Evaluating and Improving Hybrid Fuzzing

摘要

迄今为止，人们提出了各种混合模糊器，通过结合模糊策略和符号执行器的优势，尽可能地探测出程序漏洞。虽然现有的混合模糊器已显示出优于传统的基于覆盖引导的模糊器，但它们很少采用相同的评估设置，如基准和种子库的初始化设置。因此，迫切需要对现有的混合模糊器进行全面研究，为该领域的未来研究提供启示和指导。为此，对最先进的混合模糊器进行了广泛研究。通过研究表明，现有混合模糊器的性能可能无法很好地推广到其他实验环境中。同时，与传统的覆盖引导模糊器相比，混合模糊器的性能优势总体有限。此外，不是简单地更新模糊策略或符号执行器，更新它们的结合模式可能会对混合模糊器的性能产生关键影响。因此，我们提出 CoFuzz，通过改进混合模糊器的结合模式来提高其有效性。具体来说，在基准混合模糊器 QSYM 的基础上，CoFuzz 采用面向边的调度方式，通过随机梯度下降的在线线性回归模型调度边，以应用符号执行. 它还采用采样增强同步技术，通过区间路径抽象和 John Walk 以及增量式更新模型，并为模糊测试提供种子。

关键词：混合模糊器；结合策略；面向边；路径抽象

1 引言

模糊测试通常是指自动生成测试输入，以暴露潜在的软件错误或安全漏洞。迄今为止，许多现有的模糊器通过优化被测程序的代码覆盖率来促进漏洞暴露（即覆盖引导的模糊测试。[2,4,5,12,13]。然而，在许多情况下，它们被证明是无效的 [1,3,11]。为解决这一问题，有人提出了混合模糊法 Hybrid Fuzzer，通过协调模糊策略和符号执行来提高模糊效果 [6–10,14,15]。但现有的 Hybrid Fuzzer 改进后的性能优势有限，不能很好地发挥 Hybrid Fuzzer 理论的性能优势，通过研究发现大部分 Hybrid Fuzzer 优化方向集中在符号执行 concolic 的相关优化上，一些调度 Schedule 和同步 Synchronization 的优化有限。针对这一情况，选题文章提出了 Hybrid Fuzzer 的结合模式框架，清晰地展示了 Fuzzer 和 Concolic Executor 通过调度 Schedule 和同步 Synchronization 模块实现结合，并在此基础上针对传统结合模式进行优化，优化了调度策略和同步策略，有效地提高了 Hybrid Fuzzer 对于代码的覆盖能力。

2 相关工作

本项目复现的结合模式的核心算法通过在线线性模型实现面向边的调度策略优化，并将调度得到的关键边及其相关种子进行符号执行操作，得到路径约束，通过 Solver 求解器将路

径约束转换为区间，在此基础上采样生成新的种子。优化传统 Hybrid Fuzzer 框架中模糊测试和符合执行边的冗余执行，既保证提高代码覆盖率，又有效地提高了 Hybrid Fuzzer 的执行效率。复现 Binutils 工具程序中 readelf 程序的实验测试结果，得到引起 readelf 程序崩溃的 crash。 [?]

2.1 调度算法

通过识别更具探索价值的边集，并结合机器学习的线性回归模型预测边的价值，来实现面向边的调度算法。

2.2 同步算法

通过抽象和算法采样来实现种子同步，实现在保留先前工作的变异基础上进行采样生成，提高同步效率，减少重复。

3 本文方法

3.1 本文方法概述

本文实现了如下图 1所示的 Hybrid Fuzzer 框架中图 2核心算法 Schedule 调度模块功能的代码编写，并利用 Adam 优化策略改进算法线性回归模型。同时结合路径抽象的 John Walk 采样实现模型的增量更新和种子同步。优化传统 Hybrid Fuzzer 框架中模糊测试和符合执行边的冗余执行，既保证提高代码覆盖率，又有效地提高了 Hybrid Fuzzer 的执行效率。复现 Binutils 工具程序中 readelf 程序的实验测试结果，得到引起 readelf 程序崩溃的 crash，并对比了与原文的边覆盖实验结果。

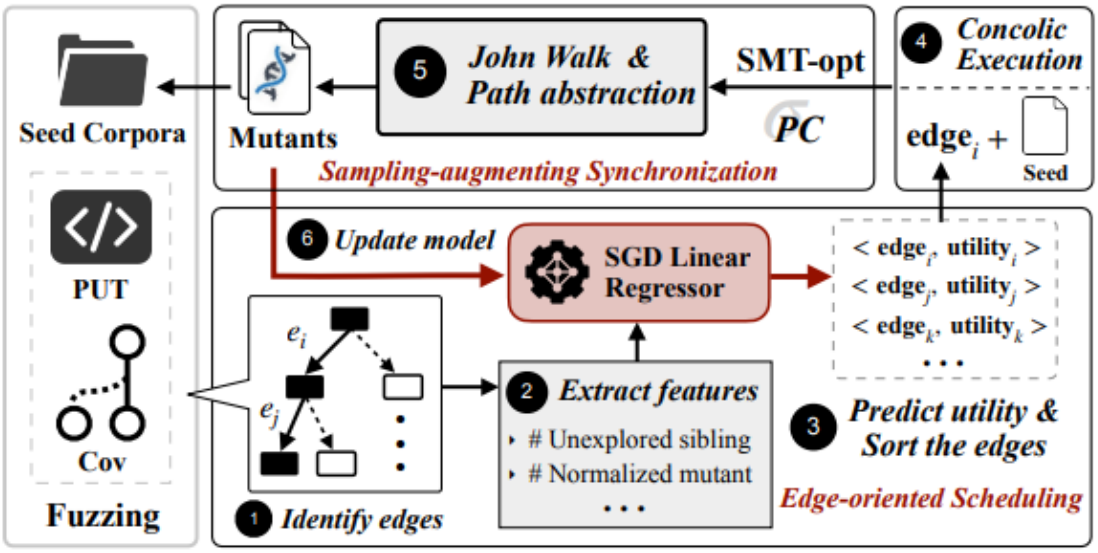

图 1. 方法示意图

Algorithm 1: Coordination Mode of CoFuzz

Input: *Model*
Result: *res*
Function *PerformingCoordinationMode*:
    res ← set()
    candidates ← Set of edges with unexplored sibling edges
    utility ← *Model.predict(candidates)* ; ▷ predict using the linear regression model with SGD
    critical_edges ← *edgeSchedule(candidates, utility)* ; ▷ schedule the edges with high utility
    for *all edge* e_i *in critical_edges* **do**
        s_i ← Identify the seed covering e_i
        pc ← *concolicExec*(s_i, e_i)
        $\hat{\varphi}$ ← *SMTopt(pc)*
        sample_set ← *JohnWalk*($s_i, \hat{\varphi}$)
        for *mutant in sample_set* **do**
            if *increaseCoverage(mutant)* **then**
                res.add(mutant)
        end
        cov_i ← Increased coverage
        Model.update(e_i, cov_i)
    end
    return *res*

图 2. 方法示意图

3.2 调度模块

本文首先通过跟踪 AFL 的种子执行路径，获取执行边情况，筛选出具有未探索同级边的边作为待调度边，使用线性回归模型预测边的优先级来实现面向边的调度。使用线性回归模型进行边预测的优势在于既能保证一定程度的边最优调度的精度，又能还满足测试所需要的在线实时要求。

3.3 特征提取

为了实现面向边的调度，在线线性回归模型的输入为边的特征值，输出为边的预测值。因此需要进行边的特征提取。针对已获取的边的信息，进行基于边到根的距离、边所属块的未探索同级边的数量、边所属块已变异次数、边所属块的条件约束分支类型、边所属块的条件约束操作数的位宽五个特征提取。边到根的距离，这个特征反映的是对应当前边到根的最短距离，代表了后续通过符号执行求解出结果的可能性，即距离越远，求解难度越大，越难得到有效解；边所属块的未探索同级边的数量，这个特征反映的是当前边对应种子具有的探索价值；边所属块已变异次数，这个特征反映的是当前已变异的次数，次数越高，后续变异优

先级相对越低；边所属块的条件约束分支类型，约束分支包括相等谓词，如 eq 或是 neq 的 cmp 工具，还有比较函数，如 memcmp 和 strcmp，代表了通过随机变异产生对应解的难度；边所属块的条件约束操作数的位宽，这个特征反映的是约束条件涉及操作数的位宽，代表了通过随机变异产生对应状态的难度。

3.4 同步模块

通过面向边调度得到的关键边，根据边调度获取边所对应的高优先级种子，进行后续种子的符号执行，并生成路径约束 pc，通过 SMT opt 算法将得到的路径约束 pc 转换为区间路径抽象，用于后续使用 John walk 在抽象域中采样生成突变体，通过这种形式保留有效信息，避免重复工作，同时保留了随机性。

4 复现细节

4.1 与已有开源代码对比

引用了原项目中同步机制模块代码，实现源程序 afl.py、concolic.py、sampler.py、sync.py。具体使用情况和实现工作如下：实现 trace.py 功能程序，针对 AFL 生成的种子，通过位图中基本快的形式进行面向边的调度处理操作，具体来说，就是跟踪执行 AFL 生成的种子，通过位图记录执行块信息，借此提取边的特征信息，如通过记录块的子节点来反映该块涉及边的未探索同级边情况；通过记录块所属种子，用于后续获取覆盖关键边的种子集；通过当前块的行数信息反映该基本块对应边到根的距离。实现采用了优化后调度和同步机制框架的执行器 executor.py 功能程序，基于 trace.py 识别出的待调度边集合，调用实现的调度模块，调度出高优先级的关键边，查询并返回覆盖了这些关键边的种子集进行 concolic 操作。executor 中引用了 afl.py、concolic.py、sampler.py、sync.py 的程序接口，实现了相关初始化配置以及后续 concolic 和采样操作。实现了调度模块中涉及模型操作的 depot.py 功能程序，程序实现功能：初始化候选关键边集合、通过模型预测得到候选关键边集合、获取覆盖关键边的种子集合、更新在线模型以提高模型准确性。

4.2 实验环境搭建

实验环境要求：Tested on Ubuntu 18.04/20.04 、Python (>= 3.8)、LLVM 10.0-12.0。本文实验环境基于 Docker 实现，通过 Dockerfile 构建实验基本运行环境镜像，并在对应容器下进行实验。

4.3 界面分析与使用说明

通过运行 AFL 和 CoFuzz 来实现改进后的 Hybrid Fuzzer 结合模式的运行使用。如下图]3为 AFL 运行界面，图4为 CoFuzz 运行界面。

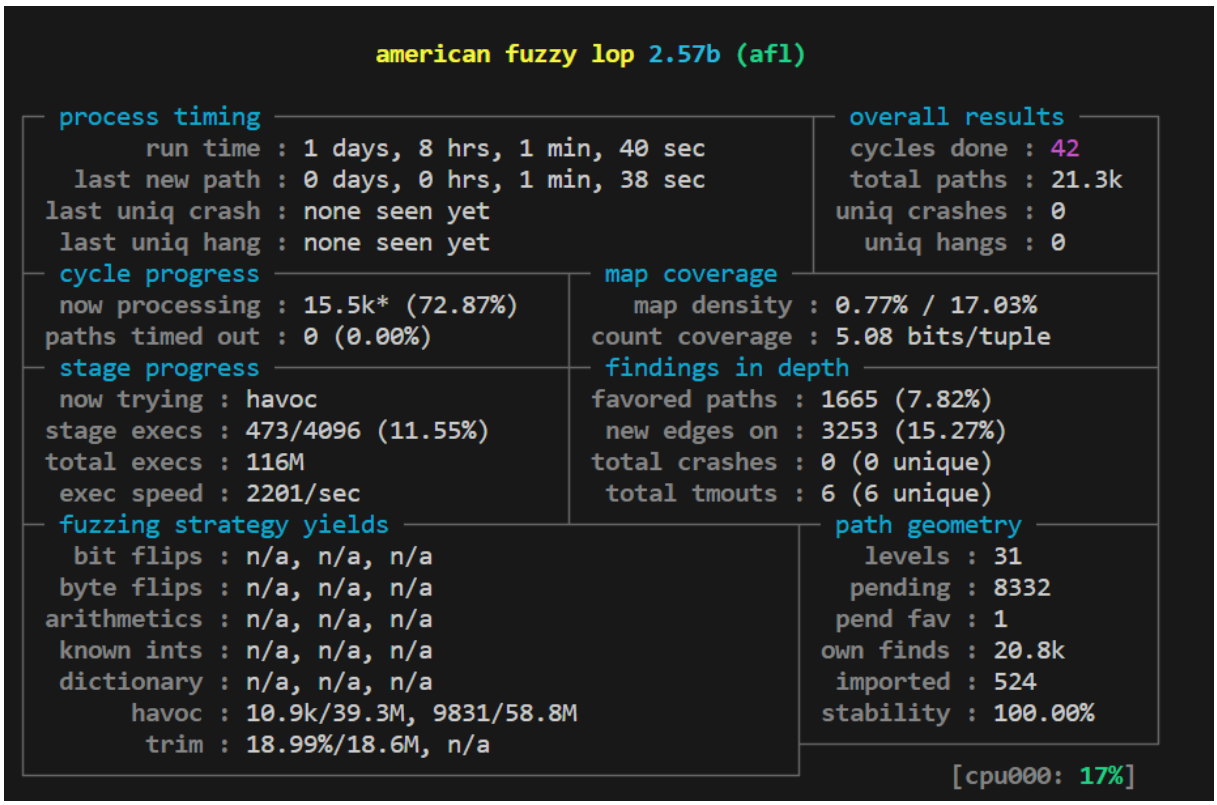

图 3. afl 操作界面示意

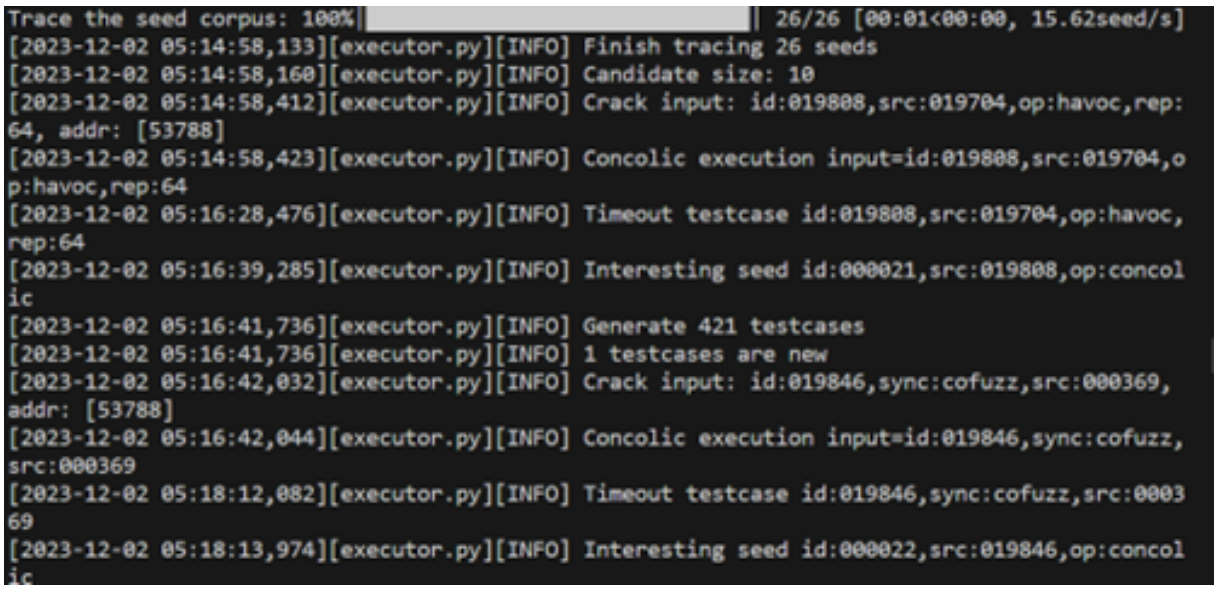

图 4. cofuzz 操作界面示意

5 实验结果分析

本文通过对 Binutils 工具程序中的 readelf 程序进行实验测试，实验结果如下图5所示。并通过实验，得到引起 readelf 程序崩溃相关的 crash，如下种子为 readelf 的 crash。

	Edges	Time
Paper	10786	24h
Reproduce	10253	104h

图 5. 实验结果示意

6 总结与展望

通过实验结果，进行分析后发现，实现和论文原文的边覆盖数所需要的时间是大约原文时间的 4 倍，通过对比发现是由于实验设备 cpu 核心数的情况导致的。通过复现，未来可以针对调度算法进行进一步优化，如实现更优的优化策略（原文中使用了 sgd），或是使用更加精准的预测模型进行边预测；除此之外，同步操作还有一步优化的空间。

参考文献

[1] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[3] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2020.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[6] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[7] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. {MEUZZ}: Smart seed scheduling for hybrid fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 77–92, 2020.

[8] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 515–530, 2019.

[9] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 736–747. IEEE, 2019.

[10] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1613–1627. IEEE, 2020.

[11] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[12] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–26, 2022.

[13] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.

[14] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[15] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.