Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

程序崩溃的根本原因分析技术

硕士研究生 赵智洋

2024年6月30日

问题回溯

- 总结反思
  - 加强PPT内容的内在关联，如加强算法与背景和基础知识的呼应
  - 增强算法讲解的逻辑和粒度，突出算法创新点、重点
  - 注意语速语气，注意断句和停顿

- 相关内容
  - 2024.06.10 谢宁 《基于突变的模糊测试》
  - 2024.05.26 邵思源 《面向网络应用程序的模糊测试》
  - 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

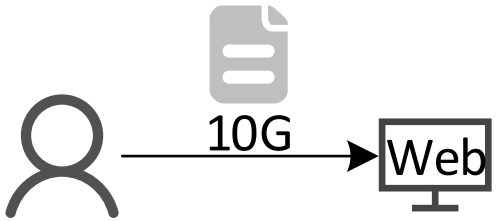
- 预期收获
- 题目内涵解析
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - AURORA
  - RACING
- 特点总结与工作展望
- 参考文献

预期收获

- 预期收获
  - 了解程序缺陷根本原因分析的基本概念及分类
  - 理解程序崩溃根本原因分析面临的挑战和难点
  - 理解程序崩溃根本原因分析的技术原理与思路
  - 了解程序崩溃根本原因分析的技术发展前景

目标内涵　程序崩溃的根本原因分析技术

- 研究目标
  - 以程序缺陷为研究对象，自动推断缺陷的根本原因位置
  - 提升确定导致缺陷的根本原因位置的准确程度
  - 辅助开发人员进行缺陷根本原因的快速定位
- 题目内涵解析
  - 崩溃的原因Crash Cause
    - 导致程序异常终止或无法正常工作的各种情况
  - 崩溃的根本原因Crash Root Cause
    - 导致崩溃的直接触发点、是崩溃原因中最关键的一个
  - 根本原因分析（Root-Cause-Analysis, RCA）
    - 从模糊测试的结果中发现导致缺陷和漏洞产生的根本原因
    - 崩溃的根本原因分析：根据崩溃输入定位根本原因



Crash Cause：用户上传的文件过大，应用程序在处理时耗尽内存，抛出“OOM”异常

Crash Root Cause：应用程序没有限制上传文件的大小，在处理文件时没有实时有效的内存管理策略

背景意义

- 研究背景
  - 模糊测试快速发展：程序的缺陷与日俱增，但修复缺陷的工作主要依靠人工
  - 软件系统日趋复杂：不同组件微妙连接，崩溃发生的位置与根本原因位置不一致
- 研究意义
  - 降本增效：降低手工分析的人力和时间成本，减少程序维护的工作量
  - 提高软件质量：尽快修复软件缺陷，提高软件的整体质量和可靠性
  - 风险管理：防止系统遭受攻击或数据泄露，降低未来出现更严重缺陷的可能性

Linux 开发者“赢了”！研究称 Linux 的漏洞修复速度比苹果 ...

网页 2022年2月22日 · 近日，谷歌 Proj

现：Linux 开发者修复安全漏洞的速度比

漏洞产生快于识别修复，开发者要如何预防安全问题？ - InfoQ

网页 2023年9月25日 · 一个典型的“生态级”漏洞是“水平越权漏洞”。 韦韬举例说，假设用户要访问

情况下，用户只对自己的相关信息感 ...

研究发现安全修补漏洞速度远远慢于黑客利用速度-腾讯云开发 ...

网页 2014年8月29日 · 研究发现安全修补漏洞速度远远

个软件漏洞的分析得出结论，尽管软件安全漏洞与缺陷已

关键漏洞平均修复时间延长至205天 - 安全内参 | 决 ...

网页 2021年6月24日 · WhiteHat Security最新发布的报告《AppSec Stats Flash》显示，修复关键网络安全漏洞所需的平均时间从2021年4月的197天增加到了2021年5月的205天。 报告数据显示，在公用事业部门使用的所有应用程 ...

研究历史与现状　根本原因分析技术

2005
Ben等人提出区分与识别与特定错误相关的预测因子，揭示错误发生的条件和失败的频率

2005
SOBER：通过分析程序在正确和错误运行时的谓词差异，识别与错误相关的谓词

2007
Nainar等人引入复杂的布尔谓词公式提高对程序失败的预测能力

2008
Yu等人设计了测试套缩减策略，探索测试套缩减与故障定位效果之间的权衡关系

2010
TaintScope:基于细粒度的动态污点追踪，定成程序中的校验和检查

2016
RETracer: 分析程序语义，通过二进制级别的后向污点分析理解崩溃原因

2016
POMP:通过反向执行机制和后向污点分析，识别导致崩溃的程序语句

2018
REPT:部署在被测软件中进行故障调试，结合硬件跟踪和二进制分析重建执行历史

2019
DEEPVSA:通过使用指令嵌入和双向神经网络来增强值集分析在内存别名识别方面的能力，辅助定位

2019
Zhang等人提出，通过识别失败执行与非失败执行在共同前缀上的第一个偏离点来定位根本原因

2020
Greyone:模糊测试驱动的污点推断方法，监控变量推断污点，从而筛选分支、字节

2021
VulnLoc: 通过统计定位分析给定漏洞的根本原因位置

2022
PATA:路径感知污点分析的模糊测试器，解决模糊测试中复杂约束问题

2024
BENZENE:设计生成崩溃与非崩溃行为，基于统计调试获知程序执行行为差异以定位根本原因

程序缺陷的根本原因分析技术
- 基于频谱信息
  - 利用谓词信息
  - 利用覆盖信息
- 非统计方法
  - 逆向执行
  - 后向污点分析

基础知识　频谱定位程序缺陷的方法

- 一般步骤
  - 执行测试用例，记录执行信息
  - 获取频谱信息
    - 将执行信息转换为某种抽象表示
    - 如覆盖信息、谓词信息
  - 根据频谱信息进行建模，排序可疑值列表
    - 辅助程序员调试错误
- 谓词 Predicate
  - 定义：
    - 以某个/几个元素为输入参数
    - 并返回True或False值的函数
  - 一元谓词、二元谓词

```
// GreaterThanFive:
// 判断整数是否大于5的一元谓词类
class GreaterThanFive {
public:
    bool operator()(int val) {
        return val > 5;
    }
};
```

```
// CompareIntegers:
// 比较大小的二元谓词类
class CompareIntegers {
public:
    bool operator()(int a, int b) {
        return a < b;
    }
};
```

- 程序的方法$fun()$
  - s_1为错误语句
    - 若m为FALSE，恰好向下传递正确状态
    - 若m为TRUE，则向下传递的错误状态，将在s_7, s_8位置使程序执行出现非预期
  - 输入5个测试用例
    - 错误：$t_1 = (TRUE, 3)$、$t_4 = (TRUE, 5)$、$t_5 = (TRUE, 0)$
    - 正确：$t_2 = (FALSE, 6)$、$t_3 = (FALSE, -6)$
  - 用方法定位导致非预期的根本原因
    - 即根据谓词对测试用例的判断计算谓词排名

	$fun(bool\ m, int\ n)\ \{$
s_1	$b = m;\ //correct: b = FALSE$
s_2	$if(n < 4)\ \{$
s_3	$while(n < 0)\{$
s_4	$n++;$
s_5	$b = TRUE;$
s_6	$\}$
s_7	$print(n);$
s_8	$print(b);$
s_9	$\}\}$

行号	谓词	测试信息					定位结果
		t_1	t_2	t_3	t_4	t_5	
s_1	$b = TRUE$	1	0	0	1	1	4
s_1	$b = FALSE$	0	1	1	0	0	10
s_7	$n > 0$	1	0	0	0	0	1
s_7	$n = 0$	0	0	1	0	1	5
s_7	$n < 0$	0	0	1	0	0	14

基础知识　程序崩溃根本原因分析中的挑战

- 手工分析工具与方法
  - Debugger调试器
    - 从崩溃开始，手动检查最后执行的几条指令、崩溃点的寄存器等
  - Sanitizer 内存检测工具
    - 在没有发生崩溃的情况下，也能识别无效内存访问
  - 反向污点分析：不适用于崩溃位置与根本原因之间没有直接数据流的情况

```
// 指针修改为指向String类型结构体
NotImplementedError = String;
Module.constants;
// 抛出异常时创建NotImplementedError实例，传递给异常处理函数
// 但把错误信息"Module.constants not implemented"(长度0x20)向下传递
// 在底层尝试引用0x20，导致非法内存访问
// 修复：引入类型检查
```

- 挑战
  - 崩溃的位置与导致崩溃的根本原因位置不一致
  - 面对复杂漏洞（如类型混淆漏洞），定位崩溃耗时且需要领域知识
  - 在利用工具和方法的前提下，关键内容仍然需要人工分析
- 自动化程序崩溃根本原因分析，有助于简化识别和解释崩溃根本原因的过程

AURORA: Statistical Crash Analysis for Automated Root Cause Explanation

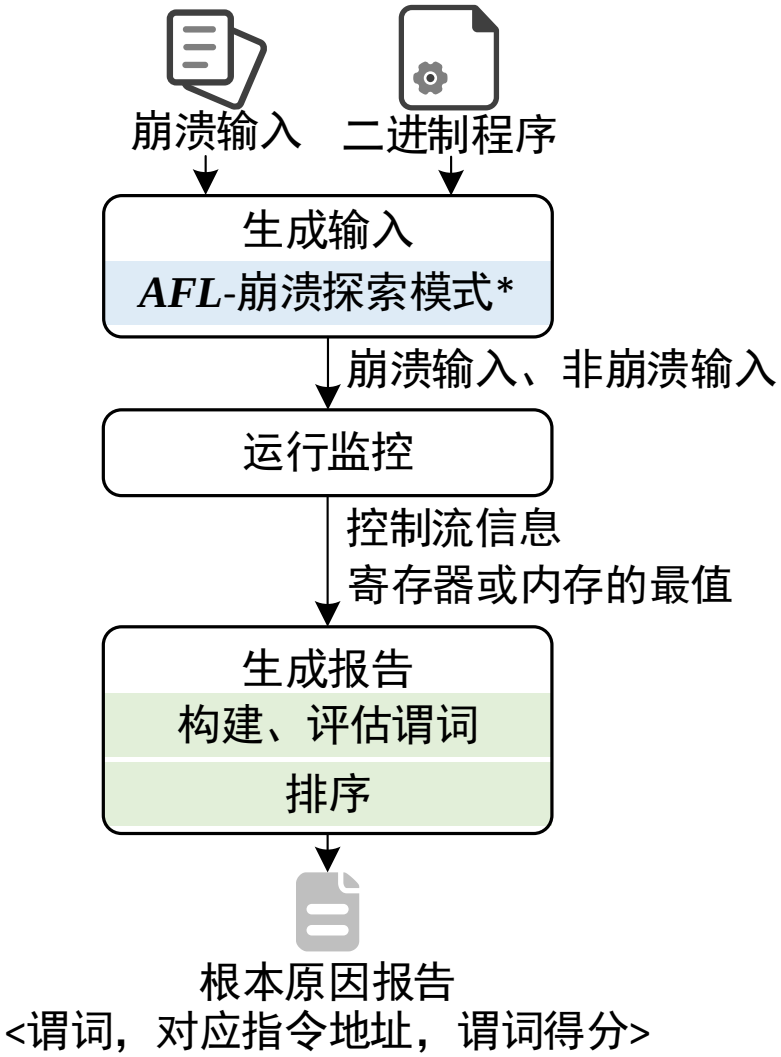
TIPO AURORA

T	目标	实现程序崩溃的根本原因分析
I	输入	1 个崩溃输入、1个二进制程序
P	处理	1.生成输入：利用AFL生成2组测试用例集合，即崩溃输入与非崩溃输入 2.运行监控：运行2组测试用例，监控记录程序行为 3.生成报告：构建谓词，评估谓词能够正确揭示程序崩溃根本原因的可能性
O	输出	1份导致程序崩溃的根本原因报告

P	问题	发生崩溃的位置并不一定是导致崩溃的根本原因的位置
C	条件	仅适用于二进制程序
D	难点	如何有效筛选出能够揭示程序崩溃根本原因的谓词
L	水平	Usenix Security 2021(CCF A)

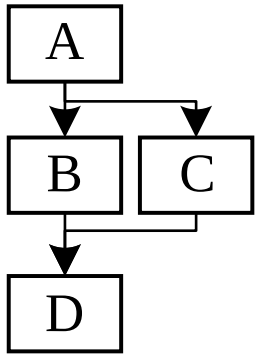
算法原理 AURORA

- 核心思想
  - 程序执行过程中，崩溃输入和非崩溃输入出现首次偏离的位置就是根本原因
  - 通过统计分析崩溃和非崩溃输入造成的程序行为差异，以确定根本原因
- 算法步骤
  - 生成输入：利用AFL生成2组测试用例集合，即崩溃输入与非崩溃输入
  - 运行监控：运行并监控2组测试用例，存储控制流，记录内存写入或寄存器修改时的最大值和最小值
  - 生成报告：根据最值构建谓词，根据崩溃输入和非崩溃输入评估谓词，对有效的谓词进行排序



算法原理　构建谓词

- 目的：筛选出与崩溃和非崩溃共同相关的指令集
  - 即指令对应的地址至少被1个崩溃和1个非崩溃输入访问过
- 控制流谓词
  - 定义：基于控制流信息构建的谓词
  - 作用：评估崩溃和非崩溃输入在执行路径上是否存在差异
- 寄存器和内存谓词
  - 定义：基于各类表达式生成的谓词
  - 作用：评估崩溃和非崩溃输入在执行行为上是否存在差异
- 标志谓词（指令：$add\ rax, rab$）
  - 定义：针对进位、零、溢出的谓词
- 每条指令可能有多个谓词



$successors(A) \geq 2$为真

$has_edge_to(A, B)$ 为真

$always_taken_to(A, B)$为假

指令：$mov\ r, 0x08$
谓词：$rax < 0x400254$, $is_heap_ptr(rax)$…

outcome	*crash*	*crash*	*non-crash*	*non-crash*
values of r	*0x08*	*0x0f*	*0x400254*	*0x400274*

算法原理　评估谓词

- 理想的谓词：能够正确预测所有测试用例的结果
  - 即能够完美区分崩溃和非崩溃输入
- 计算谓词错误预测的实际概率$\hat{\theta}$
  - C_f：错误预测的崩溃；　C_t：正确预测的崩溃
  - N_f：错误预测的非崩溃；　N_t：正确预测的非崩溃
- 计算谓词的分数 $Score$
  - 对于每个指令，仅存储分数最高的谓词
    - $Score \epsilon [0,1]$，分数越高、谓词越好
- 排序：根据谓词分数和谓词执行的时间顺序
  - 筛选出至少被1个崩溃和1个非崩溃输入访问过的指令的最佳谓词

$$\hat{\theta} = \frac{C_f + N_f}{C_f + C_t + N_f + Nt}$$

防止类别不平衡，
导致评分方案偏向更大的类别

$$\hat{\theta} = \frac{1}{2}\left(\frac{C_f}{C_f + C_t} + \frac{N_f}{N_f + Nt}\right)$$

$$\hat{\theta} = \begin{cases} 0, \text{谓词完美区分} \\ 1, \text{谓词取反时，完美区分} \\ \rightarrow 0.5, \text{谓词质量越差} \end{cases}$$

映射至分数域

$$Score = 2 * abs\,(\hat{\theta} - 0.5)$$

实验设计　数据源及评价指标

- 数据源：25个程序崩溃
  - 涉及程序18个：Bash、Lua…
  - 涉及崩溃原因7种：缓冲区溢出、段错误、类型混淆、未初始化变量…
- 评价指标
  - #Predicates：根本原因对应的谓词排名
  - #SLOC：修复真实原因需要检查的源代码行数
- Ground Truth
  - 根据开发者的修复、补丁，识别的导致崩溃的根本原因位置

	Target	Root Cause 根本原因	Crash Cause 崩溃原因	Sanitizer
#1	mruby	int overflow	heap buffer overflow	ASAN
#2	Lua	logic flaw	heap buffer overflow	ASAN
#3	Perl	logic flaw	heap buffer overflow	-
#4	screen *	logic flaw	heap buffer overflow	-
#5	readelf	missing check	heap buffer overflow	ASAN
#6	mruby	missing check	heap buffer overflow	ASAN
#7	objdump	missing check	heap buffer overflow	ASAN
#8	patch	missing check	heap buffer overflow	ASAN
#9	Python	missing check	heap buffer overflow	-
#10	tcpdump	missing check	heap buffer overflow	-
#11	NASM	logic flaw	nullptr dereference	-
#12	Bash	int overflow	segmentation fault	-
#13	Bash	int overflow	segmentation fault	-
#14	Python	race condition	segmentation fault	-
#15	nm *	missing check	stack buffer overflow	ASAN
#16	mruby	missing check	type confusion	-
#17	Python	missing check	type confusion	-
#18	Xpdf	logic flaw	uninitialized variable	ASAN
#19	mruby	missing check	uninitialized variable	MSAN
#20	PHP	missing check	uninitialized variable	MSAN
#21	libzip *	double free	use-after-free	ASAN
#22	mruby	logic flaw	use-after-free	ASAN
#23	NASM *	logic flaw	use-after-free	-
#24	Sleuthkit	logic flaw	use-after-free	-
#25	Lua	missing check	use-after-free	ASAN

实验结果

- 根本原因分析效果评估
  - 有15个崩溃能够在排名Top-10的谓词中找到根本原因
  - 有17个崩溃需要开发者排查的源代码行数在10行以内
- 特殊情况
  - #17：程序指令集合庞大、漏洞类型复杂
  - #16：需要检查的代码大多在同一个函数内
  - # 11：缺乏Ground Truth，排除在分析之外
- 根本原因定位的挑战
  - 根本原因与崩溃位置之间存在一定的距离
    - #4：2800万条指令间隔时，仍有良好的分析效果

Target		Maximum #Instructions all	unique
#3	Perl	845,689	7,321
#4	screen	28,289,736	3,441
#9	Python	3,759,699	9,330
#10	tcpdump	6,727	1,567
#11	NASM	22,678,105	8,256
#12	Bash	450,428	3,549
#13	Bash	2,584,606	1,094
#14	Python	3,923,167	13,028
#16	mruby	253,173	840
#17	Python	800	428
#23	NASM	7,401,732	4,842
#24	Sleuthkit	199	156

Target		Steps to Dev. Fix	
		#Predicates	#SLOC
#1	mruby	1	1
#2	Lua	1	1
#3	Perl	13	10
#4	screen *	26	16
#5	readelf	7	5
#6	mruby	1	1
#7	objdump	3	3
#8	patch	1	1
#9	Python	46	28
#10	tcpdump	1	1
#11	NASM	-	-
#12	Bash	10	6
#13	Bash	9	6
#14	Python	13	13
#15	nm *	1	1
#16	mruby	33	15
#17	Python	215	141
#18	Xpdf	16	11
#19	mruby	16	5
#20	PHP	42	19
#21	libzip *	1	1
#22	mruby	9	6
#23	NASM *	1	1
#24	Sleuthkit	2	2
#25	Lua	3	3

Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples

TIPO RACING

T	目标	实现程序崩溃的根本原因分析
I	输入	1 个崩溃输入、1个二进制程序
P	处理	1.生成输入：利用强化学习生成测试用例 2.评估更新：运行测试用例，并对谓词进行更新与排序 3.收敛输出：当谓词排名情况收敛时，输出最终排名结果
O	输出	1份含50个谓词的谓词排名列表

P	问题	以崩溃为中心的模糊测试导致的偏见采样方法而效率低下
C	条件	需要被分析的程序提供源代码
D	难点	以尽可能快的收敛速度和尽可能少的样本，达到尽可能高的谓词排名准确性
L	水平	USENIX Security Symposium, 2024(CCF A)

算法原理 RACING

- 核心思想
  - 通过强化学习奖励能够显著改变谓词排名的种子和变异策略
- 算法步骤
  - 生成输入$l4-l6$：利用强化学习生成测试用例
  - 评估更新$l7-l11$：运行测试用例，并对谓词进行更新与排序
  - 收敛输出$l12-l14$：当谓词排名情况收敛时，输出最终排名结果

Algorithm 1: RACING algorithm

Input: Program *Prog*, and a Proof of Concept *PoC*.
Result: Top-50 predicates and their true order r^*.

1 **begin**
2 $S_0 = [], \gamma_0 = 0.5, t = 1$
3 **do**
4 $d_t \leftarrow \texttt{Select_Action}(S_{t-1})$
5 $a_t \leftarrow \texttt{Sample_Action}(d_t, \gamma_t)$
6 $\{input\} \leftarrow \texttt{Generate_Input}(a_t)$
7 $\{(e_i, p_i)\} \leftarrow \texttt{Run}(Prog, \{input\})$
8 $r_t \leftarrow \texttt{Rank}(\{(e_i, p_i)\})$
9 $g_t \leftarrow \texttt{Compute_Reward}(\{(e_i, p_i)\}, S_{t-1})$
10 $S_t \leftarrow \texttt{Update_State}(S_{t-1}, r_t, g_t, \{(e_i, p_i)\})$
11 $\gamma_t \leftarrow \texttt{Update_Gamma}(\{g_i\}, \{d_i\}, \{a_i\})$
12 **while** $(!\,\texttt{Converge}(S_t))$
13 $\{p_i\}_{50}, r^* \leftarrow$ select top-50 p_i according to r_t
14 **return** $\{p_i\}_{50}, r^*$
15 **end**

算法原理　反例分析

- 现有问题：
  - 基于已知崩溃输入变异生成的测试用例，趋向于覆盖相似的崩溃路径
  - 这些测试用例对快速筛选与崩溃相关的谓词时效果不佳
- 排名反例（Counterexample for Ranking，CoR）
  - 定义排名反例：能显著改变谓词排名的输入
  - 目的：通过增加对CoR的采样，加快谓词排名的收敛速度
- 谓词反例（Counterexample for Predicate，CoP）
  - 定义谓词反例：执行结果与谓词不一致的测试用例
  - 目的：用反例使根本原因突出，减少所需样本数目
    - 通过增加对CoP的采样，增加谓词的错误率
    - 使非根本原因的谓词与根本原因的谓词距离增加

```
int is_positive(int num) {
    return num >= 0;
}
// num = 0
// 预测结果是非正数
// 但是实际结果是正数
```

- 以尽可能快的收敛速度和尽可能少的样本，达到尽可能高的谓词排名准确性

算法原理　强化学习问题

- 对抗性多臂老虎机（AMAB）问题
  - 玩家在每轮选择一个动作分布，而对手选择奖励向量
  - 玩家（算法）的目标：选择合适的分布最小化总遗憾G_T
  - 对手：奖励机制的设计 、程序执行的结果
- 奖励：$g_t = g_t^{order} + g_t^{count}$
  - 通过奖励指导选择CoR和CoP效果最显著的动作
  - 即指导选择最大化谓词区分崩溃与非崩溃输入能力的动作
- 动作：
  - 种子选择：从反例中选择一组输入作为模糊测试的变异种子
  - 变异策略选择：选择变异的字节位置和一个变异操作
    - 3类变异操作：插入、修改和删除字节

玩家选择动作a_t后，观察到的奖励

$$G_T = \sum_{t=1}^{T} (g_t^* - g_t(a_t))$$

第t轮的最优奖励

g_t^{order}：第t轮前后谓词排名的变化距离

g_t^{count}：违反谓词的输入比例

算法原理　算法步骤

- 选择动作 $Select_Action$
  - 输入：上一轮的状态S_{t-1}
  - 输出：覆盖Na数目的2个动作的概率分布d_t
    - 概率分布$d_t = (d_{t,seed}, d_{t,mutate})$
      - 选择第i组种子的概率
        - » $d_{t,seed}(i) = \exp(avg(i)) / \sum_{j=1}^{N_{a,seed}} \exp(avg(j))$
        - » $avg()$：平均奖励
    - 动作数目$N_a = (N_{a,seed}, N_{a,mutate})$
- 采样动作 $Sample_Action$
  - 输入：概率分布d_t、探索概率$\gamma_t = (\gamma_{t,seed}, \gamma_{t,mutate})$
  - 输出：动作$a_t = (a_{t,seed}, a_{t,mutate})$

Input: Program *Prog*, and a Proof of Concept *PoC*.
Result: Top-50 predicates and their true order r^*.

```
begin
    S_0 = [], γ_0 = 0.5, t = 1
    do
        d_t ← Select_Action(S_{t-1})
        a_t ← Sample_Action(d_t, γ_t)
        {input} ← Generate_Input(a_t)
        {(e_i, p_i)} ← Run(Prog, {input})
        r_t ← Rank({(e_i, p_i)})
        g_t ← Compute_Reward({(e_i, p_i)}, S_{t-1})
        S_t ← Update_State(S_{t-1}, r_t, g_t, {(e_i, p_i)})
        γ_t ← Update_Gamma({g_i}, {d_i}, {a_i})
    while (! Converge(S_t))
    {p_i}_50, r* ← select top-50 p_i according to r_t
    return {p_i}_50, r*
end
```

算法原理　算法步骤

- 输入生成 $Generate_Input$
  - 输入：动作$a_t = (a_{t,seed}, a_{t,mutate})$
  - 输出：一组输入测试用例$\{input\}$
- 运行 Run
  - 输入：一组输入测试用例$\{input\}$
  - 输出：$\{(e_i, p_i)\}$
    - e_i：与谓词相关的测试用例集合；p_i：谓词
- 排序 $Rank$
  - 输入：$\{(e_i, p_i)\}$；输出：根据谓词得分计算的排序r_i
- 收敛$Coverage$：
  - 相邻轮次谓词排名变动距离的方差小于阈值时，收敛、停止迭代

```
begin
    S_0 = [], γ_0 = 0.5, t = 1
    do
        d_t ← Select_Action(S_{t-1})
        a_t ← Sample_Action(d_t, γ_t)
        {input} ← Generate_Input(a_t)
        {(e_i, p_i)} ← Run(Prog, {input})
        r_t ← Rank({(e_i, p_i)})
        g_t ← Compute_Reward({(e_i, p_i)}, S_{t-1})
        S_t ← Update_State(S_{t-1}, r_t, g_t, {(e_i, p_i)})
        γ_t ← Update_Gamma({g_i}, {d_i}, {a_i})
    while (! Converge(S_t))
    {p_i}_50, r* ← select top-50 p_i according to r_t
    return {p_i}_50, r*
end
```

实验设计　数据源及评价指标

- 数据源：30个漏洞
  - 涉及21个程序、9类漏洞
  - 源代码数目范围：857~980,019
- 评价指标
  - Ranking：根本原因对应的谓词排名
  - T：运行时间（时：分：秒）
- 对比方法：Aurora
  - V1-V19来自于Aurora
- Ground Truth
  - 根据开发者的修复、补丁，识别的导致崩溃的根本原因位置

ID	Program	Version	Vulnerability ID	Vulnerability Type
V_1	readelf	binutils-2.32	CVE-2019-9077	heap buffer overflow
V_2	nm	binutils-2.28	Bugzilla-21670	stack buffer overflow
V_3	lua	lua-5.0	bug #5.0-2	heap buffer overflow
V_4	nasm	nasm-2.14rc15	bugzilla-3392556	use after free
V_5	tcpdump	tcpdump-4.9.2	CVE-2017-16808	heap buffer overflow
V_6	sleuthkit	commit 3e2332a	issue-905	double free
V_7	patch	commit dce4683	bug #54558	heap buffer overflow
V_8	bash	commit 6444760	msg00042	integer overflow
V_9	nasm	nasm-2.14rc15	CVE-2018-16517	nullptr dereference
V_{10}	libzip	libzip-1.2.0	CVE-2017-12858	use after free
V_{11}	perl	commit dca9f61	issue-17384	heap buffer overflow
V_{12}	objdump	commit 561bf3e	CVE-2017-9746	heap buffer overflow
V_{13}	mruby	commit 7483753	issue-3947	uninitialized variable
V_{14}	mruby	commit aa5c5de	CVE-2018-12248	heap buffer overflow
V_{15}	mruby	commit 88604e3	hackerone-185041	type confusion
V_{16}	mruby	commit fabc460	CVE-2018-10199	use after free
V_{17}	python	python-2.7.11	CVE-2016-5636	heap buffer overflow
V_{18}	screen	screen-4.7.0	oss-sec-2020/q1/65	heap buffer overflow
V_{19}	mruby	commit e9ddb59	CVE-2018-10191	integer overflow
V_{20}	matio	commit bcf0447	CVE-2020-19497	integer overflow
V_{21}	ezXML	ezxml-0.8.6	CVE-2021-30485	nullptr dereference
V_{22}	libpng	libpng-1.6.34	CVE-2018-13785	divide-by-zero
V_{23}	libjpeg	2.0.90 (2.1 beta1)	CVE-2021-20205	divide-by-zero
V_{24}	libtiff	tiff-4.0.7	CVE-2017-7595	divide-by-zero
V_{25}	libjpeg	commit f4b8a5c	CVE-2018-14498	heap buffer overflow
V_{26}	libxml2	commit 362b322	CVE-2017-5969	nullptr dereference
V_{27}	libtiff	tiff-4.0.6	CVE-2016-5321	heap buffer overflow
V_{28}	libsixel	sixel-1.8.4	CVE-2021-45340	nullptr dereference
V_{29}	nm	binutils-2.32	CVE-2019-17451	integer overflow
V_{30}	objdump	binutils-2.32	CVE-2019-9074	heap buffer overflow

实验结果　对比实验——谓词排名情况

- RACING
  - 22个（73.33%）排名在Top-10以内，10个（33.33%）位于Top-3，4个直接定位到根本原因
- 对比Aurora
  - 6个排名相同，14个（46.67%）排名比Aurora更高，10个排名比Aurora更低
- RACING比Aurora平均排名高出3.80
  - Aurora随机生成的测试用例充分性不确定
    - 测试用例基于崩溃输入变异生成
  - RACING筛选反例作为变异种子
    - 测试用例向提升根本原因的谓词排名准确性的方向生成

ID	RACING			
	Ranking	T_{all}	$T_{ranking}$	T_{top100}
V_1	10	00:00:40	00:00:14	00:00:12
V_2	1	00:01:36	00:00:51	00:00:01
V_3	28	00:04:38	00:01:41	00:00:03
V_4	8	00:03:06	00:02:44	00:00:01
V_5	4	00:02:12	00:00:21	00:00:10
V_6	9	00:01:57	00:01:24	00:00:18
V_7	1	00:02:38	00:00:03	00:00:01
V_8	20	00:03:25	00:03:19	00:00:02
V_9	20	00:12:07	00:02:44	00:00:03
V_{10}	2	00:14:33	00:10:15	00:10:15
V_{11}	16	00:16:46	00:13:01	00:00:30
V_{12}	4	00:23:17	00:05:04	00:02:26
V_{13}	3	00:19:22	00:01:45	00:00:27
V_{14}	10	00:24:12	00:02:05	00:01:12
V_{15}	39	00:46:52	00:46:52	00:00:13
V_{16}	3	01:05:21	00:01:33	00:00:36
V_{17}	33	01:32:38	00:34:00	00:08:10
V_{18}	9	01:55:19	00:27:50	00:01:44
V_{19}	18	04:29:23	03:46:20	03:46:20
V_{20}	3	00:00:15	00:00:07	00:00:00
V_{21}	3	00:00:12	00:00:01	00:00:00
V_{22}	1	00:00:40	00:00:30	00:00:11
V_{23}	9	00:01:56	00:00:07	00:00:00
V_{24}	3	00:02:20	00:00:06	00:00:03
V_{25}	1	00:02:32	00:00:12	00:00:12
V_{26}	6	00:05:04	00:03:16	00:00:03
V_{27}	14	00:05:37	00:01:41	00:00:07
V_{28}	8	00:09:01	00:01:31	00:01:17
V_{29}	7	00:25:58	00:14:57	00:00:40
V_{30}	4	00:53:21	00:15:08	00:01:51
Avg.	9.90	00:27:34	00:13:59	00:08:34

实验结果　对比实验——效率情况

- 总时间成本T_all
  - Aurora：生成时间+分析时间
  - RACING：生成和分析连续进行
- 倍速：
  - 最大加速602.42倍（V21）
  - 最小加速1.15倍（V18）
- 识别根本原因的平均时间：
  - Aurora需要大约6小时
  - RACING大约需要27分钟来识别根本原因，减少了13.22倍

ID	Aurora					Speedup
	Ranking	T_{all}	$T_{ranking}$	$T_{fuzzing}$	$T_{analysis}$	
V_1	6	02:06:52	00:08:00	2:00:00	0:06:52	190.30×
V_2	1	12:00:50	00:15:00	12:00:00	0:00:50	450.52×
V_3	21	12:14:49	08:00:00	12:00:00	0:14:49	158.59×
V_4	14	02:24:13	00:20:00	2:00:00	0:24:13	46.52×
V_5	4	02:02:04	00:09:00	2:00:00	0:02:04	55.48×
V_6	10	02:02:05	00:06:00	2:00:00	0:02:05	62.61×
V_7	1	02:04:17	00:00:20	2:00:00	0:04:17	47.20×
V_8	29	03:10:46	02:00:00	2:00:00	1:10:46	55.83×
V_9	58	02:39:57	01:20:00	2:00:00	0:39:57	13.20×
V_{10}	4	12:01:13	06:00:00	12:00:00	0:01:13	49.57×
V_{11}	9	06:15:05	01:30:00	2:00:00	4:15:05	22.37×
V_{12}	6	02:12:15	02:00:00	2:00:00	0:12:15	5.68×
V_{13}	1	12:57:01	00:09:00	12:00:00	0:57:01	40.12×
V_{14}	1	12:42:09	00:03:00	12:00:00	0:42:09	31.49×
V_{15}	47	14:00:14	08:00:00	12:00:00	2:00:14	17.93×
V_{16}	3	15:03:16	04:00:00	12:00:00	3:03:16	13.82×
V_{17}	28	18:08:53	10:00:00	12:00:00	6:08:53	11.75×
V_{18}	41	02:12:10	01:10:00	2:00:00	0:12:10	1.15×
V_{19}	9	14:05:54	10:00:00	12:00:00	2:05:54	3.14×
V_{20}	1	02:00:04	00:30:00	2:00:00	0:00:04	480.27×
V_{21}	3	02:00:29	00:20:00	2:00:00	0:00:29	602.42×
V_{22}	9	02:00:43	00:20:00	2:00:00	0:00:43	181.08×
V_{23}	14	02:11:49	00:10:00	2:00:00	0:11:49	68.18×
V_{24}	10	02:07:27	00:04:00	2:00:00	0:07:27	54.62×
V_{25}	1	02:01:43	00:00:20	2:00:00	0:01:43	48.05×
V_{26}	11	02:07:15	01:00:00	2:00:00	0:07:15	25.12×
V_{27}	7	02:07:06	00:10:00	2:00:00	0:07:06	22.63×
V_{28}	2	12:05:43	07:00:00	12:00:00	0:05:43	80.49×
V_{29}	47	02:03:17	00:20:00	2:00:00	0:03:17	4.75×
V_{30}	13	02:01:48	00:04:00	2:00:00	0:01:48	2.28×
Avg.	13.70	06:04:26	02:10:17	5:20:00	0:44:26	13.22×

实验结果　对比实验——效率情况

- $T_{ranking}$评估浪费时间的情况
  - 统计首次获取与最终排名一致的排名所需要的时间
  - 结论：Aurora在生成更多的测试用例时浪费了大量的时间（如V2）
- $Ranking*$ 衡量获取尽可能高的排名的速度
  - 统计Aurora花费与RACING相同时间时，谓词的排名情况
  - 结果：83.33%（25个）根本原因的排名低于RACING计算的排名

ID	Aurora					Speedup	Ranking*
	Ranking	T_{all}	$T_{ranking}$	$T_{fuzzing}$	$T_{analysis}$		
V_1	6	02:06:52	00:08:00	2:00:00	0:06:52	190.30×	14
V_2	1	12:00:50	00:15:00	12:00:00	0:00:50	450.52×	8
V_3	21	12:14:49	08:00:00	12:00:00	0:14:49	158.59×	84
V_4	14	02:24:13	00:20:00	2:00:00	0:24:13	46.52×	37
V_5	4	02:02:04	00:09:00	2:00:00	0:02:04	55.48×	9
V_6	10	02:02:05	00:06:00	2:00:00	0:02:05	62.61×	24
V_7	1	02:04:17	00:00:20	2:00:00	0:04:17	47.20×	1
V_8	29	03:10:46	02:00:00	2:00:00	1:10:46	55.83×	1720
V_9	58	02:39:57	01:20:00	2:00:00	0:39:57	13.20×	45
V_{10}	4	12:01:13	06:00:00	12:00:00	0:01:13	49.57×	6
V_{11}	9	06:15:05	01:30:00	2:00:00	4:15:05	22.37×	138
V_{12}	6	02:12:15	02:00:00	2:00:00	0:12:15	5.68×	84
V_{13}	1	12:57:01	00:09:00	12:00:00	0:57:01	40.12×	1
V_{14}	1	12:42:09	00:03:00	12:00:00	0:42:09	31.49×	1
V_{15}	47	14:00:14	08:00:00	12:00:00	2:00:14	17.93×	74
V_{16}	3	15:03:16	04:00:00	12:00:00	3:03:16	13.82×	7
V_{17}	28	18:08:53	10:00:00	12:00:00	6:08:53	11.75×	510
V_{18}	41	02:12:10	01:10:00	2:00:00	0:12:10	1.15×	41
V_{19}	9	14:05:54	10:00:00	12:00:00	2:05:54	3.14×	16
V_{20}	1	02:00:04	00:30:00	2:00:00	0:00:04	480.27×	18
V_{21}	3	02:00:29	00:20:00	2:00:00	0:00:29	602.42×	31
V_{22}	9	02:00:43	00:20:00	2:00:00	0:00:43	181.08×	4
V_{23}	14	02:11:49	00:10:00	2:00:00	0:11:49	68.18×	19
V_{24}	10	02:07:27	00:04:00	2:00:00	0:07:27	54.62×	7
V_{25}	1	02:01:43	00:00:20	2:00:00	0:01:43	48.05×	1
V_{26}	11	02:07:15	01:00:00	2:00:00	0:07:15	25.12×	15
V_{27}	7	02:07:06	00:10:00	2:00:00	0:07:06	22.63×	18
V_{28}	2	12:05:43	07:00:00	12:00:00	0:05:43	80.49×	12
V_{29}	47	02:03:17	00:20:00	2:00:00	0:03:17	4.75×	47
V_{30}	13	02:01:48	00:04:00	2:00:00	0:01:48	2.28×	13
Avg.	13.70	06:04:26	02:10:17	5:20:00	0:44:26	13.22×	100.17

特点总结与工作展望

- 特点总结

算法	Aurora	RACING
优势	1.无需使用源代码 2.能处理根本原因与崩溃之间没有数据依赖关系的缺陷	1.克服测试用例生成带来的不确定性 2.从快速收敛和缩减样本的角度提升分析效率
劣势	1.生成测试用例的效率较低 2.基于崩溃生成新的测试用例，与崩溃相关的程序实体可能被重复采样；与根本原因相关的程序实体不突出	1.需要使用源代码，即以来对程序的源代码插桩获取信息 2.仅能处理原子谓词，不支持复合谓词的情况

- 未来发展
  - 平衡采样方法：利用无偏见的模糊测试工具生成测试用例
  - 谓词列表生成的完备性：保证谓词列表中存在与根本原因相关的谓词
  - 泛化性方法：设计覆盖错误类型更广泛的根本原因分析算法

参考文献

[1] Blazytko T, Schlögel M, Aschermann C, et al. {AURORA}: Statistical crash analysis for automated root cause explanation[C].29th USENIX Security Symposium (USENIX Security 20). 2020: 235-252.

[2] Xu D, Tang D, Chen Y, et al. Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples[J].

[3] Park Y, Lee H, Jung J, et al. BENZENE: A Practical Root Cause Analysis System with an Under-Constrained State Mutation[C].2024 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 2023: 74-74.

[4]Renieres M, Reiss S P. Fault localization with nearest neighbor queries [C].Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003). San Franciso: IEEE Computer Society Press

道德经

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！