Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

程序崩溃的故障定位方法

硕士研究生 王怡男

2025年01月12日

问题回溯

- 相关内容
  - 九尾狐
  - 2024.06.30 赵智洋《程序崩溃的根本原因分析技术》
  - 2022.10.30 孔令迪《函数级漏洞检测》

内容提要

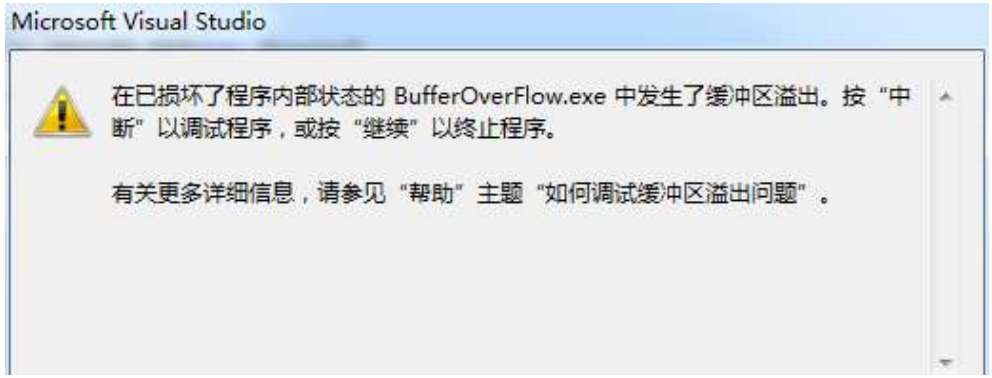
- 预期收获
- 题目内涵解析
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - ARCUS
  - BENZENE
- 特点总结与工作展望
- 参考文献

预期收获

- 预期收获
  - 1. 了解故障定位的基本概念和定位方法分类
  - 2. 理解统计调试在故障定位中的原理及应用
  - 3. 理解符号执行在故障定位中的原理及应用

题目内涵解析 何为程序崩溃？何为故障定位？

- 程序崩溃（Program Crash）
  - 程序在运行时由于未预料到的异常情况而终止
  - 常见原因
    - 内存错误：缓冲区溢出、Use After Free
    - 算术错误：整数溢出、除0错误
    - 逻辑错误：代码分支遗漏
  - 重要概念
    - 崩溃输入：触发程序崩溃的输入
    - 非崩溃输入：使程序正确运行的输入

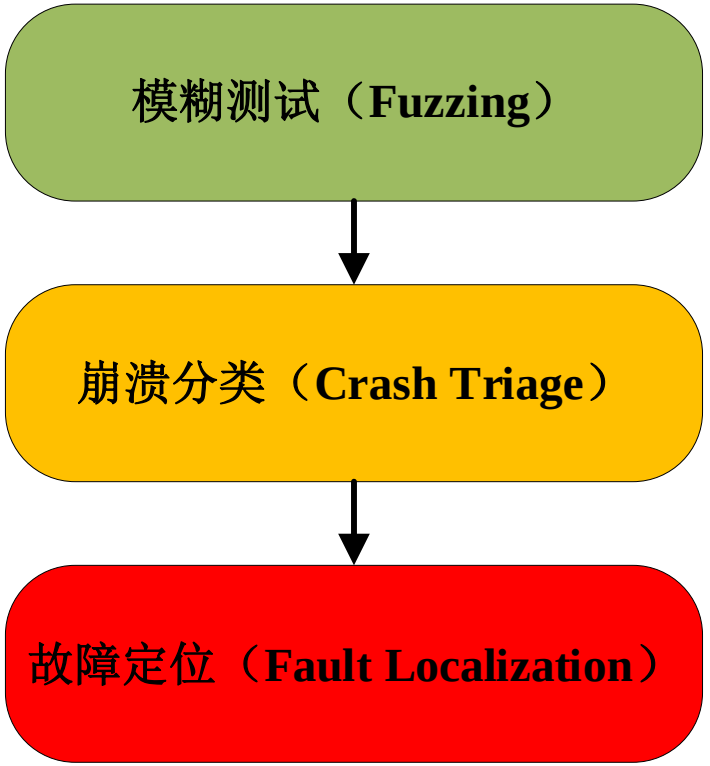



```
Traceback (most recent call last):
  File "D:\test.py", line 4, in <module>
    result = numerator / denominator
ZeroDivisionError: division by zero
```

- 故障定位（Fault Localization）
  - 在程序崩溃后，分析崩溃原因、确定导致崩溃的具体代码或程序状态
  - 根据崩溃输入，找到程序中存在缺陷的位置，并为修复问题提供依据

题目内涵解析 从模糊测试到故障定位

- 模糊测试：漏洞发现的起点
  - 通过随机生成或变异输入数据，触发程序的异常行为，发现潜在漏洞
  - 只能发现程序崩溃，但不能解释为什么崩溃
- 崩溃分类：桥接模糊测试和故障定位
  - 对模糊测试发现的崩溃进行分类和整理
  - 从产生的重复崩溃输入中，筛选、归类崩溃
- 故障定位：从现象到根因
  - 精确定位触发崩溃的代码或条件
    - 崩溃点：编译器或解释器抛出异常的位置
    - 根因点：导致崩溃的真正原因所在的代码位置




```
void process_array(int *arr, int size){
    for (int i = 0; i <= size; i++){ //根因点
        arr[i] = i * 2; //崩溃点
    }
}
```

研究背景与意义

- 研究背景
  - 信息技术飞速发展，从手机APP崩溃到关键工业系统异常，软件崩溃**广泛存在且难以避免**
  - 大型软件项目的崩溃涉及**代码量巨大**，定位复杂性几何级增长，难以手动定位根因点
- 研究意义
  - 维护关键信息系统的**稳定性**
    - 对航空航天、金融交易、医疗设备等领域的关键信息系统安全至关重要
    - 快速定位问题、解决问题，减少崩溃停机时间
  - 改善用户体验

研究历史与现状 程序崩溃的故障定位

基于统计调试

2007

Nainar等人通过添加由简单谓词派生的复杂布尔公式，丰富了谓词的词汇表，并证明了复杂谓词的可行性及精准性

2019

Kairux：将一次执行建模为一组完全有序的指令序列，通过找出崩溃执行与非崩溃执行之间最长的指令序列前缀的分歧点进行故障定位

2021

AURORA：通过一个单一的崩溃输入，生成一组多样化的崩溃输入和非崩溃输入，跟踪程序状态后对得到的谓词进行统计分析，识别出故障定位的关键谓词

2024

RACING：通过强化学习技术采样崩溃输入的变异样本，避免偏见采样时的效率低下问题

2011

Chandra等人提出了一种符号执行方法，通过机械化地修复错误来搜索程序所有可能的编辑空间进行故障定位

2019

Birch等人改进搜索方法，通过遍历测试套件减少了符号执行的工作量，进一步缩小了定位集合的范围

2021

ARCUS：通过符号执行分析程序执行路径中的数据依赖、控制依赖关系，结合动态路径约束解决路径爆炸问题

2024

BENZENE：通过结合统计调试与符号执行方法，提出欠约束状态变异，直接修改程序的中间状态生成非崩溃行为，提高故障定位成功率并大幅降低开销

基于符号执行

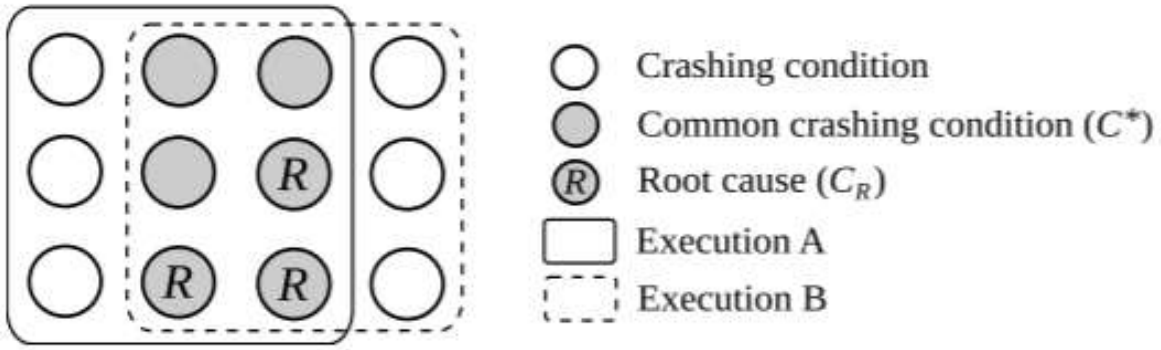
知识基础 谓词

- 谓词（Predicate）
  - 逻辑表达式，描述程序中某一时刻的布尔条件（True/False）
  - 程序运行时的“检查点”，记录了某些关键条件是否被满足
  - 如果某个谓词在崩溃时未满足，很可能该条件与崩溃相关
- 谓词与崩溃条件
  - 崩溃条件是导致程序崩溃的逻辑约束，因此可以用谓词表示
  - 在崩溃输入和非崩溃输入之间，一些谓词的真假变化会揭示关键问题

序号	表达式
E_{23}	$(\exists x)(A(x)\vee B(x))\Leftrightarrow(\exists x)A(x)\vee(\exists x)B(x)$
E_{24}	$(\forall x)(A(x)\wedge B(x))\Leftrightarrow(\forall x)A(x)\wedge(\forall x)B(x)$
E_{25}	$\neg(\exists x)A(x)\Leftrightarrow(\forall x)\neg A(x)$
E_{26}	$\neg(\forall x)A(x)\Leftrightarrow(\exists x)\neg A(x)$
E_{27}	$(\forall x)(A\vee B(x))\Leftrightarrow A\vee(\forall x)B(x)$
E_{28}	$(\exists x)(A\wedge B(x))\Leftrightarrow A\wedge(\exists x)B(x)$
E_{29}	$(\exists x)(A(x)\rightarrow B(x))\Leftrightarrow(\forall x)A(x)\rightarrow(\exists x)B(x)$
E_{30}	$(\forall x)A(x)\rightarrow B\Leftrightarrow(\exists x)(A(x)\rightarrow B)$
E_{31}	$(\exists x)A(x)\rightarrow B\Leftrightarrow(\forall x)(A(x)\rightarrow B)$
E_{32}	$A\rightarrow(\forall x)B(x)\Leftrightarrow(\forall x)(A\rightarrow B(x))$
E_{33}	$A\rightarrow(\exists x)B(x)\Leftrightarrow(\exists x)(A\rightarrow B(x))$
I_{17}	$(\forall x)A(x)\vee(\forall x)B(x)\Rightarrow(\forall x)(A(x)\vee B(x))$
I_{18}	$(\exists x)(A(x)\wedge B(x))\Rightarrow(\exists x)A(x)\wedge(\exists x)B(x)$
I_{19}	$(\exists x)A(x)\rightarrow(\forall x)B(x)\Rightarrow(\forall x)(A(x)\rightarrow B(x))$

知识基础 统计调试

- 统计调试
  - 通过统计分析崩溃行为与非崩溃行为的差异来定位故障的技术
  - 依赖程序运行的覆盖信息和行为特征，通过统计模型计算代码段与崩溃的关联度
- 一个引理的推导
  - 将每个谓词p_i设置为二进制位1或0
  - 每个崩溃条件C_i表示为一个位向量，例如C_1=1110， C_2=1011
  - 对C_1和C_2按位与，得到通用崩溃条件C^*
  - 若定义根本崩溃条件C_R是导致崩溃的最小必要条件
  - 则根本崩溃条件C_R必须包含于通用崩溃条件C^*



知识基础 符号执行

- 符号执行
  - 一种程序分析技术，通过用符号变量代替具体值来分析程序行为，推导程序的逻辑路径和状态条件
  - 在故障定位中，符号分析可用于精确描述导致崩溃的条件（谓词）并回溯到根因点
  - 核心原理
    - 符号化输入：将程序输入表示为符号变量
    - 路径条件生成：程序执行时，根据分支条件记录路径约束
    - 约束求解：使用求解器对路径条件进行求解，生成特定输入或检查条件是否满足
    - 之后，分析崩溃点的路径条件，并回溯相关变量或逻辑条件，定位导致崩溃的根因

```
void process(int x){
    if (x > 10) {   //条件：x ＞ 10
        crash(); //崩溃点
    }
}
```

约束求解器给出x=11触发崩溃

崩溃条件x>10是导致崩溃的逻辑错误

ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems

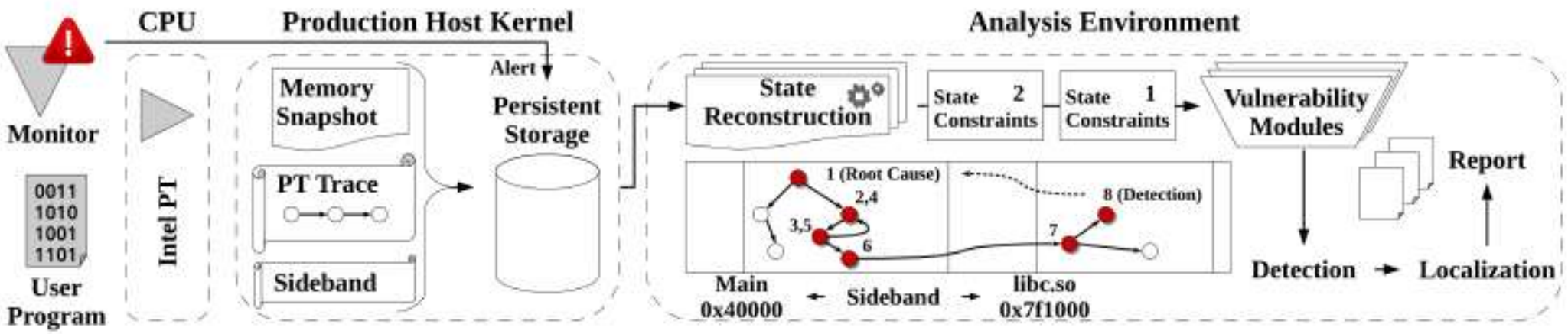
算法原理 TIPO

T	目标	实现程序崩溃的故障定位
I	输入	1个崩溃快照、控制流追踪数据、1个二进制程序、1个崩溃输入
P	处理	1. Intel PT进行硬件级别的高效控制流追踪 2. 通过符号执行分析程序执行路径中的数据依赖关系，定位导致崩溃的状态 3. 在崩溃点之前，通过控制依赖关系和动态路径约束分析定位根因
O	输出	触发崩溃的故障代码、修复建议

P	问题	符号执行在路径分析中容易产生路径爆炸
C	条件	仅覆盖特定漏洞类别
D	难点	路径追踪数据的还原；修复点选择
L	水平	SENIX Security Symposium, 2021(CCF A)

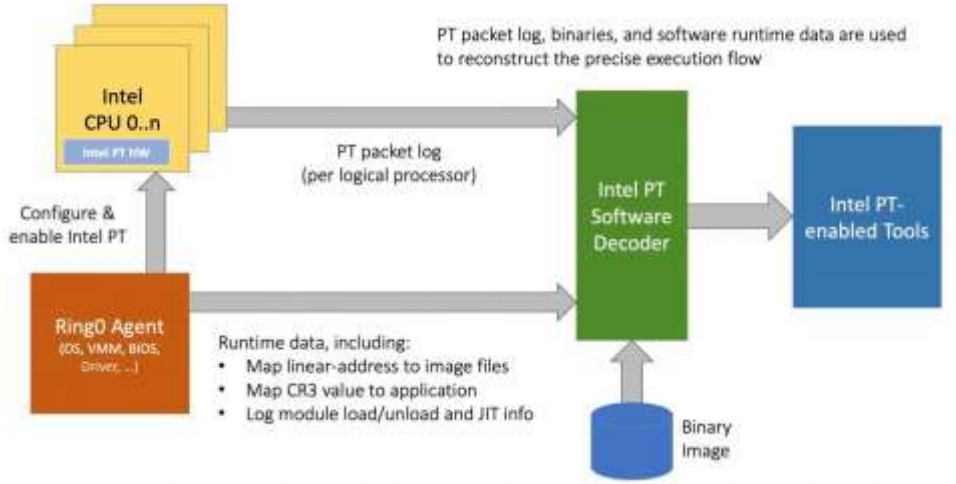
算法原理 创新说明

- ARCUS
  - 离线崩溃分析
    - 直接利用硬件级PT数据，还原崩溃前完整的执行路径，避免对生产环境产生影响，降低性能开销
  - What If Questions
    - 提出假设路径，测试替代约束下程序是否仍然崩溃



算法原理 控制流追踪与重建

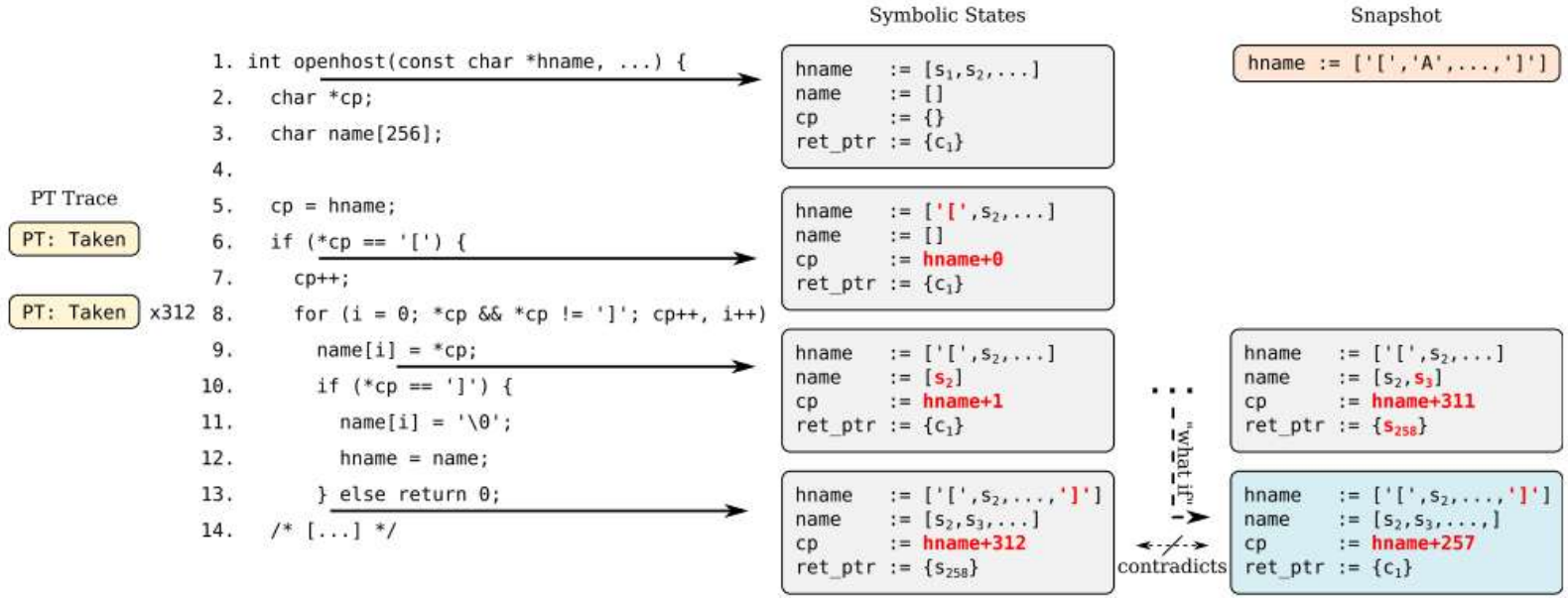
- 控制流追踪
  - Intel PT：一种硬件支持的控制流跟踪工具
    - 记录分支指令的执行信息
    - 记录间接跳转、函数调用和返回的目标地址
    - 为了优化性能，不记录数据流
  - 快照捕获
    - 程序崩溃时，捕获寄存器状态、堆栈布局和动态分配对象的位置



- 控制流重建
  - 利用PT数据中的分支记录和快照，还原程序从启动到崩溃点的执行路径
  - 重建的控制流路径为符号化执行提供准确的路径约束，避免不必要的路径分叉导致的路径爆炸问题

算法原理 符号执行

- 符号化初始状态
  - 使用崩溃快照中的数据作为初始状态；将输入符号化，表示为符号变量
- 动态路径约束
  - 结合控制流重建的路径信息，只符号化实际执行的路径，而非所有可能路径
- 污点分析
  - 识别导致崩溃状态的数据依赖
- What If
  - 通过假设路径分析和动态路径约束验证不同输入条件下的行为



算法原理 崩溃根因分析

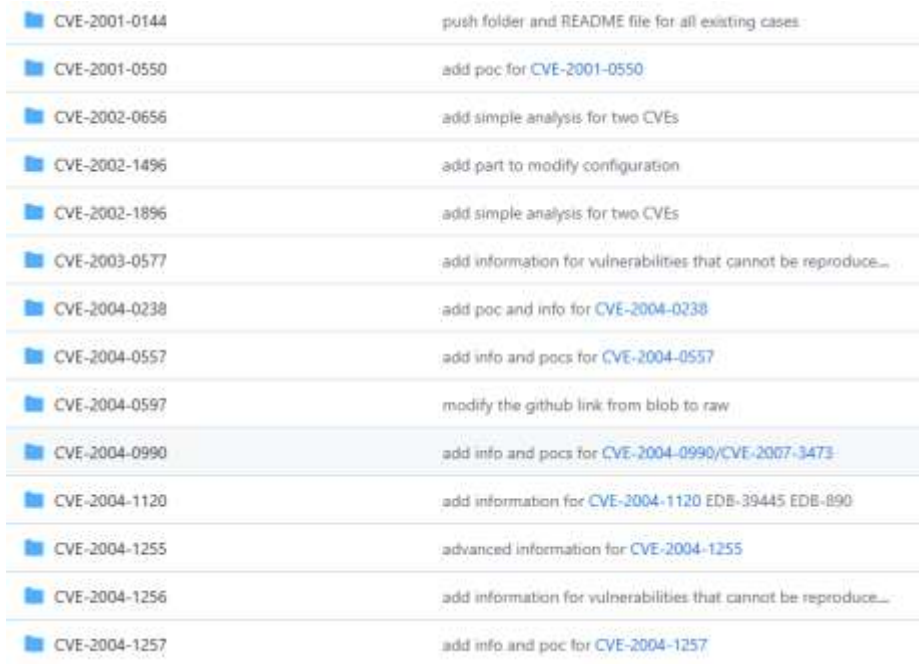
- 回溯根因状态
  - 使用污点分析回溯导致崩溃的符号变量或输入数据
  - 定位数据第一次被错误写入的位置，即根因状态
  - 对6种常见漏洞给出基于领域专业知识的分析策略及根因分类

漏洞类别	定位策略	根因分类
栈溢出Stack Overflow	符号化程序计数器	控制依赖
堆溢出Heap Overflow	符号化程序计数器	控制依赖
整数溢出Integer Overflow	溢出寄存器/内存	溢出发生的具体位置
释放后重用Use After Free	读取/写入被释放的地址	控制依赖
双重释放Double Free	追踪释放操作	控制依赖
格式化字符串Format String	符号化参数	数据依赖

- 守护点
  - 定位最接近崩溃点的修复点，验证是否可以通过增加约束防止崩溃

实验设置 数据资源

- 数据资源
  - 从LinuxFlaw和Exploit-DB中选择27个真实漏洞
- 评估标准
  - 使用ARCUS分析漏洞的根本原因，生成修复建议
  - 手动验证建议内容是否与官方漏洞公告的补丁一致，若一致，认为正确定位故障



实验设置 实验结果

- 定位能力
  - ARCUS成功定位27个漏洞中的所有根因
- 报告一致性
  - 在19个有补丁的漏洞中，ARCUS生成的修复建议与官方补丁完全一致，只有1个例外
  - 例外情况的修复是等价的

Table 4: System Evaluation for Real-World Vulnerabilities

CVE / EDB	Type	Program	# BBs	Size (MB)	ΔRoot Cause	ΔAlert	Located	Has Patch	Match
CVE-2004-0597	Heap	GIMP (libpng)	41,625,163	56.0	247	1	Yes	[61]	Yes†
CVE-2004-1279	Heap	jpegtoavi	67,772	0.65	26,216	1	Yes	No	-
CVE-2004-1288	Heap	o3read	74,723	0.65	33,211	1	Yes	[62]	Yes
CVE-2009-2629	Heap	nginx	300,071	1.10	28	33,824	Yes	[63]	Yes
CVE-2009-3896	Heap	nginx	283,157	1.10	59	16,821	Yes	[64]	Yes
CVE-2017-9167	Heap	autotrace	75,404	1.01	1,828	2	Yes	No	-
CVE-2018-12326	Heap	Redis	291,275	1.20	8	234	Yes	[65]	Yes
EDB-15705	Heap	ftp	260,986	0.85	19,322	2	Yes	No	-
CVE-2004-1257	Stack	abc2mtex	53,490	0.67	6,319	1	Yes	No	-
CVE-2009-5018	Stack	gif2png	90,738	1.09	1,848	1	Yes	[66]	Yes
CVE-2017-7938	Stack	dmitry	100,186	0.71	4,051	14,402	Yes	No	-
CVE-2018-12327	Stack	ntpq	374,830	1.85	122,740	77,990	Yes	[67]	Yes
CVE-2018-18957	Stack	GOOSE (libiec61850)	65,198	0.71	94	30	Yes	[68]	Yes
CVE-2019-14267	Stack	pdfresurrect	128,427	0.66	83,123	1	Yes	[69]	Yes
* EDB-47254	Stack	abc2mtex	53,490	0.67	6,566	-	Yes	No	-
EDB-46807	Stack	MiniFtp	60,849	0.69	335	107	Yes	No	-
CVE-2006-2025	Integer	GIMP (libtiff)	78,419,067	55.0	3	8	Yes	[70]	Yes
CVE-2007-2645	Integer	exif (libexif)	67,697	0.97	1	7	Yes	[71]	Yes
CVE-2013-2028	Integer	nginx	809,977	2.00	1	25,268	Yes	[72]	Yes
CVE-2017-7529	Integer	nginx	1,049,494	1.10	2	780,404	Yes	[73]	Yes
CVE-2017-9186	Integer	autotrace	75,142	1.00	1	1	Yes	No	-
CVE-2017-9196	Integer	autotrace	74,695	1.03	1	203	Yes	No	-
* CVE-2019-19004	Integer	autotrace	132,302	1.02	1	-	Yes	No	-
CVE-2017-11403	UAF	GraphicsMagick	2,316,152	4.61	38	1	Yes	[74]	Yes
CVE-2017-14103	UAF	GraphicsMagick	2,316,133	4.61	38	1	Yes	[74]	Yes
CVE-2017-9182	UAF	autotrace	132,302	1.02	296	58,058	Yes	No	-
* CVE-2019-17582	UAF	PHP (libzip)	5,980,255	6.40	49	-	Yes	[75]	Yes
CVE-2017-12858	DF	PHP (libzip)	5,980,255	6.40	51	719	Yes	[75]	Yes
* CVE-2019-19005	DF	autotrace	132,302	1.02	57,859	-	Yes	No	-
CVE-2005-0105	FS	typespeed	127,209	0.74	1	1	Yes	[76]	Yes
CVE-2012-0809	FS	sudo	108,442	0.69	1	1	Yes	[77]	Yes
		Average:	**4,568,619**	**5.07**	**11,722**	**36,804**			

* New vulnerability discovered by ARCUS.

† Equivalent to applied patch.

BENZENE: A Practical Root Cause Analysis System with an Under-Constrained State Mutation

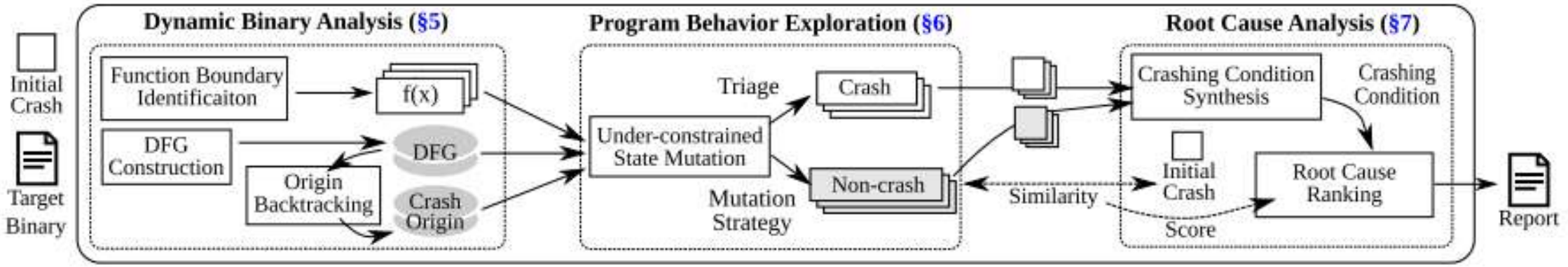
算法原理 TIPO

T	目标	实现程序崩溃的故障定位
I	输入	1个二进制程序、1个崩溃输入
P	处理	1. 通过动态分析二进制代码，回溯崩溃起源 2. 通过欠约束状态变异生成与崩溃相关但不导致崩溃的行为，并提取崩溃行为与非崩溃行为的相似性 3. 合成崩溃条件，推断崩溃的故障定位
O	输出	1份崩溃相关代码位置的详细报告

P	问题	现有基于统计调试的方法难以产生有效非崩溃行为；符号执行有路径爆炸问题
C	条件	需要提供1个导致崩溃的原始输入
D	难点	从崩溃点到根因点的回溯；生成非崩溃行为的可用性
L	水平	IEEE Symposium on Security and Privacy, 2024(CCF A)

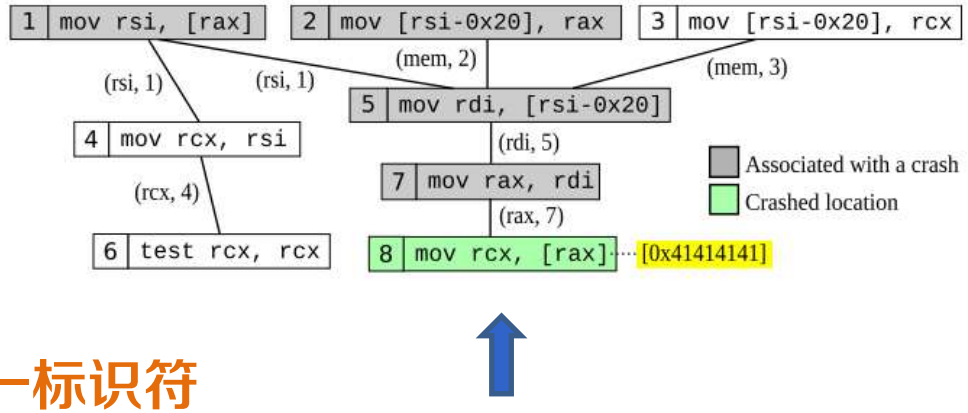
算法原理 创新说明

- **BENZENE**
  - 欠约束状态变异
    - 在程序运行时直接修改程序状态，以生成崩溃相关但非崩溃的行为，避免了对输入进行变异可能导致的巨大开销
  - 行为相似性分析
    - 引理：非崩溃行为越接近崩溃行为，但在关键谓词上越明显地“矛盾”，则越有可能揭示崩溃的根本原因
    - 使用边覆盖率测量崩溃行为和非崩溃行为的相似性

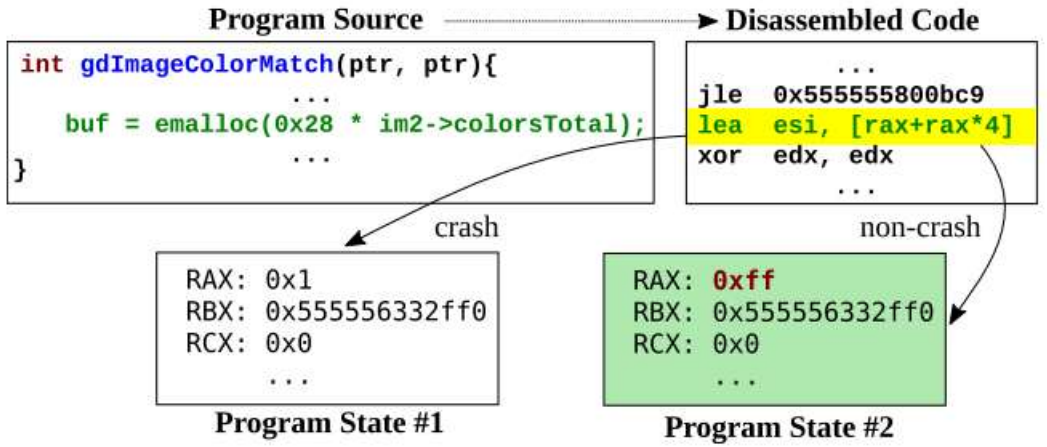


算法原理 动态二进制分析

- 函数边界识别
  - 利用动态分析监控程序的调用栈信息
  - call 指令和ret 指令识别函数边界
- 数据流图（DFG）构建
  - 跟踪寄存器和内存的值，建立数据依赖关系
    - 每条汇编指令为一个节点，每个节点赋予唯一标识符
    - 边定义为数据流，包括寄存器、内存地址和操作数
    - 标记每条指令的源操作数，通过寄存器或内存地址追踪其数据来源
- 崩溃起源回溯
  - 以崩溃点的指令为起点，沿数据流图的反向边回溯，找到相关的源节点



- 欠约束状态变异
  - 对正在运行的程序指令进行修改，改变其操作数或寄存器的值，使程序能够从崩溃路径切换到非崩溃路径，生成非崩溃行为
  - 变异策略
    - 函数级分支
      - 只变异与崩溃相关的函数
    - 目标指令定位
      - 只变异引用了外部值的指令源操作数
    - 值集合（Bag of Values, BoV）
      - 为每个指令维护一个合理值的集合
      - 合理值：从数据流图中回溯值的来源，记录观察到的历史值为合理值



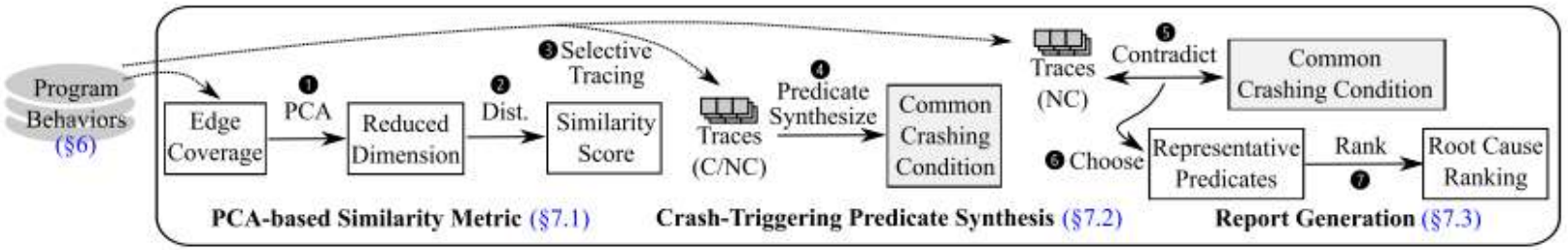
算法原理 崩溃条件合成

- 基于PCA的相似性度量
  - 收集崩溃行为与非崩溃行为的边覆盖率矩阵
  - 通过主成分分析（PCA）降维，保留最重要的特征
  - 使用L2距离评估非崩溃行为与初始崩溃行为的相似程度

$$d(\mathbf{p},\mathbf{q}) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

L2距离

- 触发崩溃的谓词合成
  - 对最相似的Top-N个非崩溃行为进行详细追踪，从行为差异中提取崩溃条件
  - 生成描述崩溃条件的逻辑表达式（谓词）
- 报告生成
  - 为每个谓词分配一个相似性分数，并根据排名输出可能的故障定位列表



实验设计 数据资源

- 数据资源
  - 来自真实应用程序的60个漏洞
  - 涵盖11种漏洞类型
  - 均有开发者提供的实际补丁
- 评估标准
  - 采用人工检查的方式，验证谓词是否与修复点匹配
  - 谓词排名前50的任一谓词正确匹配了漏洞修复点，则认为成功
- 对比方法
  - 基于统计调试的方法：AURORA
  - 基于符号执行的方法：ARCUS

Target	CVE/Issue	Bug Type	# of Instructions
01 PHP-8865	CVE-2015-8865 [39]	Heap Overflow	9,809,095
02 PHP-6977	CVE-2019-6977 [41]	Heap Overflow	12,491,703
03 mruby-0525	CVE-2022-0525 [26]	Heap Overflow	3,134,273
04 Poppler-12293	CVE-2019-12293 [45]	Heap Overflow	125,813,540
05 SoX-11358	CVE-2017-11358 [52]	Heap Overflow	680,855
06 TinyCC-20375	CVE-2018-20375 [58]	Heap Overflow	614,608
07 mruby-46020	CVE-2021-46020 [25]	Null Dereference	2,243,674
08 PHP-7226	CVE-2013-7226 [42]	Type Confusion	9,956,549
09 PHP-0273	CVE-2015-0273 [43]	Type Confusion	9,516,412
10 libical-11706	CVE-2019-11706 [16]	Type Confusion	3,975,381
11 SoX-8356	CVE-2019-8356 [53]	Stack Overflow	141,771,429
12 mruby-181321	hackerone-181321 [29]	Use After Free	13,497,624
13 PHP-2386	CVE-2012-2386 [40]	Integer Overflow	9,285,250
14 Poppler-7310	CVE-2019-7310 [46]	Integer Overflow	16,506,068
15 SQLite-13434	CVE-2020-13434 [54]	Integer Overflow	1,331,697
16 SQLite-16168	CVE-2019-16168 [55]	Division by Zero	1,338,307
17 libbfd-8393	CVE-2017-8393 [14]	Global Overflow	539,511
18 readelf-9077	CVE-2019-9077 [49]	Heap Overflow	671,011
19 objdump-9746	CVE-2017-9746 [36]	Heap Overflow	6,411,352
20 tcpdump-16808	CVE-2017-16808 [57]	Heap Overflow	1,138,487
21 perl-17384	Issue-17384 [38]	Heap Overflow	1,314,300
22 patch-54558	Issue-54558 [10]	Heap Overflow	317,534
23 mruby-12248	CVE-2018-12248 [23]	Heap Overflow	17,393,851
24 nasm-16517	CVE-2018-16517 [31]	Null Dereference	955,041
25 mruby-185041	hackerone-185041 [27]	Type Confusion	13,420,945
26 Python-116286	hackerone-116286 [48]	Type Confusion	44,778,829
27 mruby-3947	Issue-3947 [28]	Uninitialized Var.	25,646,156
28 PHP-11038	CVE-2019-11038 [44]	Uninitialized Var.	11,683,900
29 Xpdf	N/A [60]	Uninitialized Var.	7,245,661
30 nm-21670	Issue-21670 [33]	Stack Overflow	515,784
31 mruby-10199	CVE-2018-10199 [30]	Use After Free	17,191,784
32 Lua-6706	CVE-2019-6706 [21]	Use After Free	960,676
33 nasm-8343	CVE-2019-8343 [32]	Use After Free	962,302
34 Sleuthkit	N/A [51]	Double Free	2,463,131
35 libzip-12858	CVE-2017-12858 [20]	Double Free	352,871
36 Python-5636	CVE-2016-5636 [47]	Integer Overflow	105,414,833
37 bash	N/A [6]	Integer Overflow	1,423,765
38 mruby-10191	CVE-2018-10191 [24]	Integer Overflow	28,965,409

实验设计 实验结果

- 性能对比结果
  - 与AURORA的对比
    - BENZENE
      - 成功数44/46
    - AURORA
      - 成功数29/46
  - 与ARCUS的对比
    - BENZENE
      - 成功数32/34
    - ARCUS
      - 成功数12/34
  - BoV策略有效

✓: Root Cause Found, ✗: Root Cause Not Found, ✱: Bag of Values Contributed, ↛: Unable to Proceed, ⊘: Out of Scope

Target (BoV)	Sanitizer	ΔRoot (%)	RCA BEN	RCA AUR	RCA ARC	RCA M‡	Target (BoV)	Sanitizer	ΔRoot (%)	RCA BEN	RCA AUR	RCA ARC	RCA M‡
01 PHP-8865 [39]	-	3,325 (<0.1%)	1	✗	⊘	✓	31 mruby-10199 [30]	ASAN	112,609 (0.7%)	5	25	✗	✓
02 PHP-6977 [41]	ASAN	132 (<0.1%)	1	✗	⊘	✓	32 Lua-6706 (✱) [21]	-	147 (<0.1%)	1	12	↛	✓
03 mruby-0525 (✱) [26]	-	-	2	✗	⊘	✗	33 nasm-8343 [32]	-	457,901 (47.5%)	2	2	↛	✓
04 Poppler-12293 [45]	-	1,287,909 (<0.1%)	9	✗	⊘	✓	34 Sleuthkit [51]	-	483 (<0.1%)	1	5	✗	✓
05 SoX-11358 [52]	ASAN	79,470 (11.6%)	40	✗	⊘	✓	35 libzip-12858 [20]	-	46,218 (13.0%)	9	1	↛	✓
06 TinyCC-20375 [58]	-	57 (<0.1%)	1	✗	⊘	-	36 Python-5636 [47]	-	828,077 (0.8%)	5	⊘	✗	✓
07 mruby-46020 (✱) [25]	-	7,249 (0.3%)	16	✗	⊘	✓	37 bash [6]	-	75,418 (5.3%)	2	5	↛	✓
08 PHP-7226 (✱) [42]	-	5,134 (<0.1%)	1	✗	⊘	✓	38 mruby-10191 [24]	-	3,251 (<0.1%)	3	35	✗	✓
09 PHP-0273 (✱) [43]	-	266 (<0.1%)	2	✗	⊘	✓	39 libpng-0597 [18]	-	4,471 (1.7%)	2	3	✓	✓
10 libical-11706 [16]	-	25,028 (0.6%)	19	✗	⊘	✓	40 jpegtoavi-1279 [13]	-	125,152 (37.6%)	1	⊘	✓	-
11 SoX-8356 [53]	-	4,194,431 (<0.1%)	2	✗	⊘	-	41 o3read-1288 (✱) [35]	-	10,859 (4.2%)	1	1	✓	✗
12 mruby-181321 [29]	-	2,578 (<0.1%)	11	29	✗	✓	42 autotrace-9167 [3]	ASAN	7,507 (0.6%)	5	(24)	✗	-
13 PHP-2386 [40]	-	736 (<0.1%)	3	✗	✗	✓	43 Redis-12326 [50]	-	8,514 (0.7%)	1	⊘	↛	✓
14 Poppler-7310 [46]	ASAN	0 (0.0%)	2	8	✗	✓	44 ftp-15705 [9]	-	19,094 (13.1%)	3	11	↛	-
15 SQLite-16168 [54]	-	15,526 (1.1%)	6	✗	⊘	✓	45 gif2png-5018 [8]	-	76,428 (24.7%)	1	⊘	✓	✓
16 SQLite-13434 [55]	-	79,416 (5.8%)	2	10	↛	✓	46 dmitry-7938 [7]	-	56,330 (10.4%)	1	⊘	↛	-
17 libbfd-8393 [14]	ASAN	7 (<0.1%)	1	19	⊘	✓	47 ntpq-12327 [34]	-	349,775 (29.2%)	1	⊘	✓	✓
18 readelf-9077 [49]	ASAN	917 (0.1%)	2	1	⊘	✓	48 libiec-18957 [17]	-	167 (<0.1%)	1	⊘	✓	✓
19 objdump-9746 [36]	ASAN	1,778,797 (27.7%)	3	3	⊘	✓	49 pdf-re-14267 (✱) [37]	-	181,576 (9.6%)	2	1	↛	✓
20 tcpdump-16808 [57]	ASAN	5,341 (0.5%)	2	3	⊘	✓	50 abc2mtex-1257 [2]	-	28,688 (10.7%)	1	✗	✓	✓
21 perl-17384 [38]	ASAN	30,139 (2.3%)	9	36	⊘	✓	51 abc2mtex-47254 [1]	-	2,944 (1.3%)	1	⊘	✓	✓
22 patch-54558 [10]	ASAN	18,955 (6.0%)	4	3	⊘	✓	52 MiniFtp-46807 [22]	-	778 (0.3%)	1	⊘	✓	-
23 mruby-12248 [23]	ASAN	1,249 (<0.1%)	6	1	⊘	✓	53 GM-11403 [12]	ASAN	0 (0.0%)	✗	⊘	↛	✗
24 nasm-16517 [31]	-	471,868 (49.4%)	1	15	⊘	✓	54 GM-14103 [11]	ASAN	70,009 (0.9%)	✗	⊘	↛	✗
25 mruby-185041 [27]	-	1,322 (<0.1%)	✗	29	⊘	✗	55 autotrace-9182 [5]	-	1,512,761 (0.8%)	1	(4)	↛	-
26 Python-116286 [48]	-	1,108 (<0.1%)	✗	✗	⊘	✗	56 libtiff-2025 [19]	-	61 (<0.1%)	3	3	↛	✓
27 mruby-3947 [28]	MSAN	0 (0.0%)	8	14	⊘	✓	57 libexif-2645 [15]	-	400 (0.1%)	1	7	↛	✓
28 PHP-11038 [44]	MSAN	1,766,918 (15.1%)	8	⊘	⊘	✓	58 autotrace-9186 [4]	-	794 (<0.1%)	1	(16)	✓	-
29 Xpdf (✱) [60]	ASAN	0 (0.0%)	1	✗	⊘	✓	59 typespeed-0105 [59]	-	752 (0.2%)	1	⊘	✓	✓
30 nm-21670 [33]	ASAN	0 (0.0%)	2	✗	⊘	✓	60 sudo-0809 [56]	-	339,050 (33.7%)	1	⊘	✓	✓

实验设计 实验结果

- 效率对比结果
  - 时间开销
    - BENZENE的整体时间开销更低，比AURORA快8.1倍，比ARCUS快1.1倍
  - 内存占用
    - BENZENE的内存占用远低于对比方法，是AURORA的1/9，是ARCUS的1/53.7

System	Time(sec)				Mem(MB)
	Pre+Mutation	Tracing	RCA	Total	
BENZENE	1,054	397	116	1,567	1,839
AURORA	10,643	1,621	497	12,762	16,858
Difference	10.0×	4.0×	4.2×	8.1×	9.1×
BENZENE	549	143	17	710	444
ARCUS	-	-	845	845	23,858
Difference	-	-	-	1.1×	53.7×

特点总结与未来展望

特点总结与工作展望

- 特点总结
  - ARCUS
    - 硬件级追踪，高效的崩溃路径还原，生成修复建议
    - 仅支持有限个漏洞类型
  - BENZENE
    - 欠约束状态变异快速生成非崩溃行为
    - 统计分析与符号执行的结合
- 未来展望
  - 故障定位的非唯一性
    - 若“根因点”定义为“需要修复的位置”，可能会出现多个候选修复点
  - 现有方法缺乏单独衡量每个步骤独立性能的实验（消融实验）

参考文献

[1] Yagemann C, Pruett M, Chung S P, et al. {ARCUS}: symbolic root cause analysis of exploits in production systems. 30th USENIX Security Symposium (USENIX Security 21)[C]. California, CA: USENIX Association, 2021: 1989-2006.

[2] Park Y, Lee H, Jung J, et al. Benzene: A practical root cause analysis system with an under-constrained state mutation. 2024 IEEE Symposium on Security and Privacy (SP)[C]. Piscataway, NJ: IEEE, 2024: 1865-1883.

[3] Nishimura K, Sugiyama Y, Koike Y, et al. RCABench: Open Benchmarking Platform for Root Cause Analysis. 2023. arXiv. https://doi.org/10.48550/arXiv.2303.05029

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！