Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

基于突变的模糊测试

硕士研究生 谢宁

2024年06月10日

问题回溯

- 总结反思
  - 控制演讲的语速，在演讲中引入适当的停顿，帮助听众更好地消化信息
  - 在解释复杂概念时，尽量使用简洁而清晰的语言，避免使用过于专业的术语

- 相关内容
  - 2024.05.26 邵思源《面向网络应用程序的模糊测试》
  - 2023.03.26 谢宁　《软件漏洞检测及其严重性评估》
  - 2023.02.12 邵思源《自动化漏洞挖掘初探》
  - 2021.03.28 刘力源《AFL–基于覆盖的模糊测试工具》

内容提要

- 预期收获
- 题目内涵解析
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - AMSFuzz
  - FA-Fuzz
- 特点总结与工作展望
- 参考文献

背景简介

- 预期收获
  - 掌握基于突变的模糊测试的基本概念
  - 了解拥有自适应算子选择能力的模糊方法
  - 了解基于萤火虫算法的模糊测试方法

内涵解析与研究目标　基于突变的模糊测试

- 研究目标
  - 程序的输入空间：包括程序的各种输入类型和格式（文件、网络数据包等）
  - 变异操作符：变异操作符可以是位翻转、字节替换、插入特定模式等
  - 变异种子：种子是用于变异的初始输入
  - 变异策略：包括确定性变异和不确定性变异。确定性变异按照预定义的顺序和规则进行变异，而不确定性变异则随机选择变异位置和操作符
- 题目内涵解析
  - 突变：指通过改变现有的种子来生成测试用例
    - AFL及其变体是基于突变的灰盒模糊器，它使用进化算法不断地变异种子来模糊目标程序，借此获取新路径
  - 模糊测试：通过向测试对象提供大量畸形测试用例作为输入，并监视其错误响应，以揭示潜在的异常缺陷及安全漏洞

研究背景

- 研究背景
  - 模糊测试的兴起：模糊测试作为一种自动化测试技术，通过生成随机或半随机的输入数据，能够有效发现未知漏洞
  - 模糊测试的局限性：模糊测试在漏洞发现方面具有显著优势，但其纯随机生成输入数据的方式往往难以触发深层次的程序逻辑错误
  - 基于突变的模糊测试提出：通过对已有输入数据进行突变操作，生成新的测试用例，基于突变的模糊测试旨在克服传统模糊测试的局限性
- 研究意义
  - 漏洞检测能力：有效地触发复杂的程序逻辑错误，帮助发现更多隐藏的安全漏洞
  - 测试资源利用：有针对性地生成覆盖未测试代码路径的输入数据
  - 测试技术发展：为自动化测试技术的发展提供了新的思路和方法

研究历史

2015 AFL是一种面向安全的模糊器，采用了一种新型的编译时工具以及遗传算法，大大提高了模糊代码的功能覆盖率

2017 Pham等人提出了有向灰盒模糊测试（DGF），目的是有效地到达一组给定的目标程序位置。开发并评估了一种基于模拟退火的功率计划

2018 Lemieux等人提出了Fair-fuzz，该方法自动识别由少数AFL生成的输入（稀有分支）执行的分支，且提出一种新颖的突变掩模创建算法

2019 Drozd 等人提出了 Fuzzer-gym，该方法使用这种状态信息来优化使用强化学习（RL）的突变算子。通过将 OpenAI-Gy 与 lib-Fuzzer 集成，实现对多个不同基准的更深入覆盖

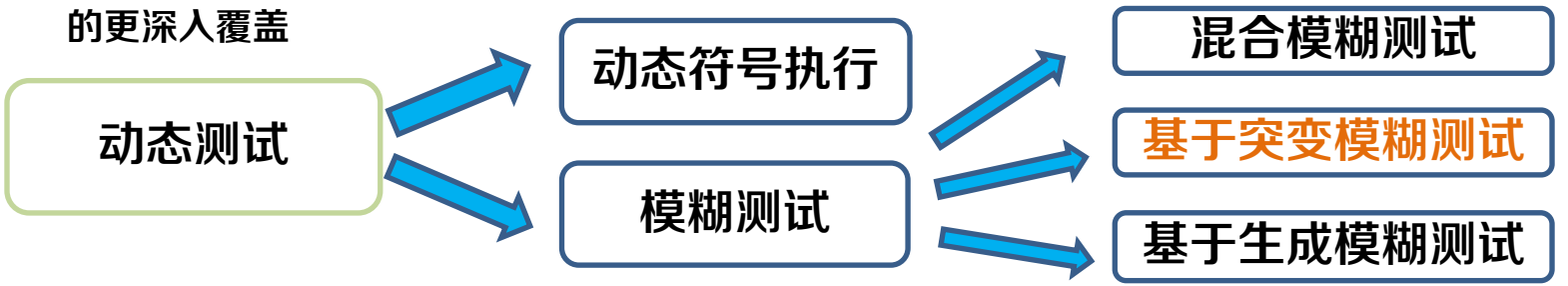
2020 Lv等人提出了MOPT，利用定制的粒子群优化（PSO）算法，找到算子在模糊测试有效性方面的最优选择概率分布

2022 zhang 等人提出了 Lib-Fuzz，允许研究人员和工程师扩展核心模糊器管道并共享他们的新组件以进行进一步评估

2023 Zhao等人提出了AMSFuzz，对突变算子的随机选择，AMS-Fuzz具有自适应调整突变算子概率分布来选择突变算子的能力；在模糊处理过程中对不同大小的种子进行动态切片，提高了模糊处理的效率

2024 Gao等人提出了FA-Fuzz，提出了一种新的通用调度方案FA-fuzz来求解变异算子的最优选择概率分布。基于萤火虫算法，提高了算法发现独特测试用例的效率

着重于算法方向研究



知识基础 AFL

- 基本概念
  - AFL（American Fuzzy Lop）是一种流行的模糊测试工具，用于自动化发现软件漏洞
- 运行方式
  - 通过生成大量随机输入数据（称为“种子”），并将其提供给目标程序运行，从而检测程序在异常输入下的行为
- 特点
  - 基于覆盖率的反馈：使用代码覆盖率作为反馈机制，以最大化覆盖新的代码路径
  - 易用性：AFL 具有简单的命令行界面，易于集成和使用

Algorithm 1: AFL Algorithm.

Input: Seed input: *seeds*, Instrumented target program: P
Output: Seed queue: Q, Crash set: C

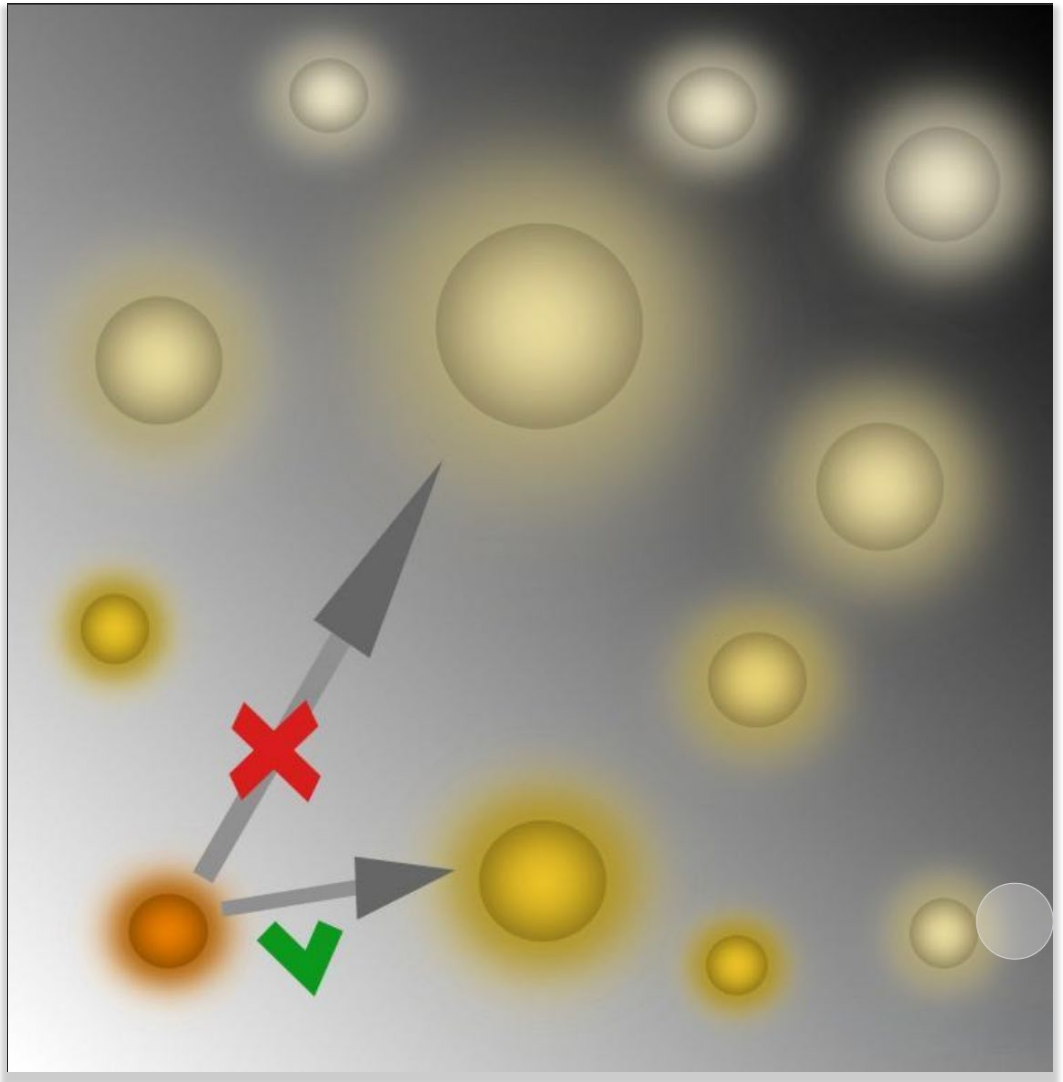
```
1  Q ← seeds
2  C ← ∅
3  while TRUE do
4      s ← ChooseNext(Q)
5      e ← AssignEnergy(s)
6      for op in deterministicOps do
7          for i in Range(0, s.length) do
8              s' ← Mutation(s, op, i)
9              status ← Run_Target(P, s')
10             Save_Update(status)
11     for i in Range(0, e) do
12         op ← Random(non−deterministicOps)
13         s' ← Mutation(op)
14         status ← Run_Target(P, s')
15         Save_Update(status)
16 Procedure Save_Update(status)
17 if is_Crash(status) then
18     C ← C ∪ s'
19     return
20 if is_NewCoverage(status) then
21     Q ← Q ∪ s'
22     return
```

知识基础　变异阶段

- 确定性阶段
  - 顺序变异：种子从头到尾逐位进行变异
  - 变异操作符：在确定性阶段使用多种操作符进行变异
    - 位翻转：通过翻转特定位数来变异种子
    - 算术操作：对特定位置进行整数加减操作
    - 有趣值替换：使用预定义的有趣值替换种子中的某些部分
    - 字典操作：使用用户提供的或自动生成的标记来覆盖或插入到种子中
- 非确定性阶段
  - 混乱阶段：变异位置和变异操作符是随机选择的
    - 通过这种随机选择，测试能够覆盖更多的变异组合
    - 变异操作符与确定性阶段类似，但其选择是随机的
  - 拼接：拼接是将两个种子文件切成两部分，然后将它们的头部和尾部拼接起来

知识基础　萤火虫算法(FireFly)

- 基本内涵
  - 模仿了萤火虫之间通过闪光进行交流和吸引的行为
    - 吸引力：吸引力取决于它们的亮度
    - 移动：会向亮度更高的萤火虫移动
- 算法步骤
  - 初始化：随机生成一组萤火虫，并计算它们的亮度（目标函数值）
  - 亮度更新：根据目标函数值更新每只萤火虫的亮度
  - 位置更新：每只萤火虫根据吸引力向亮度更高的萤火虫移动



AMSFuzz: An adaptive mutation schedule for fuzzing

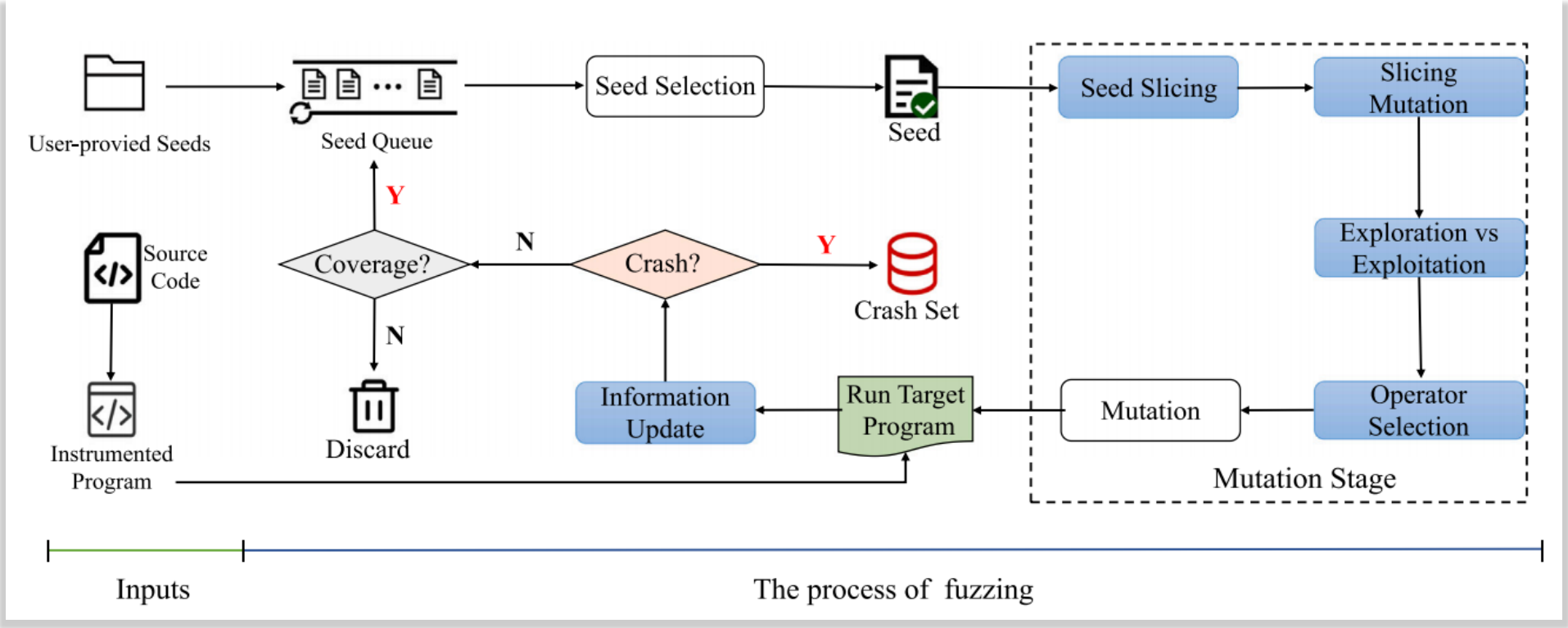
AMSFuzz TIPO

T	目标	实现一个具有自适应突变算子选择能力的模糊测试方法
I	输入	有效格式的测试用例*N
P	处理	1. 从用例序列中获取种子 2. 对变异种子进行切片，并对切片区域进行变异 3. 确定变异方案，并选择变异操作符 4. 利用变异后的用例测试待测程序
O	输出	待测程序的状态*M

P	问题	现有的模糊器中，突变算子的随机选择和突变位置的顺序选择影响了路径发现和漏洞检测
C	条件	程序必须无状态且程序的路径和崩溃数量有限
D	难点	减少由于固定的变异方案导致的变异效率的损失
L	水平	Expert Systems with Applications（SCI 一区 2023）

AMSFuzz 算法原理图

- 算法原理图
  - 种子切片：基于平均变异次数来动态分配切片大小，并使用种子偏移量将种子分割为变异完成区域和未变异区域
  - 自适应操作符选择：选择变异方案，并根据变异结果和方案类型确定操作符



创新分析　自适应操作符选择

- 建立MAB模型
  - 目标：选择一个使路径数量最大化的变异算子
  - 模型属性
    - Arms: AMSFuzz为每个操作符选择一个Arm，并试图找到最优操作符
    - Action: Action表示程序立即被模糊化(即，种子突变和程序执行)
    - Rewards: Rewards是指目标程序被模糊多次后发现的路径数量，用R表示
    - Exploration vs Exploitation:指为了获得更高奖励的待选执行方案

Algorithm 2: Operator Selection

Input: Number of mutation operators: n, Probability of mutation operators: $Prob[n]$, Mutation operators: $ops[n]$, Stage:$stage$

Output: A selected mutation operator op

```
1  if stage is exploration then
2      i ← Random(0, n)
3      op ← ops[i]
4  else
5      for i in Range(0, n) do
6          if i ≠ 0 then
7              Pt[i] ← Pt[i − 1] + Prob[i]
8          else
9              Pt[i] ← Prob[i]
10     rand ← Random(0, 100)
11     for i in Range(0, n) do
12         if rand < Pt[i] × 100 then
13             op ← ops[i]
14             break
```

依据CWE abstract类型，划分出多个更平衡的子分布

创新分析　自适应操作符选择

- 基于概率分布的操作符选择
  - 原因：不同的变异算子产生新的不确定路径
    - 不能保证获得最多奖励的突变操作符下一次一定产生新路径
  - 原理
    - 基于累积概率进行算子选择
  - 步骤
    - 概率分布建立：在Exploitation阶段，将所有突变算子的概率映射到一个连续空间
    - 突变算子选择：生成一个随机数，根据随机数来选择突变算子

Algorithm 2: Operator Selection

Input: Number of mutation operators: n, Probability of mutation operators: $Prob[n]$, Mutation operators: $ops[n]$, Stage:$stage$

Output: A selected mutation operator op

```
if stage is exploration then
    i ← Random(0, n)
    op ← ops[i]
else
    for i in Range(0, n) do
        if i ≠ 0 then
            Pt[i] ← Pt[i − 1] + Prob[i]
        else
            Pt[i] ← Prob[i]
    rand ← Random(0, 100)
    for i in Range(0, n) do
        if rand < Pt[i] × 100 then
            op ← ops[i]
            break
```

- 参数更新
  - 每次程序执行后，都会更新一组信息，以指导下一步选择突变操作符
  - 变异操作符的命中计数：在程序被模糊$(t+1)$次之后，突变op_i 的命中数计算

$$C_{op_i}^{t+1}\begin{cases} C_{op_i}^{t} + 1 & select_{op_i} = true \\ C_{op_i}^{t} & select_{op_i} = false \end{cases}$$

  - 突变操作符的奖励：在对程序被模糊$(t+1)$次之后，突变操作符op_i的奖励计算

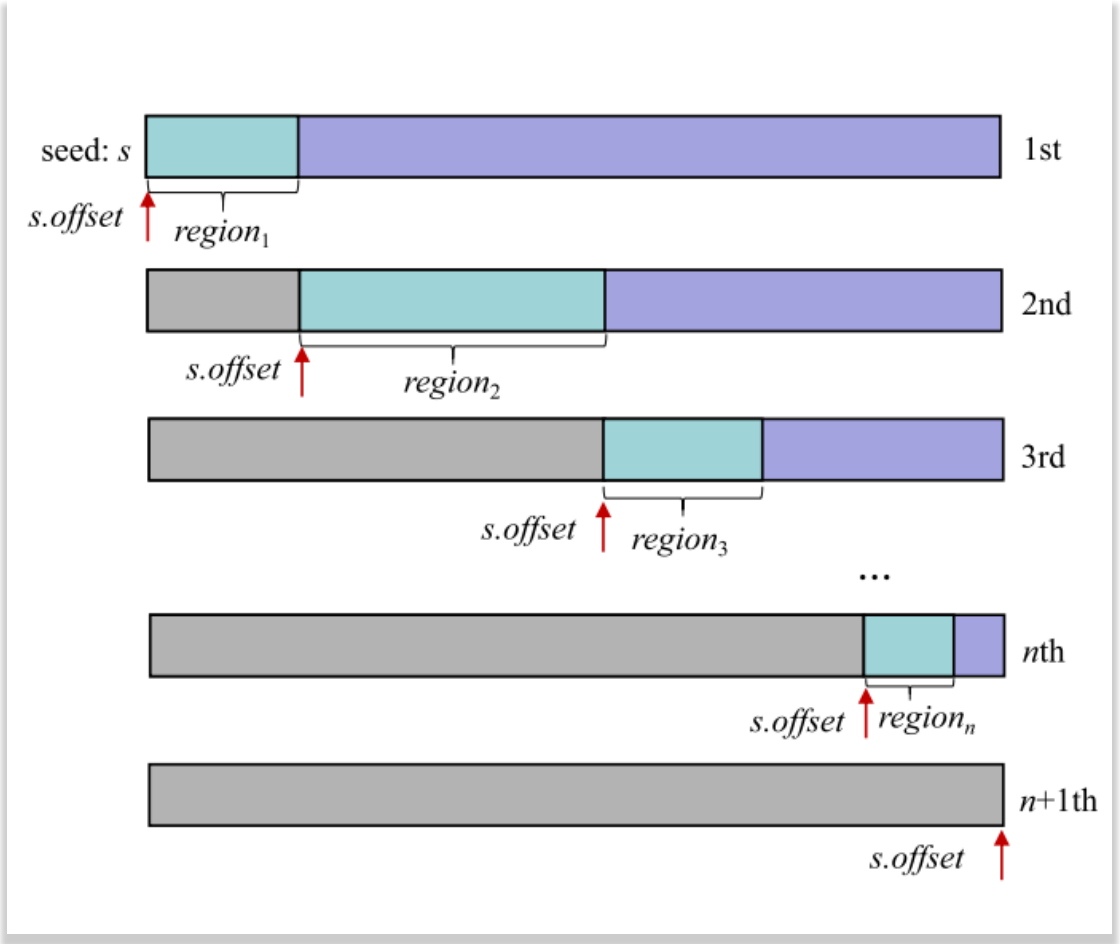
$$R_{op_i}^{t+1}\begin{cases} R_{op_i}^{t} + r_{op_i}^{t+1} & found_{op_i} = true \\ R_{op_i}^{t} & found_{op_i} = false \end{cases}$$

  - 变异算子的有效性：更新上述信息后，突变算子的有效性计算

$$E_{R_{op_i}}^{t+1}\begin{cases} \dfrac{R_{op_i}^{t+1}}{C_{op_i}^{t+1}} & C_{op_i}^{t+1} \neq 0 \\ 0 & others \end{cases} \qquad P_{R_{op_i}}^{t+1}\begin{cases} \dfrac{E_{op_i}^{t+1}}{\sum_{i=1}^{K} E_{opi}^{t+1}} & \sum_{i=1}^{K} E_{op_i}^{t+1} \neq 0 \\ 0 & others \end{cases}$$

创新分析　种子切片

- 种子切片
  - 目标：实现种子突变区域的动态分配，使更多的种子突变几率上升
  - 原因
    - AFL采用序列突变，使得数量受种子长度的影响
    - 整个模糊过程，路径搜索是动态变化的，随着时间的变化，路径的搜索空间逐渐缩小
    - 序列突变使用固定的突变长度，没有考虑路径搜索空间的影响



分块使用，择优选择

- 切片大小选择
  - 平均突变数：程序模糊t次后，计算模糊过程中生成新路径M_{avg}^{t}的平均突变次数

$$M_{avg}^{t}\begin{cases}\dfrac{M_{total}^{t}}{S_{total}^{t}} & S_{total}^{t} \neq 0 \\ M_{total}^{t} & others\end{cases}$$

  - 预切片大小：根据最后一次突变的结果，将程序模糊t次后，计算出种子s的预切片大小$pslice_{s}^{t}$

$$pslice_{s}^{t}=\begin{cases}\dfrac{M_{avg}^{t}}{n} & fuzz_{s}=false \\ Min\left(lslice_{s},\alpha\cdot\dfrac{M_{avg}^{t}}{n}\right) & lastf_{s}=fuzz_{s}=true \\ Max\left(\beta\cdot lslice_{s},\dfrac{M_{avg}^{t}}{n}\right) & lastf_{s}=false\&fuzz_{s}=true\end{cases}$$

  - 切片大小：将程序模糊t次后，种子s的大小$slice_{s}^{t}$

$$slice_{s}^{t}\begin{cases}pslice_{s}^{t} & pslice_{s}^{t} < unmuta_{s}^{t} \\ unmuta_{s}^{t} & others\end{cases}$$

创新分析　种子切片

- 切片变异
  - 初始化
    - 提供种子输入seeds和插桩后的程序P
  - 变异和执行
    - 计算出的切片大小和种子s的偏移量s.offset确定变异区域
    - 对程序进行变异并执行
  - 处理变异结果
    - 结果导致崩溃，保存到崩溃集合C
    - 结果产生新路径，保存到种子队列Q
  - 参数更新
    - 更新种子的偏移量s.offset
    - 根据偏移量标记种子

Algorithm 3: Slicing Mutation

Input: Seed inputs: *seeds*, Instrumented target program: *P*
Output: Seed queue: *Q*, Crash set: *C*

```
1  Q ← seeds
2  C ← ∅
3  while TRUE do
4      s ← ChooseNext(Q)
5      s.slice ← AssignSlice(s)
6      for op in deterministicOps do
7          for i in Range(s.offset, s.offset + s.slice) do
8              s' ← Mutation(s, op, i)
9              status ← Run_Target(P, s')
10             if is_Crash(status) then
11                 C ← C ∪ s'
12                 continue
13             if is_NewCoverage(status) then
14                 Q ← Q ∪ s'
15     s.offset ← s.offset + s.slice
16     if s.offset = s.length then
17         s.was_sliced ← true
```

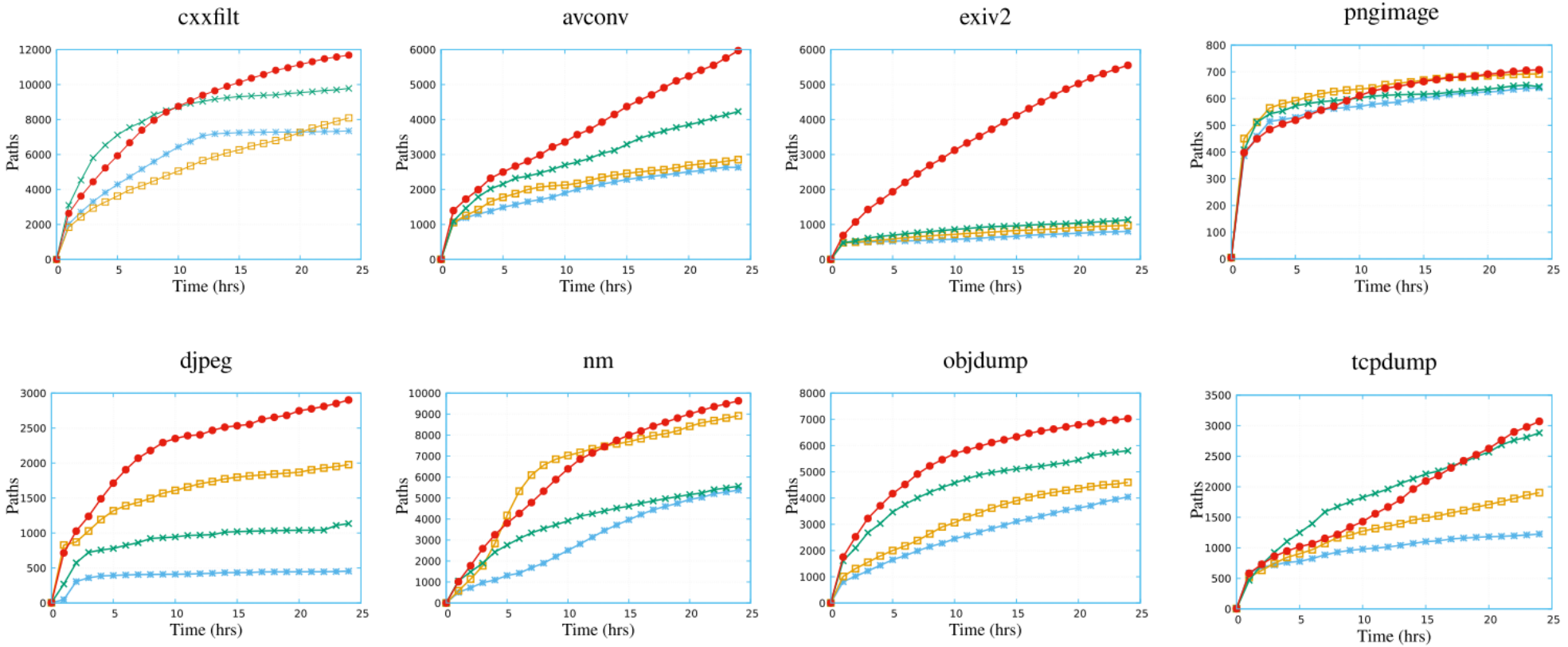
实验设计　数据资源

- 基线fuzzers
  - 流行的fuzzers工具AFL、AFLFast、FairFuzz和MOPT
- 候选程序
  - 开发工具：nm、cxxfilt、size
  - 图像处理程序：tiff2pdf、exiv2
  - 音视频处理工具：avconv
  - 包处理程序：tcpdump
- 初始种子
  - 随机收集有效格式的测试用例
- 评价指标
  - 覆盖率（Line、Func、Bran）

Program	Command line	Project version	Format
Cxxfilt	C++filt – t	GNU Binutils 2.31	ELF
Nm	Nm – C @@	GNU Binutils 2.31	ELF
Size	Size @@	GNU Binutils 2.31	ELF
Objdump	Objdump -d @@	GNU Binutils 2.31	ELF
Strings	Strings -a -d -f @@	GNU Binutils 2.31	ELF
Tiff2pdf	Tiff2pdf @@	Libtiff 4.2.0	TIFF
Exiv2	Exiv2 @@	Eciv 1.0.0.9	JPG
Pngimage	Pngimage @@	Libpng 1.6.35	PNG
Djpeg	Djpeg @@	libjpeg-turbo 2.1.1	JPEG
avconv	avconv -i @@ -f null -	libav 12.3	AVI
yara	yara @@	Yara 3.6.0	YARA
tcpdump	tcpdump -n -r @@ -v	tcpdump 5.0.0	PCAP

实验设计　对比实验

- 实验结果
  - 路径发现受到搜索空间的限制，随着发现的路径增加，路径的搜索空间变小
  - 突变位置的选择影响路径发现，确定性阶段花费较多时间，降低路径发现速度



- 实验结果
  - 与AFL、AFLFast和FairFuzz相比，AMSFuzz实现了更好的行、功能和分支覆盖
  - AMSFuzz在exiv2中实现了14.3%的线路覆盖率，而其他fuzzer的线路覆盖率不超过10%

Program	AFL			AFLFast			FairFuzz			AMSFuzz		
	Line	Func	Bran	Line	Func	Bran	Line	Func	Bran	Line	Func	Bran
Cxxfilt	2.70%	**3.10%**	**2.40%**	2.70%	**3.10%**	**2.40%**	**2.80%**	**3.10%**	**2.40%**	**2.80%**	**3.10%**	**2.40%**
Nm	4.80%	7.10%	4.00%	4.80%	7.00%	4.00%	4.40%	6.80%	3.70%	**5.00%**	**7.40%**	**4.10%**
Size	**2.50%**	**4.10%**	**1.90%**	2.10%	3.70%	1.70%	2.30%	3.90%	**1.90%**	**2.50%**	**4.10%**	**1.90%**
Objdump	4.30%	6.20%	3.60%	2.80%	4.90%	2.20%	4.60%	6.50%	**3.80%**	**4.70%**	**6.80%**	**3.80%**
Strings	2.20%	**3.70%**	**1.70%**	1.80%	3.20%	1.40%	2.00%	3.40%	1.60%	**2.30%**	**3.70%**	**1.70%**
Tiff2pdf	11.20%	16.90%	8.40%	11.40%	17.20%	8.50%	10.90%	17.10%	8.20%	**21.00%**	**30.50%**	**17.70%**
Exiv2	8.60%	13.10%	3.70%	8.40%	12.70%	3.50%	9.80%	14.30%	4.10%	**14.30%**	**21.20%**	**6.50%**
Pngimage	13.20%	19.50%	10.90%	14.00%	19.90%	11.60%	**14.10%**	19.90%	**11.80%**	14.00%	**20.00%**	11.70%
Djpeg	13.70%	19.40%	8.80%	17.90%	22.20%	14.40%	16.90%	21.50%	12.20%	**18.20%**	**22.40%**	**14.50%**
avconv	8.90%	11.50%	6.40%	9.00%	11.60%	6.50%	9.30%	11.80%	6.70%	**10.90%**	**13.30%**	**7.90%**
yara	32.00%	34.10%	17.40%	32.70%	34.40%	18.00%	**33.40%**	**34.70%**	**18.60%**	33.20%	34.20%	18.40%
tcpdump	13.10%	25.90%	9.30%	18.10%	31.80%	13.60%	22.30%	35.00%	17.30%	**22.50%**	34.60%	**17.70%**

实验设计　消融实验

- 实验结果
  - 分组方法
    - AMS.bandit：集成自适应算子选择功能
    - AMS.slicing：集成种子切片功能
  - 自适应算子选择
    - 从实验结果可以看出，所提出的自适应算子选择在Havoc阶段是有效的，能够在24小时内发现更多路径
  - 种子切片
    - 切片在路径发现方面较优
    - MOPT在cxxfilt上发现的路径比AFL低
    - MOPT在pngimage发现的路径比AFL高

本方法的所有模块均有良好效果

Program	AMS.bandit	MOPT-AFL	AFL-UCB	AMS-TS
Cxxfilt	**13387**	13140	12025	12056
Nm	**12211**	11761	9421	10046
Size	3086	**3191**	2863	3084
Objdump	**8114**	7828	6202	7353
Strings	**3430**	3303	2806	3190
Tiff2pdf	6574	**7364**	5207	6065
Exiv2	8926	3946	3628	**9208**
Pngimage	**713**	672	663	712
Djpeg	3487	**3613**	3291	3420
avconv	**12084**	9627	7793	12034
yara	**2592**	2400	2306	2478
tcpdump	7144	5656	6165	**7742**

Program	AMS.slicing	MOPT-AFL	AFL
Cxxfilt	**11342**	7041	7353
Nm	**9401**	3955	5374
Size	**3412**	3050	2359
Objdump	**6786**	3891	4044
Strings	**3445**	2720	2190
Tiff2pdf	**6836**	5763	1418
Exiv2	**5678**	2753	803
Pngimage	770	**803**	640
Djpeg	**3323**	2638	455
avconv	**6924**	5712	2633
yara	**2362**	1545	1219
tcpdump	4409	**4616**	1226

FA-Fuzz: A Novel Scheduling Scheme Using Firefly Algorithm for Mutation-Based Fuzzing

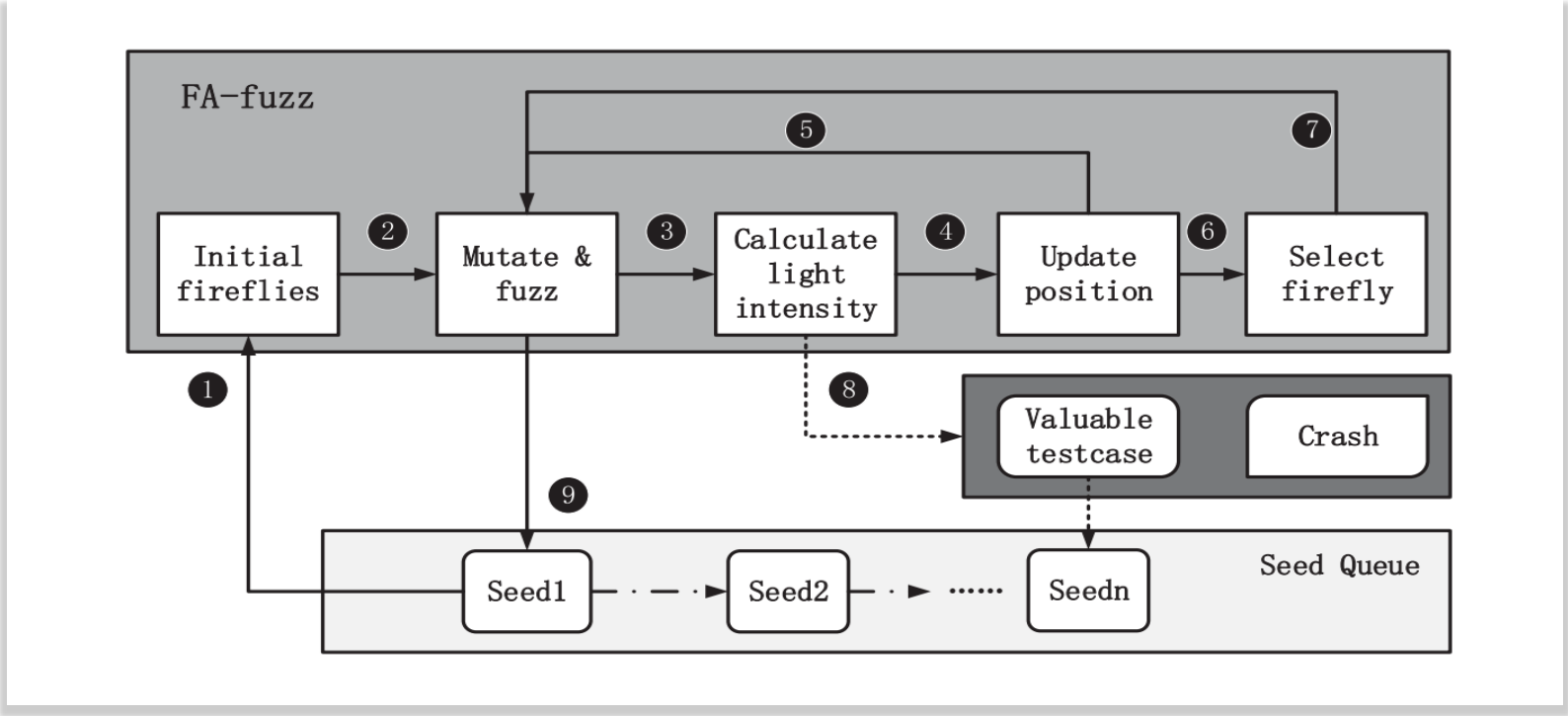
FA-Fuzz TIPO

T	目标	实现一种模糊测试中的通用调度方法
I	输入	有效格式的测试用例*N
P	处理	1. 初始化：初始化种子序列和算法参数 2. 光强计算：使用寻找有趣测试用例的效率作为萤火虫的光强 3. 位置更新：根据标准萤火虫算法进行位置更新 4. 种子选择：经过数次迭代后，保留最优分布
O	输出	种子的最优概率分布*M

P	问题	现有方法没有考虑到不同种子的最优算子分布是不同的
C	条件	程序必须无状态且程序的路径和崩溃数量有限
D	难点	寻找到不同情况下种子的最优算子分布
L	水平	IEEE Transactions on Software Engineering（SCI 一区 2024）

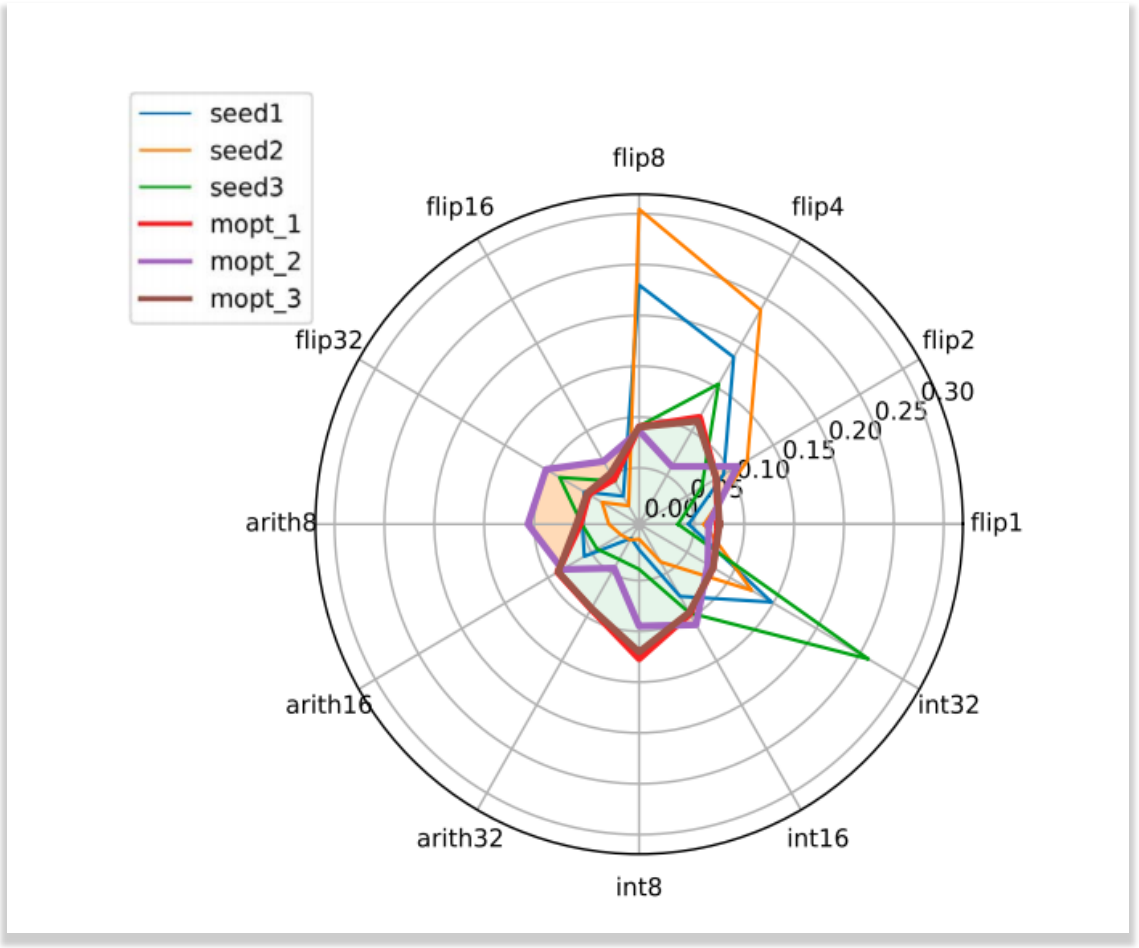
FA-Fuzz 算法原理图

- 算法原理图
  - 参数初始化：预先设定fireflies算法所需的参数，如数量、最大迭代次数等
  - 模糊测试：根据fireflies的位置变异测试用例并进行模糊测试
  - 参数更新：计算光强度来更新fireflies的位置，并继续用新的位置进行模糊测试



创新分析 FA-Fuzz

- 参数初始化
  - 对应关系：将FA-fuzz中的萤火虫视为突变算子的概率分布
  - 取值范围：萤火虫在每个维度的分量位置值都在(0,1)区间内
- 光强计算
  - 表示方式：$I(x) \propto f(x)$
  - 计算方式：有效的测试用例的数量除以在一个Havoc阶段产生的总测试用例数量
  - 必要条件：增加Havoc阶段生成的测试用例数量的下限



- 位置更新
  - 萤火虫的位置运动计算方式如下

$$x_i^{t+1} = x_i^t + \beta_0 e^{-\gamma r_{ij}^2}(x_j^t - x_i^t) + \alpha\epsilon_i^t$$

  - 吸引力β是一个相对值，它会随着萤火虫i与萤火虫j之间的距离r_{ij}而变化
  - 设计变量步长α，其中α_0为初始随机化参数，$\theta \in (0,1)$为随机下降

$$\alpha = \alpha_0 \theta^t$$

  - 位置更新后，所有变异算子的选择概率之和可能不等于1，进行式归一化计算

$$x_{ik}^t = x_{ik}^t / \sum_{k=1}^{n} x_{ik}^t$$

- 萤火虫选择
  - 依据：在多个位置中，选择亮度最强的位置作为最终的优化结果

创新分析 FA-Fuzz

- FA-Fuzz算法
  - 初始化
    - 随机分配多个萤火虫的位置
  - 迭代过程
    - 根据萤火虫的概率分布选择变异操作符，结合有趣测试用例计算亮度
  - 位置更新
    - 进行亮度比较，向强光移动，如果已经是最亮的，则提前结束该轮迭代
  - 利用最佳概率
    - 对于不是第一次遇到的种子，直接使用已收敛最佳概率变异

Algorithm 2 FA-fuzz algorithm

```
1:  procedure FA-FUZZLOOP(sds, prog, fnum)
2:      q ← sds
3:      nq ← q
4:      while true do
5:          for sd ∈ nq do
6:              FireflyMutation(sd, prog, fnum, nq)
7:          if ¬Findnewpath(q, nq) then
8:              Splice(sd, prog, nq)
9:          q ← nq
10: procedure FIREFLYMUTATION(sd, prog, fnum, nq)
11:     score ← Performscore(sd, prog)
12:     if Is_first_time(sd) then
13:         iterationnum ← 0
14:         Initspos(fnum)
15:         while iteration_num < iteration_max do
16:             for firefly_i ∈ fnum do
17:                 Havoc(sd, prog, nq, score, f_i)
18:                 intensity_i ← CalcFit(f_i)
19:                 inner_loop_end ← true
20:                 for f_j ∈ fnum do
21:                     Havoc(sd, prog, nq, score, f_j)
22:                     intensity_j ← CalcFit(f_j)
23:                     if intensity_i < intensity_j then
24:                         inner_loop_end ← false
25:                         Updatepos(f_i, f_j)
26:                         Havoc(sd, prog, nq, score, f_j)
27:                         intensity_j ← CalcFit(f_j)
28:                 if inner_loop_end = true then
29:                     break
30:             iteration_num ← iteration_num + 1
31:     else
32:         best_f ← Getbestfirefly(f_num)
33:         Havoc(sd, prog, nq, score, best_f)
```

实验设计　数据资源

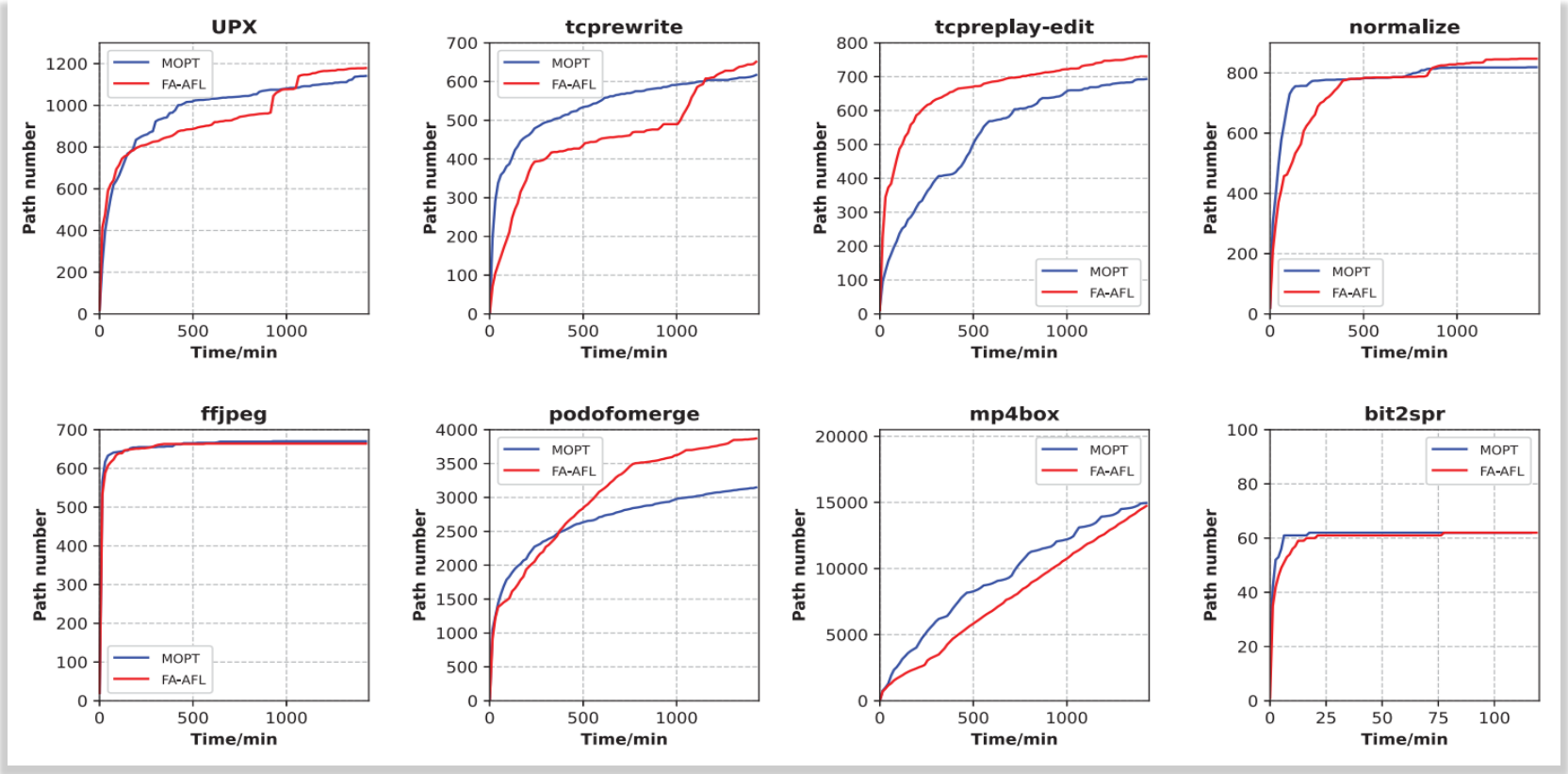
- 数据集
  - 来源：真实世界的程序和FuzzBench
  - 程序类型：图像格式转换器、音频处理器、视频处理器、比特流转换器、网络流量重放和编辑器、加密shell软件等
- 基线方法
  - AFL
    - 使用默认配置进行实验
  - MOPT
    - 目前最好的突变算子优化方法
- 评价指标
  - Unique Paths：衡量漏洞发现能力
  - Unique bugs：衡量潜在漏洞挖掘能力

No.	Software	Version	Arguments
1	bit2spr	beta	@@ /dev/null
2	UPX	0.8.0	-df @@ -o /dev/null
3	Normalize	0.7.7	@@ /dev/null
4	tcpreplay-edit	4.3.2	-r 80:84 -s 20 -b -C…
5	tcprewrite	4.3.2	-i @@ -o /dev/null
6	podofomerge	0.9.6	@@ @@ out.pdf
7	Mp4box	0.8.0	Hint @@
8	ffjpeg	Master branch	-d @@

- 实验结果-真实场景
  - 进行了10轮独立测试，并使用中位数来计算独特路径和独特错误的数量
  - 在寻找独特漏洞和独特路径方面，FA-AFL比最先进的模糊器MOPT更有效
  - bit2spr是一个较小的程序，没有足够的状态空间可以探索

Program	AFL		MOPT		FA-AFL	
	Unique Paths	Unique Bugs	Unique Paths	Unique Bugs	Unique Paths	Unique Bugs
bit2spr	59	1	62	1	**62**	**1**
UPX	20	0	1150	7	**1171**	**8**
Normalize	318	1	825	2	**839**	**2**
tcpreplay-edit	632	25	696	2	**732**	**2**
tcprewrite	339	4	609	2	**649**	**2**
Mp4box	1580	185	**15045**	9	14396	**9**
ffjpeg	445	24	**669**	2	666	**2**
podofomerge	1547	7	3169	3	**3926**	**8**

- 实验结果-真实场景
  - 在早期阶段生成大量有趣的测试用例时，这些测试用例被添加到种子队列中
    - FA-Fuzz需要一段时间来探索这些种子的最佳概率分布



实验设计　对比实验

- 实验结果-FuzzBench
  - 测试目标：在不同领域广泛使用和经过良好测试的程序
  - 指标：覆盖率中值
    - 内涵：程序的相对代码覆盖率的中值
    - 计算方式

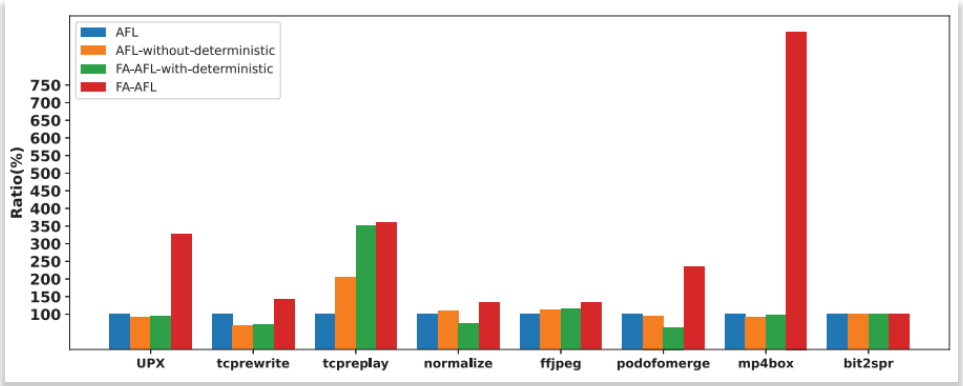
$$relative\_cov = \frac{cov}{max\_cov}$$

  - 结论
    - 与现有的工作相比，AFLplusplus_fa在一些基准测试中有一定的改进
    - 代码覆盖率是衡量模糊器的一个很好的指标，但不是唯一的指标

Program	AFLpp	AFLpp_fa	Mopt	AFL	FA-AFL
Curl-curl	**98**	97	94	95	93
Freetype2	**96**	95	67	67	67
harfbuzz	**99**	99	97	97	97
jsoncpp	**100**	**100**	99	99	99
libjpeg	**99**	99	99	99	99
mbedtls	97	**99**	96	96	95
openssl	99	**99**	99	99	99
Re2	99	**99**	99	99	99
Sqlite2	**98**	92	94	94	93
System	99	**99**	91	91	91
Vorbis	**99**	99	98	98	98
Woff2	97	**99**	95	94	94
zlib	97	**99**	97	96	96

Program	AFLpp	Mopt	AFL	FA-AFL
Curl-curl	0.22	**1**	**1**	**1**
Freetype2	0.4	**1**	**1**	**1**
harfbuzz	0.26	**0.92**	**1**	**1**
jsoncpp	NAN	**0.8**	**1**	**0.8**
libjpeg	0.48	**0.84**	**0.7**	**0.72**
mbedtls	**0.92**	**1**	**1**	**1**
openssl	**0.8**	**0.8**	**0.78**	**0.82**
Re2	**0.54**	**1**	**0.8**	**1**
Sqlite2	0	0.36	0.24	**0.52**
System	**0.54**	**1**	**1**	**1**
Vorbis	0.26	**1**	**1**	**1**
Woff2	**0.82**	**1**	**1**	**1**
zlib	**0.74**	**0.96**	**0.92**	**1**

- 模糊测试效率验证-技术
  - 结论
    - 确定性阶段作为一种准备阶段，可以为后续的阶段提供一定的语料基础
    - 确定性阶段的FA策略效果不佳的原因是确定性阶段太长，占用了FA过程
- 模糊测试效率验证-时间
  - 结论
    - FA方法可以为每个种子找到更合适的突变算子概率分布,所以更早发现漏洞
    - MOPT的策略通过统计规则定期分配相同的选择概率分布，导致在触发机会时遗漏了某些漏洞



Program	FA-AFL	MOPT
	Vulnerabilities (Median)	Vulnerabilities (Median)
Tiff2bw	**3**	**3**
Objdump	**1**	**1**
Infotocap	**2**	**2**

Program	Vuls	Median Time (FA-AFL)	Median Time (MOPT)
Tiff2bw	Vul1	**625**	16.1
	Vul2	**778**	1203
	Vul3	1872	**1005**
Objdump	CVE-2018-16617	32	32
	CVE-2018-16615	**53**	59
Infotocap	CVE-2018-16616	**52**	65

特点总结与未来展望

特点总结与未来展望

- **算法总结**
  - 变异操作符选择策略
    - AMSFuzz通过奖励和命中次数计算有效性，再基于有效性计算选择概率
    - FA-Fuzz通过萤火虫的亮度（测试效率）调整位置，以优化变异操作符的选择概率分布
  - 算法基础
    - AMSFuzz基于多臂老虎机问题（MAB）
    - FA-Fuzz基于萤火虫算法（FA）
- **未来发展**
  - 模糊测试用于特定的应用，例如网络协议。定制针对特定应用的模糊测试工具。
  - 继续改进AFL，并整合其他技术，如符号执行和污点分析，以解决魔法字节问题和分支约束问题。使用模糊测试技术和漏洞特征来实现面向漏洞的模糊测试工具

参考文献

[1] Gao Z, Xiong H, Dong W, et al. FA-Fuzz: A Novel Scheduling Scheme Using Firefly Algorithm for Mutation-Based Fuzzing[J]. IEEE Transactions on Software Engineering, 2023.

[2] Zhao X, Qu H, Xu J, et al. AMSFuzz: An adaptive mutation schedule for fuzzing[J]. Expert Systems with Applications, 2022, 208: 118162.

[3] Lyu C, Ji S, Zhang C, et al. {MOPT}: Optimized mutation scheduling for fuzzers[C]. 28th USENIX Security Symposium (USENIX Security 19). 2019: 1949-1966.

[4] Sun L, Li X, Qu H, et al. AFLTurbo: Speed up path discovery for greybox fuzzing[C]. 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2020: 81-91.

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！