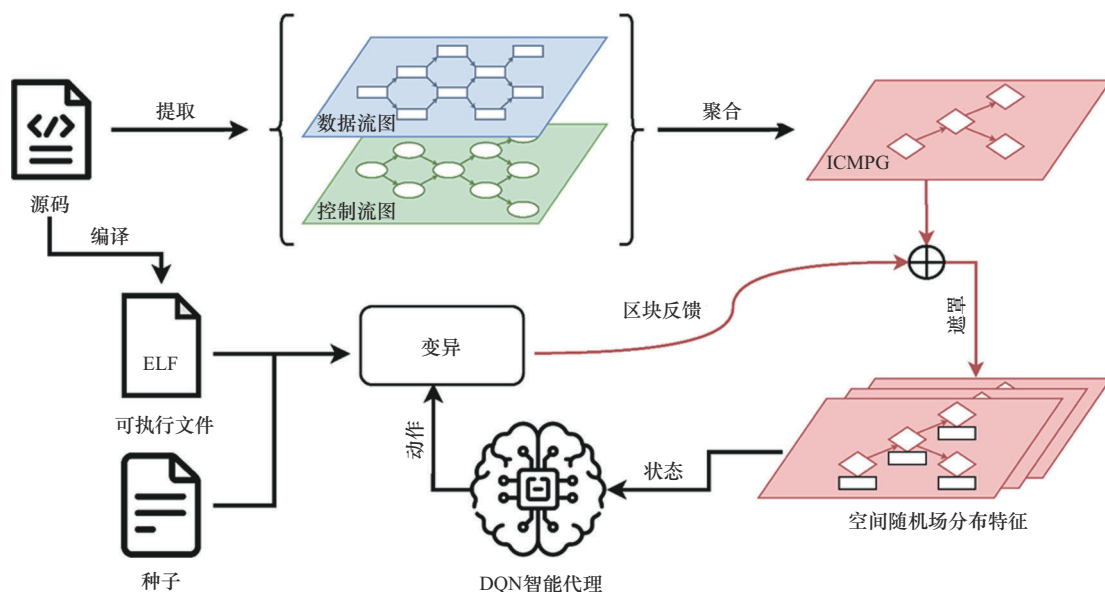


图1 基于分布散度的自适应模糊测试优化方法的概览

Figure 1 Overview of self-adaptive fuzzing optimization method based on distribution divergence

场中随机变量的近似概率分布特征。具体地，本方法将一个种子文件分成多个区块，并设置一个有偏的概率分布以选择各个区块进行变异。在采样一定时间后，本方法为每个区块和整个种子文件生成在给定变异策略（有偏概率分布）影响下随机场的近似分布特征。

随后，空间随机场的概率分布特征被输入深度强化学习的深度神经网络，以计算给定状态下备选动作的预期价值（Q值）。该神经网络是一个伪孪生神经网络，主要由两个图神经网络组成，目的是提取图级别的特征嵌入。

最后，本方法将模糊测试种子变异建模为强化学习问题，并利用深度强化学习模型在线训练智能代理指导模糊测试的变异。

2.1 空间随机场分布

空间随机场描述了一组空间相关随机变量的特征。其通常包括位置、数量和值等各种属性，并提供了这些属性在空间环境中如何变化的全面视图。本文使用这一概念的目的是在对被测程序的模糊测试种子应用变异策略后，对多个空间相关分支判断变量值的分布特征进行统计分析。使用随机场分布特征描述在拓扑空间中关联的随机变量，能够更全面和细粒度地刻画模糊测试策略作用于被测程序时产生的数据流特征。

2.1.1 过程间比较流图

为了定义一个随机场，必须定义随机场对应的拓扑空间。本文利用分支判断变量的上下文相关性描述空间相关性特征。本节介绍构建该拓扑空间（过程间比较流图）的方法。通常，分支判断变量在比较指令处被使用，并且程序依照比较指令的结果决定条件跳转的目的。因此，本文利用比较指令所在基本块的上下文关系定义随机场的拓扑。为了确保拓扑空间的每一个位置都可以索引到一个随机变量，提出以条件跳转指令所在基本块作为节点的过程间比较流图。

本文依据被测程序的过程间控制流图构建过程间比较流图。过程间比较流图是一个有向图，其节点表示具有比较指令的基本块，边表示两个节点间的上下文关系（前驱或后继关系）。图2展示了基于本文提出的方法使用AFL++^[19]提供的test-instr.c程序生成过程间比较流图的示例。

首先，本方法为源代码所有函数生成中间语言（intermediate representation, IR）代码。同时，基本块被按照中间语言指令类型分类，按照优先级依次分为条件分支基本块 BB_{cmp} 、多分支选择基本块 BB_{switch} 、函数入口基本块 BB_{entry} 、函数出口基本块 BB_{ret} 和其他类型基本块 BB_{other} 。在图2所示的过程间控制流图中，这些类型的节点分别标记为绿色（ BB_{cmp} ）、橙色（ BB_{switch} ）、蓝

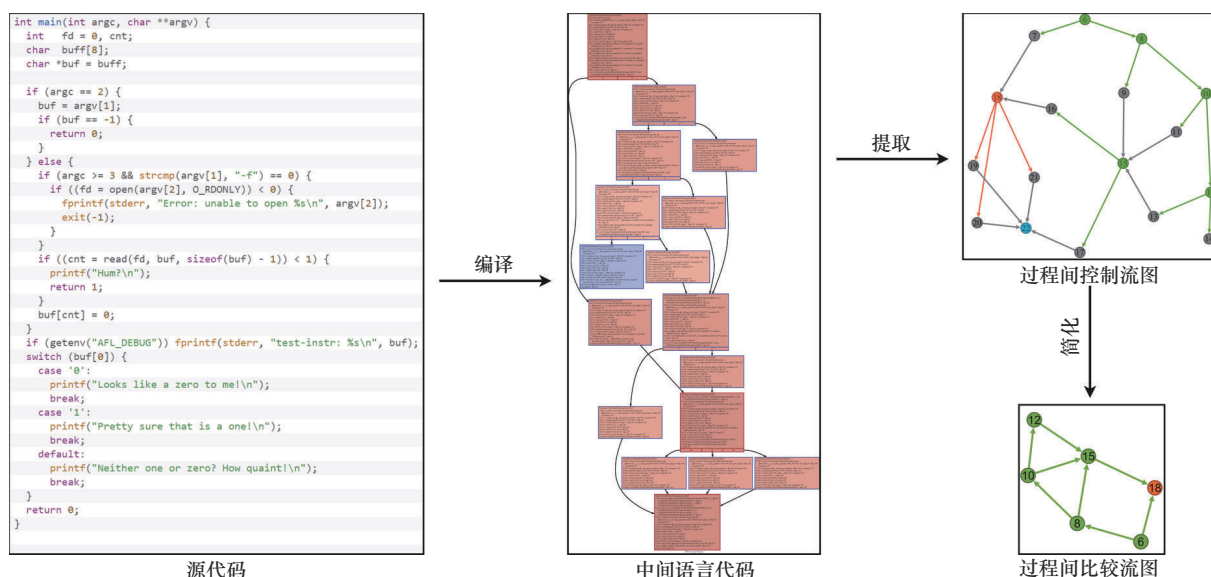


图2 使用AFL++提供的test-instr.c程序生成过程间比较流图的示例

Figure 2 An example of generating ICMPG using test-instr.c provided by AFL++

色 (BB_{ret}) 和灰色 (BB_{other})。在该示例中, 由于 BB_{entry} 类型节点的优先级低于 BB_{cmp} , 因此未被标注。

其次, 本方法根据函数调用图 (call graph, CG) 将所有函数的控制流图合并为表示整个被测程序控制流关系的过程间控制流图 (interprocedural control flow graph, ICFG)。

最后, 为了确保只有包含条件跳转类型的基本块作为节点保留在图中, 本方法将过程间控制流图简化为过程间比较流图, 具体过程见算法 1。

算法 1 简化过程间控制流图为过程间比较流图的算法

输入 $G(V, E)$ //过程间控制流图
输出 $G_{ICMPG}(V, E)$ //过程间比较流图

- 1) $V_{del} \leftarrow \{v \mid v \in V \wedge \text{type}(v) \neq BB_{cmp} \wedge \text{type}(v) \neq BB_{switch}\}$
- 2) **for each** $v \in V_{del}$
- 3) $V_{pre}^v = \{u \mid u \in V \wedge e_{u,v} \in E\}$
- 4) $V_{suc}^v = \{u \mid u \in V \wedge e_{v,u} \in E\}$
- 5) **for each** $s \in V_{pre}^v$
- 6) **for each** $d \in V_{suc}^v$
- 7) **add** $e_{s,d}$ to E
- 8) **remove** $e_{v,d}$ from E
- 9) **end for**
- 10) **remove** $e_{s,v}$ from E
- 11) **end for**
- 12) **remove** v from V

13) **end for**

14) $G_{ICMPG}(V, E) \leftarrow G(V, E)$

通过简化, 本方法既保证了图节点的同构性, 也剔除了大量无关节点, 从而确保利用图神经网络提取空间随机场分布特征嵌入的可行性。随机场中各个随机变量的空间距离由过程间比较流图中节点的距离表示。需要注意的是, 单个随机变量的分布特征与其较近的节点有更强的相关性是本文的假设之一。此外, 本文不考虑程序中的间接调用或隐式依赖关系。

2.1.2 随机场构建及分布特征获取

首先, 介绍如何将被测程序受模糊测试变异策略影响下的分支判断变量取值建模为随机场。对于原始概率空间 $(\Omega, \mathcal{F}, P)$, Ω 为被测程序的随机输入空间, 事件空间 $\mathcal{F}$ 是随机输入 Ω 的所有子集, 概率测度 P 为事件空间中的每个事件分配一个概率 (值域范围为 $[0, 1]$)。本文将 G_{ICMPG} 中第 i 个节点对应的分支判断变量的值定义为随机变量 $X_i: \Omega \rightarrow \mathbb{R}$, 其所在的概率空间是诱导概率空间 $(\mathbb{R}, \mathcal{F}', P')$ 。在这个诱导概率空间中, $\mathcal{F}'$ 表示 $\mathbb{R}$ 的所有子集。如图 3 所示, 对于一个种子文件, 变异本质上是对种子文件中部分字节的随机替换, 故使用一个变异策略作用于模糊测试, 并用变异生成的测试用例运行被测程序是对输入空间 Ω 的有偏差采样。该有偏采样也会引起诱导概率空间 $(\mathbb{R}, \mathcal{F}', P')$ 的概率测度 P' 发生变化。本文的目的是尽可能在模糊测试中最大化这种变化, 使得模糊测试能够探索被测程序更多样的输

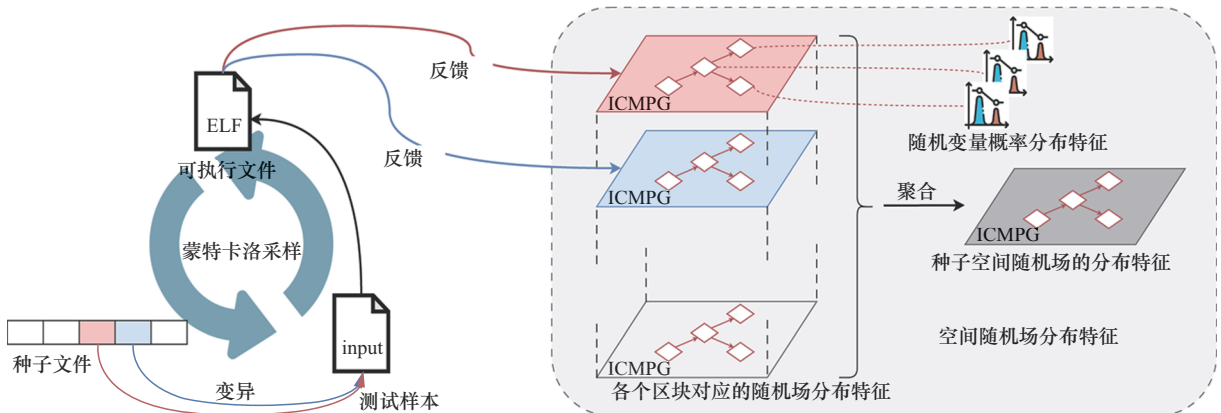


图 3 使用蒙特卡洛方法获取变异策略作用于被测程序后的随机场近似概率分布特征

Figure 3 Using the Monte Carlo method to obtain the approximate probability distribution features of the random field after a mutation strategy is applied to the PUT

入空间。按照随机场的形式化定义，随机场是由多个在某个拓扑空间中能够被唯一索引的随机变量所构成的集合。因此，必须要确保拓扑空间中的位置能够索引到唯一的随机变量。继而，本文将随机场定义为 $\{X_i; v_i \in G_{ICMPG}\}$ 。其中， G_{ICMPG} 是随机场的拓扑空间， X_i 是与 G_{ICMPG} 中节点一一对应的随机变量，表示与 G_{ICMPG} 中第 i 个节点 v_i 相对应的分支判断变量经过映射函数 F_i 映射得到的值。其中， F_i 将比较指令的左右操作数映射至实数空间。

然后，介绍如何获取随机场的分布特征。本方法优化的手段使得种子特定区块内的字节发生变异的概率更高，因此需要在实施变异策略后，通过随机场分布特征判断随机场分布的差异。为此，本方法选取随机场中随机变量的近似概率分布密度作为随机场分布特征，并通过蒙特卡洛采样获得随机场中每个随机变量 X_i 的近似概率分布。本方法将随机场中第 i 个随机变量的采样值记录为一个集合 $\{x_i^k\}$ （ x_i^k 是 X_i 的第 k 个采样值），并同时记录采样值的最大值 $\max_k x_i^k$ 和最小值 $\min_k x_i^k$ 。本方法设定了区间总数 $N_{interval}$ ，并统计随机变量在每个区间的概率，即随机变量 X_i 的近似概率分布 $\hat{d}(x_i)$ ，从而得到随机场的近似概率分布 $\hat{D} = \{\hat{d}(x_i)\}$ 。对于给定的变异策略，本方法以一定的概率对被测程序的每个分支判断变量值进行采样。为了同时保证随机变量近似概率分布的准确度和模糊测试的吞吐量，本方法将采样

开销动态控制在10%以内。

2.2 神经网络模型

如图4所示，在通过蒙特卡洛模拟得到变异策略影响下种子及其区块对应的近似随机场分布特征后，本文使用由伪孪生图神经网络和多层感知机组成的神经网络模型提取它们各自的嵌入。神经网络的输入是种子和各个区块的近似空间随机场分布特征图，种子和区块对应的特征图被分别输入两个结构相同但参数不共享的图神经网络（具体的，SAGPool^[20]）。图神经网络的输出是各个区块对应的空间随机场分布特征的嵌入，种子文件不同区块对应的特征嵌入最终被输入MLP中，并在输出层被激活函数（具体的，softmax）预测各个区块的分数，即各个动作（重点选择各个区块进行变异）的预测价值。

2.2.1 神经网络构建

本方法使用两个参数独立但结构相同的图卷积网络 GCN_{seed} 和 GCN_{block} 构建伪孪生图神经网络。伪孪生神经网络常用于提取两个输入数据之间的相似度特征，且由于两个神经网络的参数可以根据输入特征独立优化，因此伪孪生神经网络通常比共享权重的孪生网络具备更强的学习能力和鲁棒性。

本方法使用随机场中每个随机变量的近似概率分布特征 $\hat{d}(x_i)$ 作为图中节点的特征，并通过图卷积神经网络的多个“卷积-读出层”和读出函数获得图的嵌入。在该图卷积神经网络中，第 l 层“卷积-读出层”的卷积过程可表示为

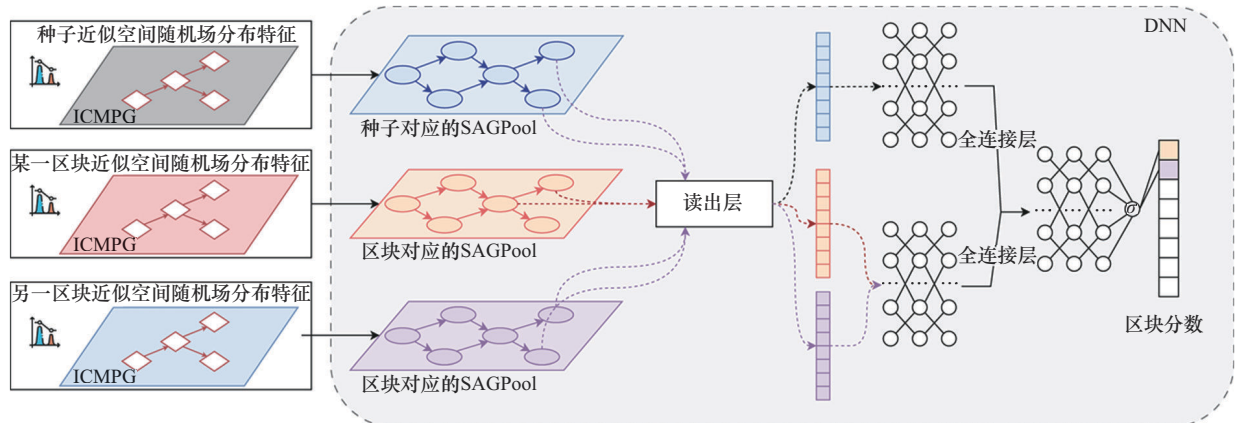


图4 用于提取近似随机场分布特征嵌入的伪孪生神经网络模型

Figure 4 The pseudo-siamese neural network model used to extract embeddings of approximate random field distribution features

$$h_i^{(l+1)} = \sigma(b^{(l)} + \sum_{j \in \mathcal{N}(i)} e_{ji} h_j^{(l)} W^{(l)}) \quad (2)$$

其中, σ 是激活函数 (relu); $b^{(l)}$ 是该层卷积中可学习的偏置张量; $W^{(l)}$ 是该层卷积中可学习的权重张量; h_j 是第 i 个节点的邻居节点 $\mathcal{N}(i)$ 中第 j 个节点的节点特征。需要说明的是, 本方法通过蒙特卡洛模拟, 删除 G_{ICMPG} 中不具备有效概率分布的随机变量及其相应节点。因此, 本方法引入了边权重 $e_{ji} = 1/s_{ji}$ (其中 s_{ji} 是第 j 个节点到第 i 个节点的最短路径长度), 从而避免了重复计算图邻接矩阵的开销。本方法使用平均和最大读出函数获得图嵌入, 第 l 层卷积层之后获得读出结果 $r^{(l)}$ 的过程可表示为

$$r^{(l)} = \left\{ \max_{i=1}^N (h_i^{(l+1)}[j]) \right\} \oplus \left\{ \frac{1}{N} \sum_{k=1}^N h_i^{(l+1)}[k] \right\}, 1 \leq j \leq \dim(h^{(l+1)})[0] \quad (3)$$

其中, N 是图中的节点数, j 是维度索引, $\dim()$ 表示张量的维度。然后, 本方法将每个“卷积-读出层”的结果连接, 得到图的嵌入 $r = r^{(1)} \oplus r^{(2)} \oplus \dots \oplus r^{(L)}$, 其中 L 是“卷积-读出层”的数量。

此外, 对于种子的每个区块, 本方法使用 $\text{GCN}_{\text{block}}$ 提取其图嵌入 r_{block} , 并将其与使用 GCN_{seed} 提取的种子对应的图嵌入 r_{seed} 连接。

2.2.2 神经网络回归

通过蒙特卡洛模拟和伪孪生图神经网络, 本方法得到了种子对应的随机场分布特征嵌入和每个种子文件区块对应的随机场特征嵌入。为了使神经网络能够预测单个区块的变异对随机场分布特征的影响, 本方法使用 MLP 和激活函数 softmax 获得对每个区块进行变异的预测价值 (第 i 个区块对应 $\hat{y}_i$)。数值最大的区块被作为主要的区块进行变异, 且该区块被选中的概率被设定为其他区块的 $\mathcal{K}$ 倍 ($\mathcal{K}$ 是神经网络的一个超参数)。然后, 该策略被用于指导模糊测试对种子进行变异, 并使用预测值 $\hat{y}_i$ 和目标值 y_i 计算损失。该损失将被用于对神经网络进行反向传播, 以调整网络参数。本方法使用该神经网络作为深度强化学习的深度神经网络 (DQN 的 Q 网络), 为模糊测试过程中的每个种子提供变异策略指导。

2.3 深度强化学习模型

本节介绍本文如何将模糊测试建模为强化学习问题以及本文使用的在线式强化学习方法。

2.3.1 强化学习模型

如图 5 所示, 强化学习基本模型包含环境、智能代理、状态、动作、奖励这 5 个关键要素。下面围绕这 5 个关键要素说明本方法如何将模糊测试建模为强化学习模型。



图 5 强化学习基本模型

Figure 5 Basic model of reinforcement learning

1) 环境。环境向智能代理提供状态, 从智能代理接收动作, 然后向智能代理提供在某个状态下执行某个动作的奖励。在本文的方法中, 模糊测试过程为环境 $\mathcal{E}$, 其为智能代理 $\mathcal{A}$ 提供模糊测试状态 $\mathcal{S}$, 部署智能代理 $\mathcal{A}$ 提供的动作 a 作为模糊测试的变异策略, 并根据模糊测试的反馈信息向智能代理 $\mathcal{A}$ 提供奖励 r 。

2) 智能代理。智能代理 $\mathcal{A}$ 是一个智能实体, 其从环境 $\mathcal{E}$ 接收状态 $\mathcal{S}$, 根据其策略为环境 $\mathcal{E}$ 提供建议的动作 a , 并根据环境 $\mathcal{E}$ 提供的奖励 r 改进其策略, 从而自适应地进化。

3) 状态。本方法将状态 $\mathcal{S}$ 定义为 $\mathcal{S} = \{\hat{D}_{\text{seed}}, \hat{D}_{\text{block}}^1, \hat{D}_{\text{block}}^2, \dots, \hat{D}_{\text{block}}^{N_{\text{block}}}\}$, 其中 $\hat{D}_{\text{seed}}$ 是种子对应的随机场近似概率分布, $\hat{D}_{\text{block}}^i$ 是第 i 个区块对应的随机场近似概率分布。环境 $\mathcal{E}$ 根据种子文件中每个区块变异产生的反馈计算随机场的近似概率分布, 并将这些区块的近似概率分布汇总为整个种子的突变所产生的近似概率分布。

4) 动作。本方法将动作 a 定义为数量 N_{block} 的子变异策略, 其中每个子变异策略都将种子的某个区块作为主要变异区域。即, 对于一个包含 N_{block} 个区块的种子, 本方法将主要区块的变异概率设定为 $\mathcal{K}/(N_{\text{block}} + \mathcal{K} - 1)$, 将其他 $N_{\text{block}} - 1$ 个区块的变异概率设定为 $1/(N_{\text{block}} + \mathcal{K} - 1)$, 其中 $\mathcal{K}$ 是在前文提到的一个超参数。

5) 奖励。本方法的目的是通过对动作 a 的选

择, 增加所选区块的变异概率, 并尽可能改变种子对应的近似随机场的分布特征。故本方法使用实施动作 a 前后随机场中各个随机变量分布的 JS 散度 (Jensen-Shannon divergence, JSD) 之和计算奖励 R 。该过程可被表示为 $a_{p,q} = \sum_{i=1}^{|X^p \cap X^q|} \text{JSD}(\hat{d}(X_i^p), \hat{d}(X_i^q))$, 其中 $\text{JSD}()$ 是 JS 散度算子, X^p 和 X^q 是实施变异策略 (即动作 a) 前后覆盖的随机场中的随机变量集合。JS 散度能够有效地衡量两个概率分布之间的相似性或差异性 (越接近 1, 差异越大), 且其具有对称性的特点。因此本方法使用其定义奖励, 且该奖励与两个随机场分布特征之间的差异正相关, 从而可以为智能代理提供准确和有意义的奖励信号。

算法 2 展示了本方法基于深度强化学习模型调整模糊测试策略的过程。

算法 2 基于深度强化学习调整模糊测试策略的算法

```

输入 corpus //初始种群
put //被测程序 (program under test)
1) while True
2) seed ← ChooseOneSeed(corpus) //选择种子
3) field ← GetFiled(seed) //获取种子对应随机场
4) if IsUpdateStrategy(seed) then //判断是否更新策略
5) strategy ← UpdateStrategy(seed, field) //更新策略
6) else
7) strategy ← GetStrategy(seed) //获取变异策略
8) end if
9) cnt ← GetMutationCnt(seed) //获取变异次数
10) for  $i \leftarrow 1$  to cnt
11) mutator ← ChooseOneMutator() //选择变异算子
12) pos ← ChoosePostion(seed, strategy) //选择变异位置
13) testcase ← DoMutation(seed, mutator,

```

```
pos) //变异
```

```
14) feedback ← DoExec(testcase, put) //执行被测程序
```

```
15) RandFieldSample(feedback, field) //随机场采样
```

```
16) end for
```

```
17) end while
```

本方法在原有模糊测试样本变异过程中添加了从深度强化学习智能代理处获取变异策略的步骤, 同时也添加了根据模糊测试反馈获取随机场分布特征的步骤。并且, 在各个变异算子选择变异位置时, 本方法基于深度强化学习提供的策略选择变异区块, 替换了样本变异步骤中原有的随机选取变异位置的步骤。

2.3.2 在线学习方法

本小节介绍如何在模糊测试环境中进行高效的在线式学习。本方法使用一种异策略在线式强化学习模型 (支持批量学习的 DQN 模型), 目的是确保强化学习模型在学习的同时将开销控制在合理范围内。本方法使用蒙特卡洛方法确保智能代理在不完全识别环境的情况下学习并优化 Q 网络参数。具体地, 本方法对环境状态进行概率采样, 对经验进行概率记忆, 并利用相对较近的经验批次来训练智能代理。该设计赋予本文的深度强化学习方法一些特性, 包括对模型的马尔可夫特性的弱依赖性和并行决策记忆能力。

智能代理为两个种子并行提供行动的序列图如图 6 所示, 并行决策记忆能力极大地提高了在模糊测试中使用深度强化学习模型的效率。本方法的智能代理可以对处于多个状态的种子并行做出多个行动决策, 并分别收集这些决策的奖励。在积累了一定的经验后, 智能代理会定期从记忆中随机选择经验并从中学习。本方法使用双重深度 Q 网络模型将选择行动和估计目标行动值的过程解耦, 从而避免朴素 DQN 算法中对 Q 值的高估问题。本方法定期将用于行动选择的 Q 网络参数覆盖用于目标行动评估的 Q 网络参数。本方法根据用于目标行动评估 Q 网络预测值 (Q_{target}) 和环境提供的奖励 (R) 来训练 Q 网络, 计算损失的过程可表示为

$$l = \mathcal{L}_{\text{MSE}}(Q(\mathcal{S}, a), R + \gamma \max_{a'} Q_{\text{target}}(\mathcal{S}', a')) \quad (4)$$

其中, $\mathcal{L}_{\text{MSE}}()$ 是均方差损失函数; γ 是折扣因子, 其控制智能代理对未来奖励的权重, 本方法将其设置为 0.5。

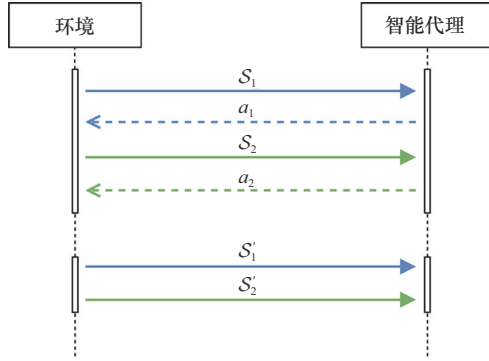


图6 智能代理为两个种子并行提供行动的序列图
Figure 6 Sequence diagram of the agent providing actions for two seeds in parallel

3 实现与评估

3.1 实现

本节简要介绍本方法工程实现的关键部分。为了提取被测程序控制流图和数据流图的统计信息, 本方法使用 LLVM^[21]开发了新的插桩工具, 使用 Clang 和 Clang++ 编译了自定义插件并将其集成至 AFL++^[19]。本方法使用 NetworkX^[22]分析被测程序的控制流图和数据流图, 构建过程间控制流图及过程间比较流图。为了在深度强化学习智能代理和模糊测试器之间实现高效的数据通信链路, 本方法使用 Linux 操作系统的内存文件描述符 (memfd) 和用于进程间通信的本地通信套接字 (AF_UNIX)。此外, 本方法使用 PyTorch^[23]和 DGL^[24]实现深度强化学习模型及其神经网络。本文将实现的原型命名为 COVUZZ。

3.2 评估

本文的评估重点是回答以下研究问题。

1) 本文提出的深度强化学习模型能否提高奖励, 深度强化学习模型的超参数如何设置? 为此, 本文从 FuzzBench^[9]中随机选择了 6 个基准测试目标作为测试的数据集。首先, 本文确定了深度强化学习模型关键的超参数 $\mathcal{K}$, 其决定了指定种子区块与其他区块相比的变异概率的倍率。为此, 本文设定对比对象 COVUZZ_{k2}和

COVUZZ_{k4}。前者将超参数 $\mathcal{K}$ 设定为 2, 后者将 $\mathcal{K}$ 设定为 4。二者进行对比实验以确定超参数 $\mathcal{K}$ 。此外, 为了证明本文提出模型的智能代理在做决策时能否获取更多奖励, 本文设置了消融实验的比较对象。分别是 COVUZZ_{k1} (平均策略) 和 COVUZZ_{rand} (随机策略)。前者给种子文件的所有区块以相同的变异概率且是本文基线方法 AFL++ 的默认策略, 后者随机给种子文件中的某个区块更高的变异概率。

2) 本文提出的方法在覆盖引导的模糊测试方面的性能如何? 为此, 本文使用 FuzzBench 中的全部 24 个基准测试目标作为测试集, 并与基线方法 AFL++^[19]和 Havoc_{MAB} 进行性能对比。本文将各个模糊测试器对每个基准测试目标的模糊测试定义为一个任务, 将每个任务持续 24 h 并重复 5 遍。为了确保计算资源的公平性, 本文为每个模糊测试任务提供了独立的 Docker 容器并为其分配独立的逻辑 CPU。

3) 本文提出的方法在漏洞检测方面的性能如何? 本文使用 Magma 测试集^[10]中的全部 9 个基准测试目标, 并从基准测试目标随机选择 1 个模糊测试程序。本文将原型系统与基线方法 AFL++ 和 Havoc_{MAB} 进行对比, 将每个评估任务持续 24 h 并重复 5 遍。

4) 本研究原型系统开销如何? 本文从研究问题 2) 的实验中收集了原型系统使用 GPU 内存资源的情况, 并与基线方法 AFL++ 对比了模糊测试吞吐量。

所有实验均在一台操作系统为 Ubuntu 22.04 的服务器上进行。该服务器配备 2 个 CPU (Intel Xeon Gold 6326 CPU @ 2.90 GHz)、378 GB 内存、21 TB 固态硬盘 (RAID 设置为 JBOD) 和 2 个 GPU (32 GB Nvidia GPU Tesla V100)。本文接下来按序说明上述研究问题对应的实验结果。

3.2.1 深度强化学习模型的性能及超参数设定

为了证明深度强化学习模型的有效性, 本文设定了对比对象 COVUZZ_{k1}, 其使用与本文基线模糊测试器 AFL++^[19]相同的变异策略。同时, 本文设定了对比对象 COVUZZ_{rand}, 其使用随机的重点区块选择方法, 用于证明导向性优化的有效性。此外, 为了确定深度强化学习模型的超参

数 $\mathcal{K}$, 本文分别设定了 $\mathcal{K}$ 为2和4的对比对象COVUZZ_{k2}和COVUZZ_{k4}。

表1显示了从FuzzBench^[9]中随机抽取的6个基准测试目标程序的代码复杂度指标。分支数量、区域数量、函数数量和行数量均由LLVM提供的llvm-cov插件提供。这些选定基准目标的代码复杂度分布具有一定代表性, 包括简单基准测试目标(如harfbuzz、libpcap)和复杂基准测试目标(如sqlite3、freetype2、libxml2)。本实验对这6个目标进行了12 h的模糊测试, 使用上述4种模糊变异策略生成相应的模糊器, 每个模糊实例重复3次。在测试过程中, 本实验收集每个模糊测试策略在执行推荐动作时获取的奖励值, 以评估这些策略的优劣。

表1 测试集基准测试目标的代码复杂度

Table 1 Code complexity of benchmarks in the test set

基准测试目标	代码复杂度			
	分支数量	区域数量	函数数量	行数量
sqlite3	28 832	53 886	2 275	66 153
libpcap	7 618	12 283	539	16 811
harfbuzz	6 976	11 637	1 507	13 616
freetype2	26 740	56 626	1 984	61 918
libxml2	62 870	84 441	2 630	87 274
libxslt	49 670	67 040	2 241	69 759

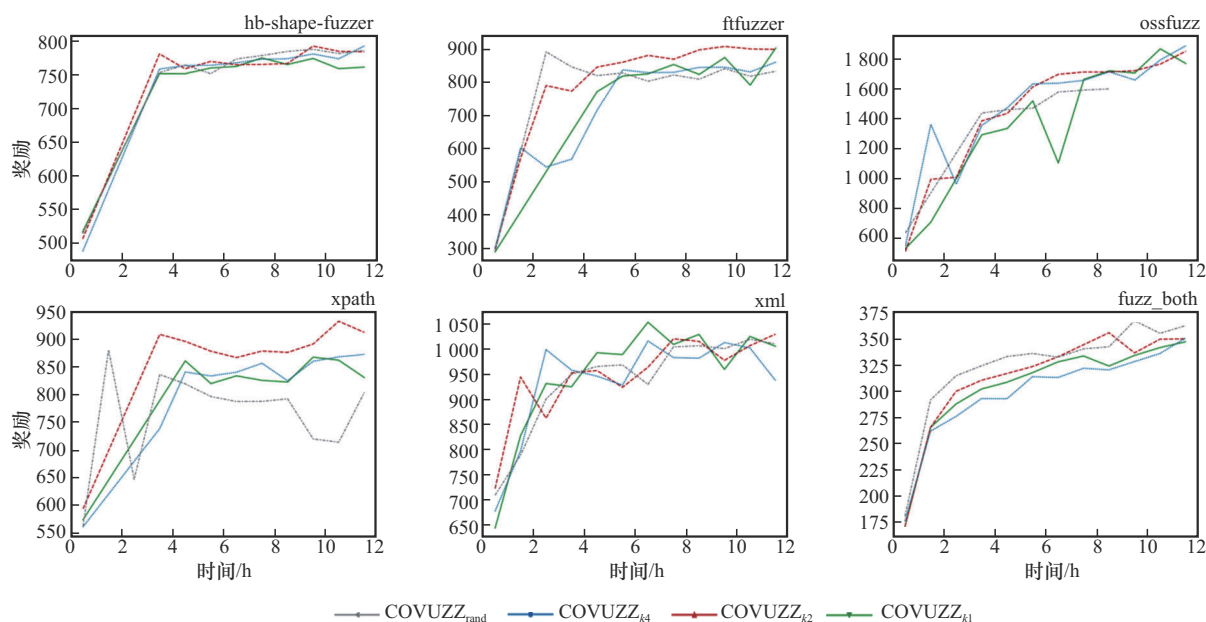


图7 每个策略在12 h内每小时获得的平均奖励

Figure 7 The mean reward obtained by each strategy in each hour over 12 hours

如图7所示, 本实验统计了每种变异策略在每个规定时间间隔(1 h)内获得的奖励值的平均值。该图直观地展示了每种策略随时间推移对奖励的影响。

本文的基线模糊测试器AFL++使用的默认策略(COVUZZ_{k1})为所有种子区块分配了相同的变异概率, 与之相比, COVUZZ_{k2}在6个基准测试目标中的5个基准测试目标上取得了明显改善。与赋予所选区块更大变异概率的策略(COVUZZ_{k4})相比, COVUZZ_{k2}在6个基准测试目标中的5个基准测试目标上取得了更好的效果。此外, 随机策略(COVUZZ_{rand})在一个基准测试目标(libpcap的fuzz_both)上的表现优于COVUZZ_{k2}, 但在一些基准测试目标(libxslt的xpath和freetype2的ftfuzzer)上, 随机策略不能保证奖励随时间增长。

同时, 本实验统计了每种变异策略在12 h内的平均奖励, 并将其展示在表2中。表中加粗部分为在每个基准测试目标上表现最好的变异策略。实验结果显示, COVUZZ_{k2}的表现最好, 而且明显优于AFL++默认的变异策略(COVUZZ_{k1})和随机选择重点区块的变异策略(COVUZZ_{rand})。

实验表明, 本文提出的深度强化学习模型能够明显提高被测目标分支判断变量随机场分布的分布

散度。并且,通过对比实验,本文将超参数 K 设定为2,并将其作为后续测试中本文原型系统的参数。

表2 各策略在12 h内获得的平均奖励

Table 2 Mean reward of each strategy within 12 hours

基准测试目标	平均奖励			
	COVUZZ _{k1}	COVUZZ _{k2}	COVUZZ _{k4}	COVUZZ _{rand}
sqlite3	728.12	738.07	731.29	735.89
libpcap	724.29	810.05	731.77	786.28
harfbuzz	1 370.17	1 475.04	1 496.60	976.27
freetype2	795.06	852.75	788.98	769.50
libxml2	961.92	955.95	949.74	946.32
libxslt	308.53	317.03	301.14	327.28

3.2.2 覆盖引导的模糊测试性能

在进行覆盖引导的模糊测试性能实验前,本实验从源码编译了FuzzBench^[9]中所有24个及基准测试目标,统计了这些测试目标的代码复杂度指标和使用本文方法生成的过程间比较流图的复杂度,结果如图8所示。

实验测试集的基准测试目标程序能够代表常见用户空间程序的代码复杂度,其LLVM区域数量从数千到数十万不等。结果表明,插桩程序为这些基准测试目标生成的过程间比较流图有效降低了过程间控制流图的复杂度,因为过程间比较流图只保留了与分支判断变量相关的控制流节点。

本实验的对比对象是本文工作的基线模糊测试器AFL++、在线优化模糊测试变异算法的解决方案Havoc_{MAB}^[25]以及REDQUEEN^[7]在AFL++中

的增强集成功能CmpLog。Havoc_{MAB}将模糊测试变异算子的选择建模为多臂老虎机(multi-armed bandit, MAB)问题,并使用UCB-1(upper confidence bound 1)算法处理该问题,从而在线优化模糊测试变异策略。Havoc_{MAB}使用了解决特定MAB问题的强化学习算法UCB,是使用强化学习算法优化模糊测试的最新工作之一。CmpLog则利用比较指令的反馈信息优化模糊测试的变异算子及变异策略,旨在优化模糊测试对被测程序中的幻数比较以及嵌套校验和检验等逻辑的处理。

本实验为每个模糊测试实例分配了独立的CPU资源,持续运行每个实例24 h,并重复每个测试实例5次。为确保公平性,本实验使用llvm-cov编译的统一测量程序,以30 min为间隔评估各个模糊测试器的覆盖率指标。图9展示了AFL++、COVUZZ、Havoc_{MAB}和CmpLog在FuzzBench所有24个基准测试目标上的24 h平均分支覆盖增长,阴影部分表示95%置信区间。同时,本实验统计了4个实验对象24 h的平均覆盖情况,其结果如表3所示。

实验结果表明,COVUZZ在所有的基准测试目标上均优于(或等同于)基线模糊测试器AFL++,并在一些基准测试目标(如proj4、libxslt、sqlite3、openh264、libpcap、freetype2等)上取得了明显的性能提升。总体上,在24 h模糊测试中,COVUZZ在代码区域平均覆盖方面相对AFL++提高3.09%,在代码分支平均覆盖

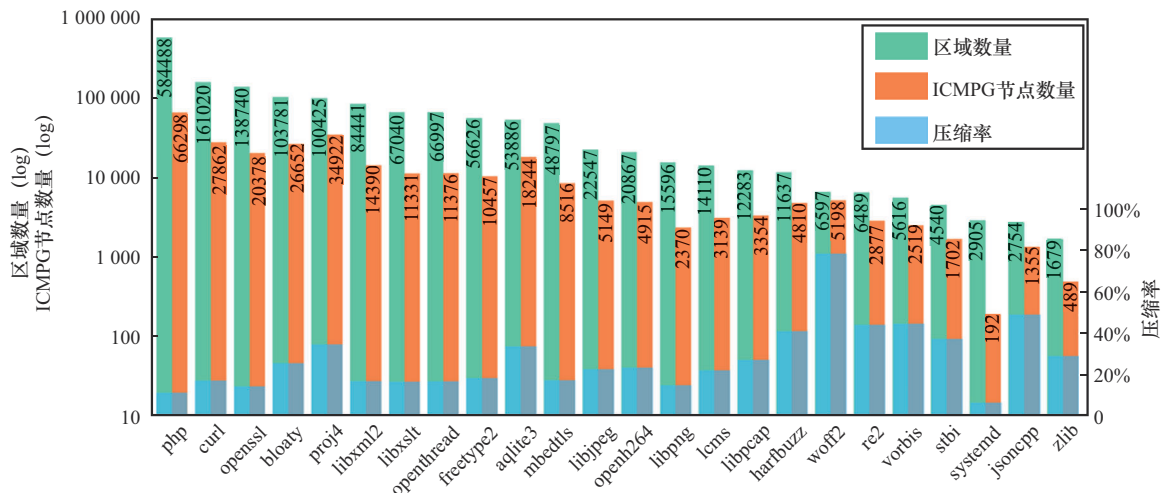


图8 FuzzBench基准测试目标代码复杂度及对应ICMPG复杂度 and 压缩率

Figure 8 Code complexity and corresponding ICMPG complexity and compression ratio for benchmarks in FuzzBench

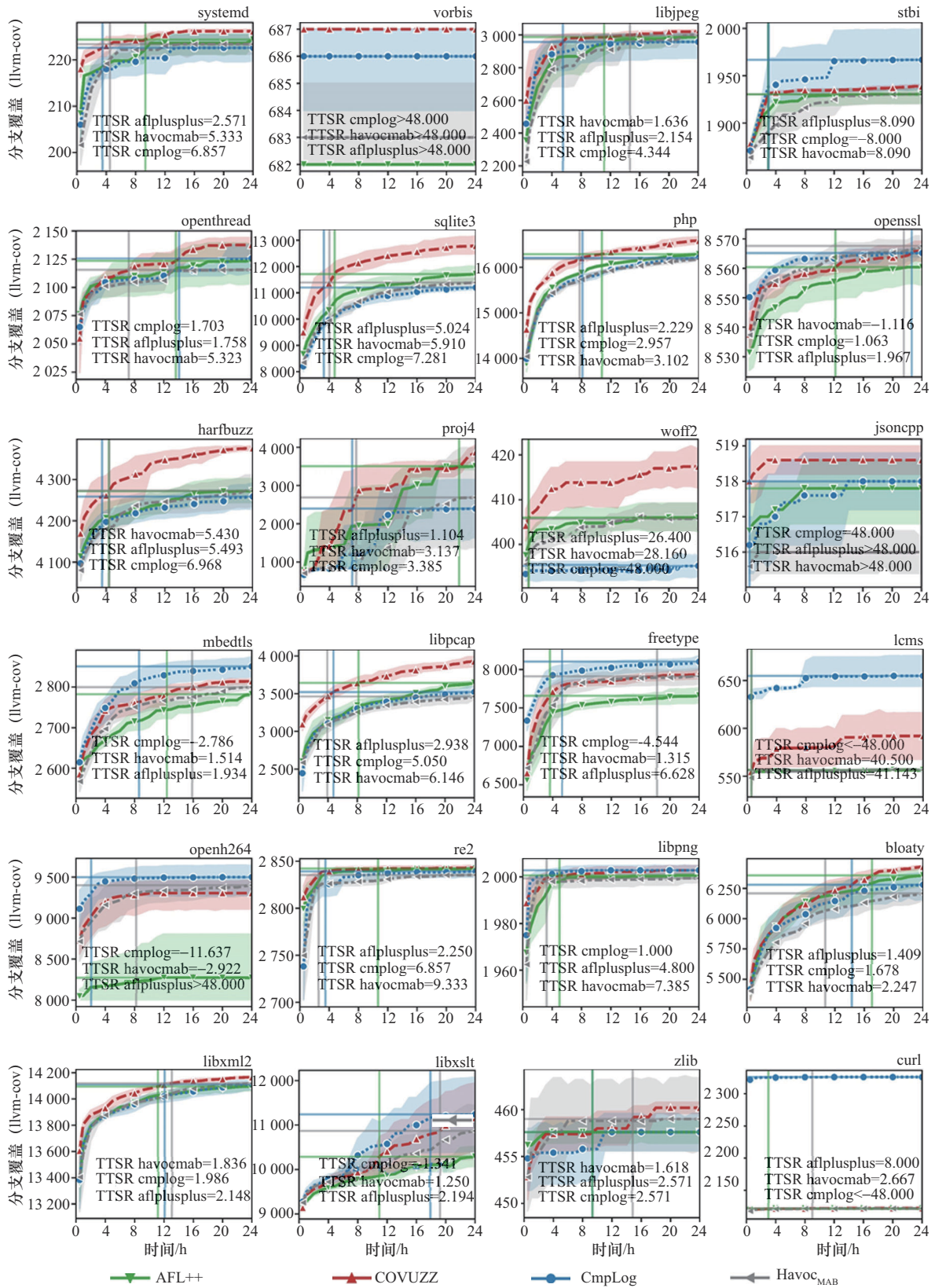


表3 COVUZZ与对比对象在FuzzBench所有24个基准测试目标上的24 h平均覆盖情况

Table 3 24-hour average coverage of COVUZZ and comparison objects on all 24 benchmark targets of FuzzBench

基准测试目标	区域覆盖				分支覆盖			
	COVUZZ	AFL++	Havoc _{MAB}	CmpLog	COVUZZ	AFL++	Havoc _{MAB}	CmpLog
php	46 313.00 (7.92%)	45 525.60 (7.79%)	45 190.20 (7.73%)	45 405.60 (7.77%)	16 582.40 (9.43%)	16 287.00 (9.26%)	16 181.40 (9.20%)	16 208.40 (9.22%)
curl	5 103.20 (3.17%)	5 102.60 (3.17%)	5 102.80 (3.17%)	5 693.00 (3.54%)	2 122.20 (2.43%)	2 121.60 (2.43%)	2 121.80 (2.43%)	2 326.00 (2.67%)
openssl	20 660.60 (14.89%)	20 652.80 (14.89%)	20 665.20 (14.89%)	20 663.20 (14.89%)	8 565.80 (13.01%)	8 560.40 (13.00%)	8 567.60 (13.01%)	8 565.20 (13.01%)
bloaty	8 514.20 (8.20%)	8 622.80 (8.31%)	8 157.20 (7.86%)	8 144.20 (7.85%)	6 429.20 (8.44%)	6 356.20 (8.34%)	6 208.20 (8.15%)	6 279.40 (8.24%)
proj4	7 508.00 (7.48%)	6 957.80 (6.93%)	5 177.80 (5.16%)	4 721.00 (4.70%)	3 871.20 (5.97%)	3 508.20 (5.41%)	2 689.60 (4.15%)	2 395.60 (3.69%)
libxml2	20 754.80 (24.58%)	20 663.20 (24.47%)	20 690.40 (24.50%)	20 680.60 (24.49%)	14 165.40 (22.53%)	14 095.80 (22.42%)	14 118.80 (22.46%)	14 109.80 (22.44%)
libxslt	18 005.60 (26.86%)	16 625.20 (24.80%)	17 583.20 (26.23%)	18 154.20 (27.08%)	11 161.00 (22.47%)	10 284.60 (20.71%)	10 868.80 (21.88%)	11 240.80 (22.63%)
openthread	6 939.80 (10.36%)	6 918.80 (10.33%)	6 906.40 (10.31%)	6 922.60 (10.33%)	2 137.60 (7.98%)	2 123.60 (7.93%)	2 115.60 (7.90%)	2 125.60 (7.93%)
freetype2	19 845.20 (35.05%)	19 291.20 (34.07%)	19 728.00 (34.84%)	20 196.40 (35.67%)	7 946.60 (29.72%)	7 658.80 (28.64%)	7 911.20 (29.59%)	8 102.20 (30.30%)
sqlite3	27 734.60 (51.47%)	25 574.80 (47.46%)	24 924.40 (46.25%)	24 536.00 (45.53%)	12 789.00 (44.36%)	11 712.80 (40.62%)	11 390.00 (39.50%)	11 200.00 (38.85%)
mbedtls	8 757.40 (17.95%)	8 679.00 (17.79%)	8 715.00 (17.86%)	8 750.80 (17.93%)	2 814.60 (13.82%)	2 782.40 (13.66%)	2 799.60 (13.74%)	2 850.80 (13.99%)
libjpeg	5 701.40 (25.29%)	5 650.60 (25.06%)	5 680.40 (25.19%)	5 601.80 (24.84%)	3 020.40 (27.79%)	2 986.80 (27.48%)	2 998.40 (27.59%)	2 957.40 (27.21%)
openh264	17 562.00 (84.16%)	16 478.20 (78.97%)	17 671.00 (84.68%)	17 756.20 (85.09%)	9 304.80 (76.02%)	8 271.80 (67.58%)	9 400.20 (76.80%)	9 498.80 (77.60%)
libpng	4 724.00 (30.29%)	4 723.00 (30.28%)	4 721.20 (30.27%)	4 723.00 (30.28%)	2 002.80 (33.42%)	2 000.60 (33.39%)	1 999.40 (33.37%)	2 002.80 (33.42%)
lcms	1 148.80 (8.14%)	1 084.20 (7.68%)	1 084.20 (7.68%)	1 194.40 (8.46%)	592.00 (7.75%)	556.20 (7.28%)	556.40 (7.28%)	654.80 (8.57%)
libpcap	5 796.80 (47.19%)	5 412.00 (44.06%)	5 152.00 (41.94%)	5 195.00 (42.29%)	3 928.40 (51.57%)	3 642.60 (47.82%)	3 463.00 (45.46%)	3 521.20 (46.22%)
harfbuzz	7 756.80 (66.66%)	7 582.80 (65.16%)	7 595.00 (65.27%)	7 551.60 (64.89%)	4 376.00 (62.73%)	4 271.80 (61.24%)	4 273.20 (61.26%)	4 258.60 (61.05%)
woff2	691.80 (10.49%)	672.40 (10.19%)	670.25 (10.16%)	657.00 (9.96%)	417.40 (9.15%)	406.00 (8.90%)	405.75 (8.89%)	395.20 (8.66%)
re2	3 937.20 (60.67%)	3 934.20 (60.63%)	3 931.40 (60.59%)	3 932.40 (60.60%)	2 843.00 (63.49%)	2 842.20 (63.47%)	2 835.40 (63.32%)	2 839.20 (63.40%)
vorbis	1 302.00 (23.18%)	1 297.00 (23.09%)	1 298.00 (23.11%)	1 301.00 (23.17%)	687.00 (18.76%)	682.00 (18.62%)	683.00 (18.65%)	686.00 (18.73%)
stbi	3 165.60 (69.73%)	3 157.40 (69.55%)	3 154.80 (69.49%)	3 209.80 (70.70%)	1 938.80 (65.10%)	1 930.40 (64.82%)	1 930.40 (64.82%)	1 966.80 (66.04%)
systemd	606.00 (20.86%)	601.00 (20.69%)	599.80 (20.65%)	595.20 (20.49%)	226.20 (22.62%)	224.40 (22.44%)	223.40 (22.34%)	222.60 (22.26%)
jsoncpp	671.80 (24.39%)	671.20 (24.37%)	669.00 (24.29%)	671.20 (24.37%)	518.60 (24.39%)	517.80 (24.36%)	516.00 (24.27%)	518.00 (24.37%)
zlib	968.00 (57.65%)	966.00 (57.53%)	969.40 (57.74%)	966.20 (57.55%)	460.20 (50.24%)	457.60 (49.96%)	459.00 (50.11%)	457.60 (49.96%)
平均	10 173.69 (30.69%)	9 868.49 (29.89%)	9 834.88 (29.99%)	9 884.27 (30.10%)	4 954.19 (28.88%)	4 761.73 (27.91%)	4 779.84 (28.17%)	4 807.62 (28.35%)
相对增长		3.09%	3.45%	2.93%		4.04%	3.65%	3.05%

方面相对AFL++提高4.04%。这表明本文提出的方法能够明显优化基线模糊测试器的性能。

与Havoc_{MAB}相比,本文原型系统在24个基准测试目标中的22个上性能更优(或等同),并在一些基准测试目标(如woff2、libpcap、sqlite3、libxslt、proj4等)上的表现明显更优。总体上,在代码区域平均覆盖方面相比Havoc_{MAB}提高3.45%,COVUZZ在代码分支平均覆盖方面相比Havoc_{MAB}提高3.65%。这表明本文提出的方法比其他相关工作的性能更优。

与CmpLog相比,本文原型系统在24个基准测试目标中的17个性能更优(或相同),并在sqlite3、php、harfbuzz、libpcap等目标上的表现明显更优。总体上,在24 h模糊测试中,在代码区域覆盖方面相比CmpLog提高2.93%,在代码分

支覆盖方面相比CmpLog提高3.05%。这表明本文提出的方法比利用分支比较反馈信息指导模糊测试变异策略的工业界主流方案性能更优。

除了总覆盖率,本实验还评估了COVUZZ相较于对比对象的速度提升。本实验将达到相同覆盖率所需的时间(time to the same, TTS)作为评估指标。Entropic^[13]的工作中也使用了这一指标。在此基础上,本实验引入TTSR(达到相同覆盖率所需的时间的比率)指标来评估某个模糊测试器相对其他模糊测试器的速度提升。计算TTSR的过程可以表示为

$$TTSR(b,a) = \frac{TTS(b, \min(\text{cover}_{\text{end}}(a), \text{cover}_{\text{end}}(b)))}{TTS(a, \min(\text{cover}_{\text{end}}(a), \text{cover}_{\text{end}}(b)))} \quad (5)$$

本实验的统计结果如表4和图9所示, TTSR

表4 COVUZZ相较于对比对象的速度提升
Table 4 Speedup of COVUZZ against competitors

基准测试目标	TTSR					
	AFL++		Havoc _{MAB}		CmpLog	
	区域覆盖	分支覆盖	区域覆盖	分支覆盖	区域覆盖	分支覆盖
php	2.11	2.23	3.14	3.10	2.51	2.96
curl	8.00	8.00	2.67	2.67	< -48.00	< -48.00
openssl	1.92	1.97	-1.69	-1.12	-1.98	1.06
bloaty	-1.11	1.41	3.47	2.25	3.81	1.68
proj4	1.09	1.10	3.23	3.14	4.01	3.39
libxml2	2.53	2.15	2.12	1.84	2.24	1.986
libxslt	2.18	2.19	1.24	1.25	-1.36	-1.34
openthread	1.79	1.76	4.06	3.32	1.71	1.70
freetype2	6.75	6.63	1.70	1.32	-5.87	-4.54
sqlite3	5.20	5.02	6.19	5.91	7.77	7.28
mbdttls	1.74	1.93	1.58	1.51	1.21	-2.79
libjpeg	4.09	2.15	1.72	1.64	4.59	4.34
openh264	> 48.00	> 48.00	-3.30	-2.92	-11.45	-11.64
libpng	2.40	4.80	7.39	7.39	2.40	1.00
lcms	43.60	41.14	43.60	40.50	< -48.00	< -48.00
libpcap	2.42	2.94	5.06	6.15	4.56	5.05
harfbuzz	6.57	5.49	5.71	5.43	12.33	6.97
woff2	> 48.00	26.40	> 48.00	28.16	> 48.00	> 48.00
re2	8.44	2.25	9.34	9.33	9.00	6.86
vorbis	> 48.00	> 48.00	> 48.00	> 48.00	> 48.00	> 48.00
stbi	4.65	8.09	8.10	8.09	-8.41	-8.00
systemd	4.88	2.57	5.38	5.33	8.91	6.86
jsoncpp	48.00	> 48.00	> 48.00	> 48.00	48.00	48.00
zlib	2.48	2.57	-4.25	1.62	2.40	2.57
中位数	2.53	2.57	3.23	3.12	2.46	2.28

出现大于 (<) 和小于 (>) 的情况是因为本实验的统计粒度为 30 min, 故在 30 min 以后一直小于对比对象则用小于 (<) 符号表示, 反之类似。此外, 实验结果中较差的部分则在表中用黑色标出。

实验结果表明, 相比 AFL++、Havoc_{MAB}, COVUZZ 在接近所有基准测试目标上能够以更短的时间达到相同的覆盖率; 相比 CmpLog, COVUZZ 在大多数基准测试目标上能够以更短的时间到达相同的覆盖率。相较于 AFL++, 在 FuzzBench 所有 24 个基准测试目标中速度倍率的中位数为 2.5 倍以上; 相较于 Havoc_{MAB}, 速度倍率的中位数为 3.1 倍以上; 相较于 CmpLog, 速度倍率的中位数为 2.2 倍以上。这说明本文提出

的方法具备更强的模糊测试代码覆盖能力。

此外, 由于模糊测试具有较强随机性, 为了进一步验证本文方法在重复实验中是否具有优势, 本文使用 Vargha 和 Delaney 提出的 $\hat{A}_{12}$ 指标对实验结果进行统计。 $\hat{A}_{12}$ 指标用于比较两个独立样本的分布, 衡量一组中的观察值在多大程度上优于另一组的观察值。当 $\hat{A}_{12}$ 大于 0.5 时, 表明一组分布优于另一组分布的概率大于 50%, 越接近 1 说明概率越大。COVUZZ 相较于对比对象的 $\hat{A}_{12}$ 度量结果如表 5 所示, 且较差部分在表中用黑色标出。

表 5 表明, 针对 FuzzBench 测试集所有基准测试目标 24 h 的模糊测试中, 相对于基线模糊测试器 AFL++ 以及使用强化学习指导种

表 5 COVUZZ 相较于对比对象的 $\hat{A}_{12}$ 度量结果
Table 5 $\hat{A}_{12}$ results of COVUZZ against competitors

基准测试目标	$\hat{A}_{12}$					
	AFL++		Havoc _{MAB}		CmpLog	
	区域覆盖	分支覆盖	区域覆盖	分支覆盖	区域覆盖	分支覆盖
php	1.00	1.00	1.00	1.00	1.00	1.00
curl	0.76	0.68	0.68	0.68	0.00	0.00
openssl	0.66	0.68	0.28	0.42	0.36	0.54
bloaty	0.48	1.00	1.00	1.00	0.90	1.00
proj4	0.92	1.00	0.68	0.64	1.00	1.00
libxml2	1.00	1.00	1.00	1.00	1.00	0.88
libxslt	0.68	0.92	0.64	0.68	0.44	0.40
openthread	0.70	0.92	1.00	0.92	0.78	0.72
freetype2	1.00	0.78	0.72	0.64	0.00	0.16
sqlite3	1.00	0.90	1.00	1.00	1.00	1.00
mbedtls	1.00	1.00	0.88	0.84	0.32	0.16
libjpeg	0.84	1.00	0.86	0.90	0.80	0.88
openh264	0.92	0.96	0.36	0.32	0.16	0.16
libpng	0.80	0.58	1.00	0.86	0.70	0.52
lcms	0.88	1.00	0.88	0.86	0.48	0.00
libpcap	1.00	0.90	1.00	1.00	1.00	1.00
harfbuzz	1.00	0.84	1.00	1.00	1.00	1.00
woff2	0.96	0.74	1.00	0.95	1.00	1.00
re2	0.92	0.96	1.00	1.00	0.96	0.66
vorbis	1.00	1.00	0.90	0.90	0.60	0.60
stbi	0.72	0.68	0.84	0.90	0.50	0.46
systemd	0.80	0.68	0.80	0.84	0.80	0.74
jsoncpp	0.60	1.00	1.00	1.00	0.68	0.68
zlib	0.86	1.00	0.62	0.64	0.78	0.84
中位数	0.90	0.94	0.89	0.90	0.78	0.70

子变异的模糊测试器 Havoc_{MAB}，本文提出的方法能够以更高的概率获取更高的覆盖率。相较 CmpLog，本文的原型系统在大部分基准测试目标上能够以更高的概率获取更高的覆盖率。

最后，为了确定本文原型系统在某些基准测试目标上性能不佳的原因，本实验利用 LLVM 编译器的 llvm-cov 工具对代码覆盖情况进行了进一步的分析。经过分析，本文发现 CmpLog 在生成满足模式匹配条件分支的样本方面具有较强的能力。COVUZZ 与 CmpLog 在基准测试目标 stbi 上代码覆盖的差异典型的例子如图 10 所示，CmpLog 在模糊测试中执行第 6 347 行的幻数比较语句共约 1.70×10^3 次，其中 208 次成功满足了该条件判断并成功执行了后续代码，且这 208 次执行中又有 203 次满足了另一个幻数比较，并确保了该函数的正确返回。相比之下，COVUZZ 执行了该函数共约 1.41×10^3 次，但是未能获得一次该函数的正确返回。这主要是因为 CmpLog 面向幻数比较语句优化了变异算子，而 COVUZZ 仅面向数据流分布多样性优化了变异策略。二者的方法并不互斥，在后续研究中将二者进行结合具备可行性。

此外，本实验还对 COVUZZ 在大多数基准测试目标上性能较强的原因进行了分析。其中一个关键的原因是 COVUZZ 能够通过多样化一些分支判断变量的取值以获得更多的分支探索。从图 11 可以发现 COVUZZ 比 CmpLog 探索了更多的选择跳转语句的分支，从而探索了被测程序 libpcap 更多的代码空间。

本节实验对研究问题 2) 进行了回答，验证了

本文所提方法的原型系统在覆盖引导的模糊测试方面的性能。实验证明本文原型系统 COVUZZ 在覆盖引导的模糊测试方面的性能明显优于基线模糊测试器 AFL++ 和 Havoc_{MAB}，并在大多数基准测试目标上优于利用比较指令反馈优化模糊测试变异算子及策略的工作 CmpLog。

行索引	执行次数	行索引	执行次数
5377	static struct block *	5377	static struct block *
5383	2.66k igen_proto_abbrev_internal(...., ...)	5378	1.04k igen_proto_abbrev_internal(...., ...)
5385	14 switch (proto) {	5383	1.04k switch (proto) {
5389	373 case Q_SCTP: {	5385	2 case Q_SCTP: {
5393	191 case Q_TCP: {	5389	109 case Q_TCP: {
5397	885 case Q_ICMP: {	5393	11 case Q_ICMP: {
5405	101 case Q_IGMP: {	5397	86 case Q_ICMP: {
5412	41 case Q_IGRP: {	5405	31 case Q_IGMP: {
5420	0 case Q_PPM: {	5412	11 case Q_IGRP: {
5428	21 case Q_VRRP: {	5420	0 case Q_PPM: {
5436	241 case Q_CARP: {	5428	0 case Q_VRRP: {
5440	4581 case Q_LSP: {	5436	11 case Q_CARP: {
5444	54 case Q_ARP: {	5440	1091 case Q_LSP: {
5448	41 case Q_RARP: {	5444	81 case Q_ARP: {
5452	21 case Q_IINK: {	5448	11 case Q_RARP: {
5455	0 case Q_ATAK: {	5452	11 case Q_IINK: {
5459	0 case Q_AARP: {	5455	0 case Q_ATAK: {
5463	0 case Q_DECNET: {	5459	31 case Q_AARP: {
5467	371 case Q_SCA: {	5463	0 case Q_DECNET: {
5471	41 case Q_LAT: {	5467	11 case Q_SCA: {
5475	0 case Q_MOPDL: {	5471	11 case Q_LAT: {
5479	0 case Q_MOPRC: {	5475	0 case Q_MOPDL: {
5483	1001 case Q_IPV6: {	5479	0 case Q_MOPRC: {
5490	1461 case Q_ICMPV6: {	5483	251 case Q_IPV6: {
5497	81 case Q_AH: {	5490	121 case Q_ICMPV6: {
5504	151 case Q_ESP: {	5497	51 case Q_AH: {
5508	231 case Q_ESP: {	5504	11 case Q_ESP: {
5512	241 case Q_ESIS: {	5508	71 case Q_ESP: {
5516	61 case Q_ISIS: {	5512	0 case Q_ESIS: {
5520	211 case Q_ISIS_L1: {	5516	0 case Q_ISIS: {
5522	241 case Q_ISIS_L2: {	5520	4541 case Q_ISIS_L1: {
5544	0 case Q_ISIS_I1H: {	5522	1821 case Q_ISIS_L2: {
5552	311 case Q_ISIS_LSP: {	5544	0 case Q_ISIS_I1H: {
5558	1801 case Q_ISIS_SNP: {	5552	121 case Q_ISIS_LSP: {
5568	71 case Q_ISIS_CSNP: {	5558	51 case Q_ISIS_SNP: {
5574	81 case Q_ISIS_PSNP: {	5568	0 case Q_ISIS_CSNP: {
5580	121 case Q_CLNP: {	5574	0 case Q_ISIS_PSNP: {
5584	1621 case Q_STP: {	5580	0 case Q_CLNP: {
5588	51 case Q_IPX: {	5584	71 case Q_STP: {
5592	0 case Q_NETBEUI: {	5588	61 case Q_IPX: {
5596	0 case Q_RADIO: {	5592	0 case Q_NETBEUI: {
5599	0 default: {	5596	0 case Q_RADIO: {
5600	0 abort();	5599	0 default: {
5601	2.66k }	5600	0 abort();
5602	2.62k return b1;	5601	1.04k }
5603	2.66k }	5602	1.01k return b1;
	COVUZZ	5603	1.04k }
			CmpLog

图 11 COVUZZ 与 CmpLog 在基准测试目标 libpcap 上代码覆盖的差异

Figure 11 Differences in code coverage between COVUZZ and CmpLog on the benchmark libpcap

3.2.3 漏洞检测能力

为了评估本文方法的漏洞检测能力，本实验从 Magma^[10]测试集的 9 个软件项目中各选择了一个基准测试目标以形成测试集。Magma 测试集包含了 9 个软件项目的 138 个错误，其中大部分错误为获得漏洞编号的真实世界漏洞。本实验为每个基准测试目标进行了 24 h 模糊测试，每个模

行索引	执行次数	行索引	执行次数
6343	static int stbi_pic_test_core(stbi_context *s)	6343	static int stbi_pic_test_core(stbi_context *s)
6344	1.70×10 ³ {	6344	1.41×10 ³ {
6345	1.70×10 ³ int i;	6345	1.41×10 ³ int i;
6346		6346	
6347	1.70×10 ³ if (!stbi_pic_is4(s, "\x53\x80\xf6\x34"))	6347	1.41×10 ³ if (!stbi_pic_is4(s, "\x53\x80\xf6\x34"))
6348	1.49×10 ³ return 0;	6348	1.41×10 ³ return 0;
6349		6349	
6350	17.6×10 ³ for(i=0; i<84; ++i)	6350	340 for(i=0; i<84; ++i)
6351	17.4×10 ³ stbi_get8(s);	6351	336 stbi_get8(s);
6352		6352	
6353	208 if (!stbi_pic_is4(s, "P1CT"))	6353	4 if (!stbi_pic_is4(s, "P1CT"))
6354	51 return 0;	6354	4 return 0;
6355		6355	
6356	203 return 1;	6356	0 return 1;
6357	208 }	6357	4 }
	CmpLog		COVUZZ

图 10 COVUZZ 与 CmpLog 在基准测试目标 stbi 上代码覆盖的差异

Figure 10 Differences in code coverage between COVUZZ and CmpLog on the benchmark stbi

糊测试实例重复 5 次。实验统计 5 次模糊测试后 COVUZZ、AFL++ 以及 Havoc_{MAB} 总计覆盖的漏洞代码片段以及触发的漏洞。各模糊测试漏洞到达情况和漏洞触发情况的统计结果如表 6 和表 7 所示。表中的基准测试目标按序为 lua 软件项目的 lua、poppler 的 pdf_fuzzer、php 的 exif、libpng 的 libpng_read_fuzzer、libsndfile 的 sndfile_fuzzer、sqlite3 的 sqlite3_fuzz、openssl 的 x509、libtiff 的 tiff_read_rgba_fuzzer、libxml2 的 libxml2_xml_read_memory_fuzzer 对应的漏洞编号前缀。

表 6 各模糊测试器漏洞到达情况

Table 6 Vulnerabilities reached by each fuzzers

基准测试目标	COVUZZ	AFL++	Havoc _{MAB}	CmpLog
LUA	3	3	3	3
PDF	16	15	16	16
PHP	5	5	5	5
PNG	6	6	6	6
SND	8	8	8	8
SQL	14	16	13	16
SSL	5	5	5	5
TIFF	8	8	7	7
XML	9	8	9	8
总计	74	74	72	74

表 7 各模糊测试器漏洞触发情况

Table 7 Vulnerabilities triggered by each fuzzers

基准测试目标	COVUZZ	AFL++	Havoc _{MAB}	CmpLog
LUA	2	2	2	2
PDF	3	5	7	4
PHP	3	3	3	3
PNG	2	2	2	3
SND	7	7	7	7
SQL	7	7	5	6
SSL	1	0	0	0
TIFF	5	4	4	5
XML	6	4	6	4
总计	36	34	36	34

实验结果表明, 本文原型系统与基线方法 AFL++ 以及 CmpLog 在漏洞到达方面的能力接近, 但是优于 Havoc_{MAB}。同时, 本文原型系统与 Havoc_{MAB} 在漏洞触发方面的能力接近, 但是优于基线方法。在漏洞触发能力方面, 本文原型系统仅在 poppler 的 pdf_fuzzer 这一个基准测试目标上弱于基线方法 AFL++ 和 Havoc_{MAB}, 以及在 libpng 的 libpng_read_fuzzer 上弱于 CmpLog。但是, 本文原型系统在 openssl 的 x509、libtiff 的 tiff_read_rgba_fuzzer 以及 libxml2 的 libxml2_xml_read_memory_fuzzer 这 3 个基准测试目标上优于基线方法 AFL++; 在 sqlite3 的 sqlite3_fuzz、openssl 的 x509 以及 libtiff 的 tiff_read_rgba_fuzzer 这 3 个基准测试目标上优于 Havoc_{MAB}; 并在 openssl 的 x509、libxml2 的 libxml2_xml_read_memory_fuzzer 上优于 CmpLog。

通过分析, 本实验发现 COVUZZ 的变异策略使其能够以较小的概率触发 SSL009 (CVE-2017-3735, 单字节溢出漏洞)、TIF002 (CVE-2016-5314, 缓冲区溢出漏洞)、XML002 (CVE-2017-0663, RCE 漏洞) 以及 XML012 (CVE-2016-1836, UAF 漏洞) 等漏洞。这说明本文的方法在一些真实软件目标上具备更强的漏洞触发能力。

3.2.4 开销

本实验统计了 COVUZZ 和 AFL++ 在第 3.2.2 节所述实验中的每个模糊测试实例在 24 h 内的平均总吞吐量, 以及 COVUZZ 在第 3.2.2 节所述实验中的 GPU 内存开销。其实验结果如表 8 所示。

与 AFL++ 相比, 在 FuzzBench 的 24 个基准测试目标中, COVUZZ 的吞吐量平均降低 8.8%, 最高降低 36.17%, 这主要是由深度强化学习模型和模糊测试器的串行处理部分引入的计算开销, 但该开销处于合理范围内。

此外, COVUZZ 平均需要 924.08 MB 的额外 GPU 内存, 且最大需求不超过 2 GB。这是由深度强化学习模型中的双重深度 Q 网络引入的。COVUZZ 在引入开销后仍能保持性能优势, 说明本文的原型系统具备工业使用的价值, 亦说明本文提出方法在样本变异方面的优化明显。

表8 COVUZZ性能开销
Table 8 The overhead of COVUZZ

基准 测试目标	吞吐量/个·秒 ⁻¹			GPU内存/MB
	AFL++	COVUZZ	相对增益	COVUZZ
bloaty	3 932.65	4 254.92	8.19%	882
curl	15 519.15	16 837.09	8.49%	796
freetype	16 725.18	14 007.14	-16.25%	1 864
harfbuzz	18 031.66	15 822.01	-12.25%	1 368
jsoncpp	26 516.27	22 568.19	-14.89%	738
lcms	28 054.66	22 982.90	-18.08%	744
libjpeg	12 698.23	10 764.95	-15.22%	706
libpcap	8 682.85	6 830.99	-21.33%	816
libpng	49 499.14	50 741.60	2.51%	888
libxml2	2 521.91	1 908.04	-24.34%	748
libxslt	5 012.13	4 467.27	-10.87%	932
mbedtls	989.37	951.72	-3.81%	802
openh264	27.57	18.64	-32.39%	702
openssl	9 494.62	12 813.68	34.96%	976
openthread	3 453.96	3 323.41	-3.78%	804
php	8 161.30	8 124.66	-0.45%	786
proj4	9 186.96	7 872.70	-14.31%	798
re2	16 295.27	10 959.80	-32.74%	826
sqlite3	9 105.29	8 763.13	-3.76%	1 534
stbi	7 564.86	4 828.77	-36.17%	1 500
systemd	1 796.62	1 463.57	-18.54%	462
vorbis	8 433.46	7 124.38	-15.52%	990
woff2	23 848.94	30 765.84	29.00%	730
zlib	67 598.28	67 877.39	0.41%	786
均值	14 714.60	14 003.03	-8.80%	924.08

4 相关工作

4.1 覆盖引导的模糊测试

覆盖引导的模糊测试已经在工业界被广泛应用于漏洞挖掘的工作。然而，其在处理复杂输入格式和探索被测目标深层路径方面仍然面临挑战。近年来，学术界对覆盖引导的模糊测试的兴趣激增，从而开展了大量旨在增强其性能的研究。研究人员使用污点分析（taint analysis）^[26-27]、符号执行（symbolic execution）^[7,28]、程序转换（program transformation）^[29]等方法优化覆盖引导的模糊测试，以提高其效率。这些方法利用程序分析方法辅助模糊测试，从而提高模糊测试探索被测程序深层代码以及漏洞检测的能力。但受限于现有程序分析方

法存在的诸多限制，如静态分析的不准确、符号执行的路径爆炸等，使用传统程序分析方法辅助模糊测试难以确保模型的准确性和分析的全面性。

4.2 人工智能支持的模糊测试

为了克服利用传统程序分析方法优化模糊测试时面临的挑战，越来越多的人工智能方法被使用到模糊测试的优化中。目前，随着并行计算能力的迅速提升和神经网络模型的不断进化，研究人员开始尝试使用神经网络模型来构建模糊测试反馈信息与各种模糊测试策略之间的映射关系。

一部分工作尝试使用深度学习方法优化模糊测试。NEUZZ^[30]利用神经网络拟合程序的行为，基于提取种子的覆盖反馈特征的神经网络预测变异字节变化对程序分支的影响。MTFuzz^[31]和PreFuzz^[32]则在NEUZZ的基础上进行了改进，分别引入了多任务训练和改进了神经网络对变异算子Havoc的指导。Rajpal等^[33]的研究讨论了各种神经网络模型在指导测试用例生成方面的性能。FuzzGuard^[34]则用神经网络预测定向模糊测试中种子是否能够到达目标。MEUZZ^[35]提取种子对应的特征并使用回归模型预测种子子代树的节点数，从而提高混合模糊测试的种子调度效率。Graphfuzz^[36]使用模糊测试历史经验训练图神经网络以预测种子在提高代码覆盖率方面的潜力。这部分工作使用深度学习模型代理传统的人工模型，以构建更复杂和智能的映射关系。但是这些工作使用的神经网络为预训练的模型，因此在泛化性方面仍然存在缺陷，导致其在新目标上的适应能力受限。

另一部分工作尝试使用强化学习方法优化模糊测试。Konstantin等^[37]将模糊测试种子变异建模为强化学习问题，其利用输入字符集的闭包定义强化学习模型的状态，利用程序基本块覆盖数量、执行路径长度和执行世界等指标定义奖励。该工作是利用强化学习模型优化模糊测试的早期探索，但是其缺乏具有说服力的评估以证明方法的有效性。Havoc_{MAB}^[25]将模糊测试变异算子的选择建模为多臂老虎机问题并使用UCB-1算法对其优化。但是其使用的模型较为简单且定义的奖励仅基于是否探索到新的代码分支，因此粒度

较粗, 优化的能力也有限。RainFuzz^[38]的方法与 Havoc_{MAB} 类似, 但是其定义了独特的“热图”以评估动作的奖励。AFLHier^[11]将模糊测试种子调度建模为多臂老虎机问题, 但是其奖励仅考虑了函数覆盖率和基本块覆盖率等控制流反馈, 没有考虑数据流反馈。此外, 该文原型系统收集的反馈依赖于动态二进制翻译器 (Qemu), 该方法严重限制了模糊测试器的吞吐量, 因此难以被用于工业界。考虑到该方法的吞吐量过低, 且本文方法使用了更高效的源代码插桩, 因此没有与其进行实验对比。上述方法定义的状态都较为简单, 且没有构建鲁棒的模型以将丰富的数据流和控制流反馈映射为奖励。这些方法难以观察到模糊测试中被测程序复杂的控制流和数据流特征。此外, 一些工作尝试利用深度强化学习模型提高定向模糊测试的性能, 并基于执行路径和目标节点之间的距离计算奖励, 其动作是选择或放弃某个样本^[39]。本文方法利用空间随机场及其分布特征定义种子在模糊测试策略下的状态, 并构建了深度神经网络以获取强化学习的近似 Q 值。本文方法观测了丰富的数据流和控制流统计特征, 且模型复杂度更高, 并以在线学习的方式运作, 具有适配各类被测程序的潜力。

5 讨论

本文利用随机场模型及其分布特征刻画模糊测试变异策略作用于被测程序产生的数据流特征。具体而言, 本文提出了过程间比较流图以表示各个比较指令之间的拓扑关系, 基于分支判断变量的取值定义随机变量, 并依此构建了随机场。该方法相较于已有方法可以获取模糊测试过程中更加全面和细粒度的数据流反馈特征。但是本文在构建过程间比较流图时没有考虑隐式控制流关系, 因此可能造成随机场部分信息的缺失。为了进一步提高本方法的准确性, 引入必要的隐式控制流分析方法是有益的, 这也是对本文方法进行进一步优化的可行方案之一。

此外, 本文使用了在线深度强化学习模型, 其具有可根据不同被测程序特征进行自适应优化的优点, 且不需要使用预先准备的训练集进行预训练。这在一些难以高效获得训练集标签的问题

上是一种有效的解决方案。但是由于智能体在线学习使用的数据有限, 因此准确性不足的问题是必然存在的, 这也会一定程度上影响本方法的鲁棒性。为了解决该问题, 一种可能的方案是结合预训练和在线学习两种智能体学习方案, 从一定程度上缓解使用单一在线学习方案存在的鲁棒性问题。然而, 该方案在某些场景中仍然面临预训练所用的数据集难以获取的问题, 但在训练集相对容易获取的情况下, 其是一种可行的解决方案。

6 结束语

本文提出了一种利用分布散度和深度强化学习模型对模糊测试进行自适应优化的方法。具体而言, 本文提出了一种利用过程间比较流图描述被测程序分支判断变量空间随机场的方法, 并使用蒙特卡洛方法获取变异策略产生的随机场分布特征。本文使用强化学习模型在线训练智能代理, 自适应调整模糊测试策略, 从而提高分支判断变量近似分布的分布散度。在代码覆盖能力方面, 在测试集 FuzzBench^[9]上长达 24 h 的重复实验结果表明, 与基线模糊测试器 AFL++^[19]相比, 本文提出的方法可以显著提高代码覆盖的速度 (中位数超过 2.5 倍), 并能提高总体代码覆盖; 与利用强化学习指导模糊测试的解决方案 Havoc_{MAB}^[25]相比, 本文提出的方法在覆盖速度和总体覆盖指标上均明显占优; 与工业界相关工作 CmpLog 相比, 本文提出的方法在大多数基准测试目标上更优。在漏洞检测能力方面, 基于测试集 Magma^[10]的实验结果表明, 与 AFL++、Havoc_{MAB} 和 CmpLog 相比, 本文提出的方法在一些基准测试目标上具备更强的漏洞触发能力。实验证明本文提出的方法是利用人工智能模型优化模糊测试的有效探索, 可以为后续基于强化学习优化模糊测试的研究提供参考。

参考文献:

- [1] BÖHME M, PHAM V T, ROYCHOUDHURY A. Coverage-based greybox fuzzing as markov chain[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 1032-1043.
- [2] RAWAT S, JAIN V, KUMAR A, et al. VUzzer: application-aware

- Evolutionary Fuzzing[C]//NDSS: 2017: 1-14.
- [3] GAN S, ZHANG C, QIN X, et al. Collafl: path sensitive fuzzing[C]//2018 IEEE Symposium on Security and Privacy (SP). 2018: 679-696.
- [4] 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.
- [5] 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). USENIX Association, 2020: 2307-2324.
- [6] MANTOVANI A, FIORALDI A, BALZAROTTI D. Fuzzing with data dependency information[C]//7th IEEE European Symposium on Security and Privacy (EuroS&P2022). 2022: 286-302.
- [7] ASCHERMANN C, SCHUMILO S, BLAZYTKO T, et al. REDQUEEN: fuzzing with input-to-state correspondence[C]//26th Annual Network and Distributed System Security Symposium (NDSS2019). 2019: 2019.
- [8] FIORALDI A, D'ELIA D C, COPPA E. WEIZZ: automatic greybox fuzzing for structured binary formats[C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020: 1-13.
- [9] METZMAN J, SZEKERES L, SIMON L, et al. Fuzzbench: an open fuzzer benchmarking platform and service[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 1393-1403.
- [10] HAZIMEH A, HERRERA A, PAYER M. Magma: A ground-truth fuzzing benchmark[C]//Proc. ACM Meas. Anal. Comput. Syst., 2020, 4(3).
- [11] WANG J, SONG C, YIN H. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing[C]//28th Annual Network and Distributed System Security Symposium(NDSS2021). 2021.
- [12] SHE D, SHAH A, JANA S. Effective seed scheduling for fuzzing with graph centrality analysis[C]//43rd IEEE Symposium on Security and Privacy(SP 2022). 2022: 2194-2211.
- [13] BÖHME M, MANÈS V J M, CHA S K. Boosting fuzzer efficiency: an information theoretic perspective[C]//28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 678-689.
- [14] ZALEWSKI M. American fuzzy lop, 2017[EB].
- [15] SCARSELLI F, GORI M, TSOI A C, et al. The graph neural network model[J]. IEEE Trans Neural Networks, 2009, 20(1): 61-80.
- [16] SUTTON R S, BARTO A G. Reinforcement learning - an introduction[M]. MIT Press, 1998.
- [17] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Human-level control through deep reinforcement learning[J]. Nature, 2015, 518: 529-533.
- [18] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Playing atari with deep reinforcement learning[J]. CoRR, 2013, abs/1312.5.
- [19] FIORALDI A, MAIER D C, EISSFELDT H, et al. AFL++: combining incremental steps of fuzzing research[C]//14th USENIX Workshop on Offensive Technologies, WOOT 2020. 2020.
- [20] LEE J, LEE I, KANG J. Self-attention graph pooling[C]//Proceedings of the 36th International Conference on Machine Learning. 2019: 3734-3743.
- [21] LATTECNER C, ADVE V. LLVM: a compilation framework for life-long program analysis & transformation[C]//International Symposium on Code Generation and Optimization(CGO 2004). 2004: 75-86.
- [22] HAGBERG A, SWART P, S CHULT D. Exploring network structure, dynamics, and function using NetworkX[R]. 2008.
- [23] PASZKE A, GROSS S, MASSA F, et al. PyTorch: an imperative style, high-performance deep learning library[M]//Proceedings of the 33rd International Conference on Neural Information Processing Systems. 2019.
- [24] WANG M, YU L, ZHENG D, et al. Deep graph library: towards efficient and scalable deep learning on graphs[J]. CoRR, 2019, abs/1909.01315.
- [25] WU M, JIANG L, XIANG J, et al. One fuzzing strategy to rule them all[C]//44th IEEE/ACM 44th International Conference on Software Engineering(ICSE 2022). 2022: 1634-1645.
- [26] GAN S, ZHANG C, CHEN P, et al. GREYONE: Data flow sensitive fuzzing[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2577-2594.
- [27] CHEN P, CHEN H. Angora: efficient fuzzing by principled search[C]//2018 IEEE Symposium on Security and Privacy(SP). 2018: 711-725.
- [28] STEPHENS N, GROSEN J, SALLS C, et al. Driller: augmenting fuzzing through selective symbolic execution. [C]//NDSS, 2016, 16: 1-16.
- [29] PENG H, SHOSHITAISHVILI Y, PAYER M. T-Fuzz: fuzzing by program transformation[C]//2018 IEEE Symposium on Security and Privacy (SP). 2018: 697-710.
- [30] SHE D, PEI K, EPSTEIN D, et al. Neuzz: efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). 2019: 803-817.
- [31] SHE D, KRISHNA R, YAN L, et al. MTFuzz: fuzzing with a multi-task neural network[C]//Association for Computing Machinery. 2020: 737 - 749.
- [32] WU M, JIANG L, XIANG J, et al. Evaluating and improving neural program-smoothing-based fuzzing[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 847-858.
- [33] RAJPAL M, BLUM W, SINGH R. Not all bytes are equal: Neural byte sieve for fuzzing[J]. arXiv preprint arXiv:1711.04596, 2017.
- [34] 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.
- [35] CHEN Y, AHMADI M, FARKHANI R M, et al. MEUZZ: Smart seed scheduling for hybrid fuzzing[C]//23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). 2020: 77-92.
- [36] XU H, CHEN L, GAN S, et al. Graphfuzz: data-driven seed scheduling for coverage-guided greybox fuzzing[J]. ACM Trans Softw Eng Methodol, 2024.

- [37] KONSTANTIN B, GODEFROID P, SINGH R. Deep reinforcement fuzzing[C]//2018 IEEE Security and Privacy Workshops (SPW). 2018: 116-122.
- [38] BINOSI L, RULLO L, POLINO M, et al. RainFuzz: reinforcement-learning driven heat-maps for boosting coverage-guided fuzzing [C]//Proceedings of the 12th International Conference on Pattern Recognition Applications and Methods(ICPRAM 2023). 2023: 39-50.
- [39] 肖天, 江智昊, 唐鹏, 等. 基于深度强化学习的高性能导向性模糊测试方案[J]. 网络与信息安全学报, 2023, 9(2): 132-142.
- XIAO T, JIANG Z H, TANG P, et al. High-performance directional fuzzing scheme based on deep reinforcement learning[J]. Chinese Journal of Network and Information Security, 2023, 9(2): 132-142.

[作者简介]



许航 (1996-), 男, 江苏扬州人, 网络空间安全教育部重点实验室博士生, 主要研究方向为智能漏洞挖掘、虚拟化漏洞挖掘。



计江安 (1999-), 男, 江西九江人, 网络空间安全教育部重点实验室博士生, 主要研究方向为嵌入式设备安全、二进制静态分析。



马哲宇 (1997-), 男, 河北秦皇岛人, 清华大学博士生, 主要研究方向为 Linux 内核安全、虚拟化安全。



张超 (1986-), 男, 湖北黄冈人, 博士, 清华大学副教授、博士生导师, 主要研究方向为软件安全、程序分析、人工智能安全、数据安全。