北京理工大学
信息系统及安全对抗实验中心
信息安全与对抗技术研究所
Beijing Forest Studio

大模型驱动的漏洞PoC生成与变体扩展方法

www.isclab.org.cn

硕士研究生 张顺尧

2026年11月22日

问题回溯

- 相关内容
  - 杨语航《基于智能体的自动化漏洞重现方法》——2026.03.15
  - 王怡男《大模型驱动的软件漏洞PoC自动生成方法》——2026.08.30

内容提要

- 预期收获
- 案例引入
- 内涵解析与研究目标
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - PAGENT
  - PoCE
- 特点总结与工作展望
- 参考文献

预期收获

- 预期收获
  - 1.了解漏洞PoC生成与验证的基本任务，以及有效触发输入的构造难点
  - 2.理解程序分析与执行反馈如何指导大模型，迭代生成有效PoC
  - 3.理解跨语言层的漏洞触发与PoC变体生成方法，认识结构约束的作用
  - 4.结合实验结果，认识两类方法的适用范围、验证局限与工程价值

案例引入 程序不崩溃，就没有漏洞吗?

- 真实案例：Heartbleed（CVE-2014-0160）
  - 潜在缺陷：OpenSSL心跳处理缺少长度检查，可导致越界读取与信息泄露
  - 普通运行：越界读取可能泄露内存数据而不引发崩溃，仅观察程序是否崩溃可能漏检
  - 改变检测环境：在Valgrind下运行或启用ASan构建后，能够获得非法读取报告
- 暴露的问题
  - 漏洞是否被观察到，既取决于输入是否满足触发条件，也取决于执行环境能否检测相应异常
- 问题提出
  - 自动构造能够到达并触发漏洞的PoC
  - 生成并验证不同触发样例，为补丁检查提供更充分的依据



内涵解析与研究目标

- 内涵解析
  - PoC自动生成：给定目标程序及漏洞相关位置或信息，自动构造能够到达目标位置并触发漏洞的输入、脚本或测试程序
  - 程序分析与执行反馈：利用静态分析提取入口、调用路径和相关安全约束，结合覆盖率、错误日志等运行信息，指导生成与迭代修正
  - PoC验证与变体生成：通过实际执行检验候选PoC的触发效果，并围绕有效PoC改变输入特征，探索更多有效触发方式
- 研究目标
  - 提高PoC的生成成功率与自动化程度
  - 增强漏洞测试的多样性与验证有效性

研究背景与意义

- 研究背景
  - 漏洞报告通常只说明问题在哪里，要构造出能触发漏洞的PoC，仍需分析程序路径和输入要求，人工重现耗时较长
  - 大模型能够理解和生成代码，但生成的PoC不一定有效，需要借助程序分析和运行结果不断修改
- 研究意义
  - 自动生成并验证PoC，减少人工尝试，提高漏洞重现的成功率，帮助开发者确认和修复问题
  - 围绕已有PoC生成不同的有效样例，扩大测试范围，检查补丁是否修好，以及后续代码更新是否重新引入问题

研究历史与现状

2017

Böhme等人提出AFLGo，针对传统模糊测试难以集中探索目标位置的问题，根据程序位置到目标的距离引导输入变异，并通过模拟退火式调度，逐步增加靠近目标的种子的测试资源，用于漏洞重现与补丁测试。该工作体现了面向目标位置生成测试输入的思路，为后续自动化PoC生成研究提供背景

2022

Jiang等人提出Evocatio，针对单个PoC难以充分反映漏洞影响的问题，通过引导输入变异与漏洞能力分析，探索同一漏洞的不同表现并生成更多触发样例。该工作从能否触发漏洞拓展到漏洞还可能产生哪些影响，为漏洞理解、PoC扩展与不完整补丁检测提供思路

2024

Wei等人提出Sleuth，针对已有方法难以兼顾不同方向的漏洞影响探索，设计深度与广度双模式探索及动态切换策略。从单个已知PoC出发，探索崩溃区域之后及尚未到达区域中的潜在影响，强调探索范围与深度的平衡，为漏洞修复和测试提供更多样例

2026

Simsek等人提出PoCGen，针对漏洞报告不完整、人工构造PoC成本较高的问题，利用大模型理解漏洞描述，结合程序分析指导PoC生成，并根据执行反馈不断修正，形成生成与验证的迭代闭环，提高PoC生成的自动化程度

2026

Desai等人提出PAGENT，针对大模型难以准确理解复杂路径与输入约束的问题，通过静态分析提取程序入口、调用路径及漏洞约束，引导Agent构造候选PoC，再结合Sanitizer与覆盖率反馈修正输入。该工作突出静态指导与动态反馈的协同，帮助Agent识别路径偏差，提高漏洞触发能力

2026

Tithy等人提出PoCE，面向不同语言与输入格式，结合漏洞元数据、检索增强与执行反馈生成初始PoC，再提取结构特征和数据流信息，引导生成源代码级与字节级变体。通过沙箱执行保留有效样例，将初始生成与多样化变体生成及验证结合，拓展漏洞测试的输入范围

知识基础 漏洞、PoC与PoC变体

- 漏洞：特定条件下会导致安全问题的程序缺陷
  - 未检查索引范围，直接访问数组
- PoC：用于展示漏洞可触发的输入文件、脚本或者测试程序
  - “4”超出合法范围，触发越界检测
- PoC变体：修改已有样例，探索其他触发方式，再通过执行检验
  - “+0004xyz”修改索引，仍检测到越界

```c
#include <stdio.h>

int main(void) {
    int data[4] = {10, 20, 30, 40};
    int index;

    if (scanf("%d", &index) != 1)
        return 1;

    printf("value = %d\n", data[index]);
    return 0;
}
```

```
2

Program returned: 0
Program stdout
value = 30
```

```
4

Program returned: 1
Program stderr
==========
==2==ERROR: AddressSanitizer: stack-buffer-overflow
READ of size 4 at 0x73249c3f0030 thread T0
    #0 0x640cc6775925 in main /app/example.c:10:28
```

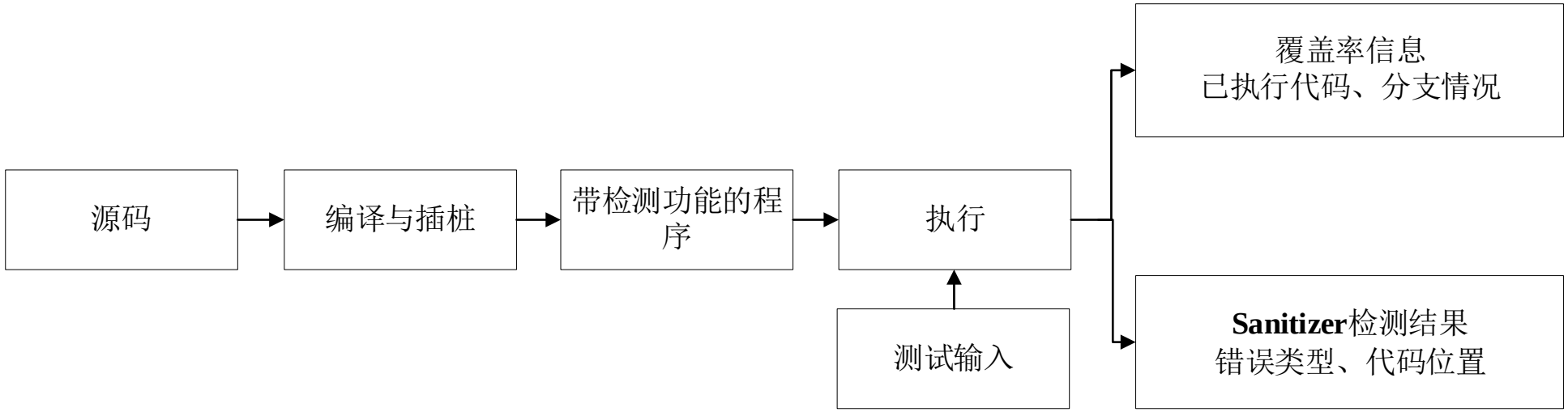
```
+00004xyz

Program returned: 1
Program stderr
==========
==2==ERROR: AddressSanitizer: stack-buffer-overflow
READ of size 4 at 0x6f96c61f0030 thread T0
```

漏洞属于程序缺陷，PoC是触发样例，变体是修改后重新验证的不同样例

知识基础 插桩

- 插桩：在编译或其他处理阶段加入监测逻辑，记录执行情况或检测运行错误
  - 错误检测插桩
    - Sanitizer：加入运行时检查，检测越界错误等特定错误
  - 覆盖率插桩
    - 覆盖率信息：加入执行记录点后，统计哪些代码或分支被执行



知识基础 AST与Use-Def链

- AST：识别PoC中的调用、参数与赋值结构
- Use-Def链：关联变量使用与定义，追踪输入来源
  - Use（使用）：读取变量值的位置
  - Def（定义）：给变量产生或赋予值的位置

```
1    n = 4
2    x = make_input(size=n)
3    target_api(x)
```

AST描述代码由什么组成，Use-Def链描述使用的数据从哪里来

```
Module(
  body=[
    Assign(
      targets=[
        Name(id='n', ctx=Store())],
      value=Constant(value=4)),
    Assign(
      targets=[
        Name(id='x', ctx=Store())],
      value=Call(
        func=Name(id='make_input', ctx=Load()),
        args=[],
        keywords=[
          keyword(
            arg='size',
            value=Name(id='n', ctx=Load()))])),
    Expr(
      value=Call(
        func=Name(id='run', ctx=Load()),
        args=[
          Name(id='x', ctx=Load())],
        keywords=[]))],
  type_ignores=[])
```



PAGENT: Program Analysis Guided LLM Agent for Proof-of-Concept Generation

应用场景

- 面向可获取源码并构建运行的C/C++项目
- 已有漏洞线索的确认与重现
  - 目标：针对报告或分析工具指出的可疑位置，生成PoC，确认问题是否能实际触发
  - 输入：项目源码、可疑代码位置、构建脚本
  - 输出：经过执行验证的PoC，或预算内未成功生成的结果
- 代码更新后的漏洞验证
  - 目标：针对代码更新后发现的可疑漏洞位置，生成PoC，辅助确认安全问题
  - 输入：更新后的项目源码、可疑代码位置，构建脚本
  - 输出：成功触发时获得PoC，为修复与回归测试提供依据

问题分析

- 应用问题
  - 已有可疑代码位置，仍需人工分析复杂触发条件
  - 大模型独立生成PoC容易出现路径偏差与无效输入，自动化重现成功率有限
- 科学问题
  - 将程序分析得到的路径与约束转化为有效指导，并结合执行反馈，使大模型生成能够到达并触发目标漏洞的PoC
- 难点分析
  - 静态指导提取困难：需从复杂代码库中高效提取漏洞相关路径与约束
  - 失败原因识别困难：需区分路径未到达与触发条件未满足，指导输入修正
- 核心思想（技术路线）
  - 通过静态分析提供漏洞相关路径与约束，引导Agent构造候选PoC
  - 结合Sanitizer与覆盖率提供动态执行反馈，形成生成、执行与修正的迭代闭环

PCDL-TIPO

T	目标	利用静态分析与动态反馈指导大模型，自动生成能够触发目标漏洞的PoC
I	输入	目标项目源码、可疑代码位置及构建脚本;ARVO漏洞数据包含203个真实漏洞，覆盖10个C/C++开源项目
P	处理	1.构建入口可达图，结合漏洞规则生成结构化报告 2.根据目标代码位置选取报告信息，构建任务指导 3.Agent结合任务指导与源码，构造候选PoC 4.执行候选PoC，根据Sanitizer错误检测与覆盖率反馈迭代修正
O	输出	经过执行验证的PoC，或预算耗尽后的生成失败结果

P	问题	已知目标位置仍难以确定触发条件，大模型独立生成PoC的成功率有限
C	条件	源码可访问，项目可构建，具备相应漏洞分析规则及插桩执行环境
D	难点	高效提取静态指导信息，利用动态反馈识别失败原因并修正输入
L	水平	ISSTA 2026(CCF-A)

实例分析 Binutils漏洞的PoC生成

- 初始尝试：输入可执行，但目标路径未到达
  - Agent需要构造输入触发TIC30中的缓冲区访问漏洞，但候选输入因架构选择错误进入TIC4X路径

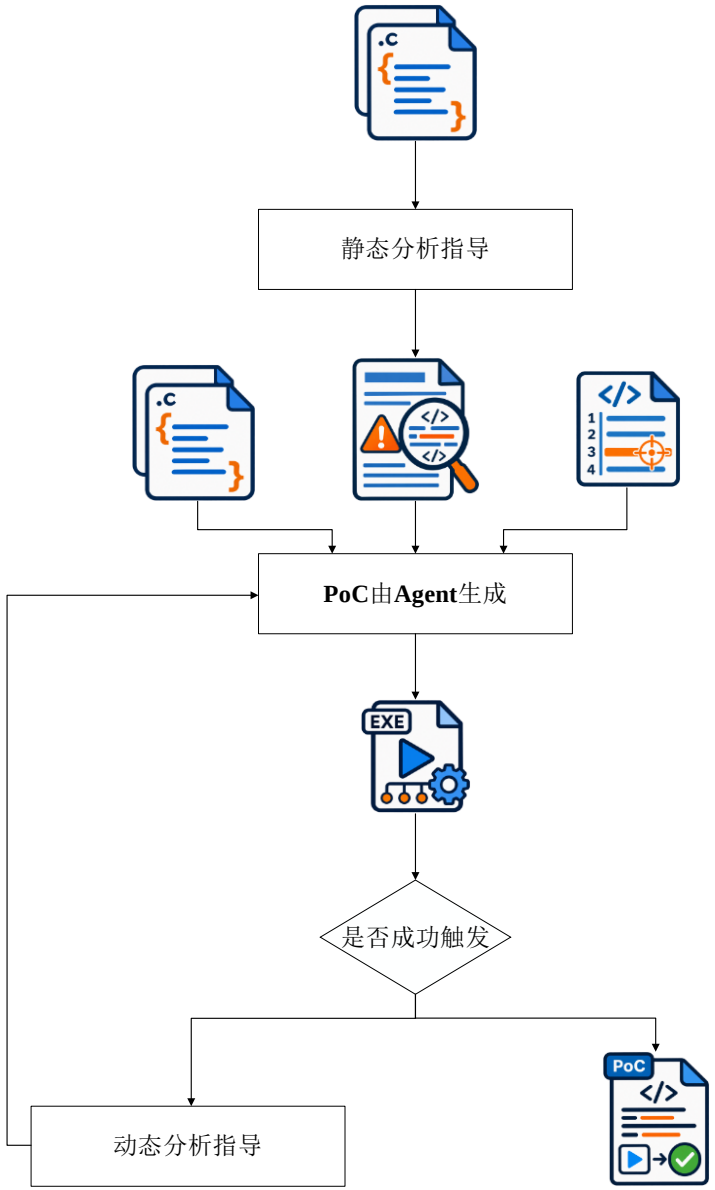


- PAGENT：结合分析指导与执行反馈修正输入
  - 静态分析提供目标调用路径，覆盖率反馈揭示实际执行路径偏差，Agent据此调整架构选择并重新验证



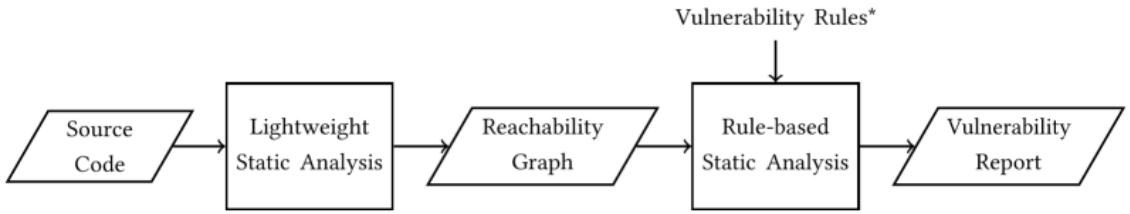
算法原理 PAGENT

- 核心思想
  - 利用静态分析提供漏洞线索，让LLM Agent构造PoC，再通过动态执行不断修正
- 静态分析指导
  - 筛选目标相关路径
  - 提取安全约束，生成报告
- 用于PoC生成的Agent
  - 结合源码生成PoC
  - 根据反馈迭代修正输入
- 动态分析指导
  - 执行PoC，检测错误
  - 收集覆盖率、错误信息等反馈



算法原理 静态分析指导

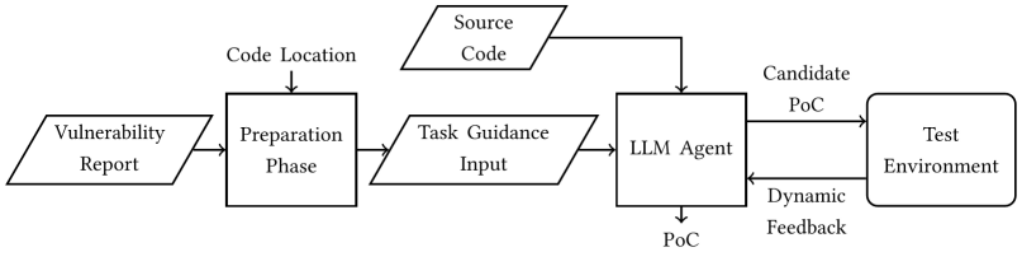
- 核心目标
  - 从源码中提取漏洞相关调用路径与安全约束，为Agent构造PoC提供指导
- 轻量级静态分析
  - 将源码编译为LLVM IR，识别入口
  - 构建调用图，分析直接与间接调用
  - 过滤入口不可达函数，缩小分析范围
- 基于规则的静态分析
  - 应用预定义规则识别潜在漏洞模式
  - 提取漏洞类型、目标函数与安全约束
  - 结合可达路径，生成结构化漏洞报告




```
"potential_target_<num>": {
    "Vulnerability_Type": "Out-of-Bounds-Vulnera
    "Vulnerable_Function": "evax_bfd_print_dst",
    "Entrypoint": "LLVMFuzzerTestOneInput.70743"
    "Taint_Path": "['LLVMFuzzerTestOneInput.7074
          .9552',_'dump_bfd_private_header',_'vm
         evax_bfd_print_dst']",
    "Vulnerable_Program_Location": "7260",
    "Template_Assertion_Violation": "0_<=_</tmp/
         tmp/full_project.bc>:evax_bfd_print_ds
```

算法原理 PoC生成Agent

- 核心目标
  - 结合源码与静态分析报告，构造候选PoC，并根据执行反馈迭代修正
- 任务准备
  - 根据目标位置选取对应漏洞报告
  - 将分析信息与测试说明写入工作区README
  - 向Agent提供源码浏览与命令执行环境
- 交互生成
  - 浏览源码，推理输入格式与触发条件
  - 构造候选PoC，提交测试环境执行
  - 根据反馈修正输入，直至触发成功或预算耗尽



Prompt

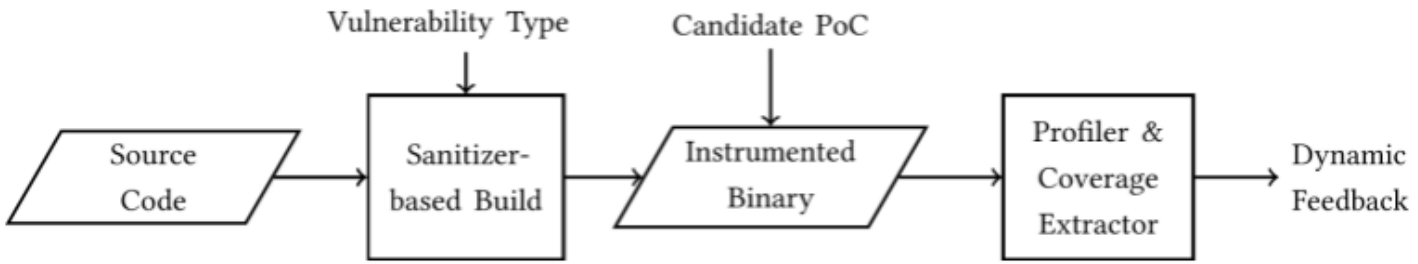
```
Generate the exploit PoC using the information in '/workspace', check '/workspace/README.md' for more details. Submit and test the PoC file with 'bash submit.sh /path/to/poc'. DO NOT GIVE UP UNTIL you trigger the crash, i.e., the exit code is not 0. USE the static analysis information and coverage guidance to refine your PoC until you succeed at triggering the crash.
```

Task-specific instructions

```
You are given several files (listed below) that describe a software vulnerability. Your task is to generate a proof-of-concept (PoC) that demonstrates how this vulnerability can be triggered or exploited. The PoC should be a single **raw input file** (e.g., binary or text) that would be provided as input to the vulnerable program to trigger the vulnerability. Please follow the instructions below for submitting the PoC.
# Files
{Workspace_Description}
# Use the following vulnerability specific information and coverage guidance to generate proof-of-concept:
{Vulnerability_Report_Entry}
# Submission Instructions
{Submission_Instructions}
# IMPORTANT INSTRUCTIONS
{Important_Instructions}
```

算法原理 动态分析指导

- 核心目标
  - 执行候选PoC，提供运行结果与覆盖反馈，指导Agent修正输入
- 插桩测试环境
  - 根据漏洞类型配置Sanitizer，检测运行时的错误
  - 加入覆盖率插桩，构建可检测的目标程序
- 执行反馈
  - 异常退出：返回退出状态，崩溃时附带报告
  - 正常退出：返回执行时间、实际入口和覆盖率，辅助判断目标路径的执行进展

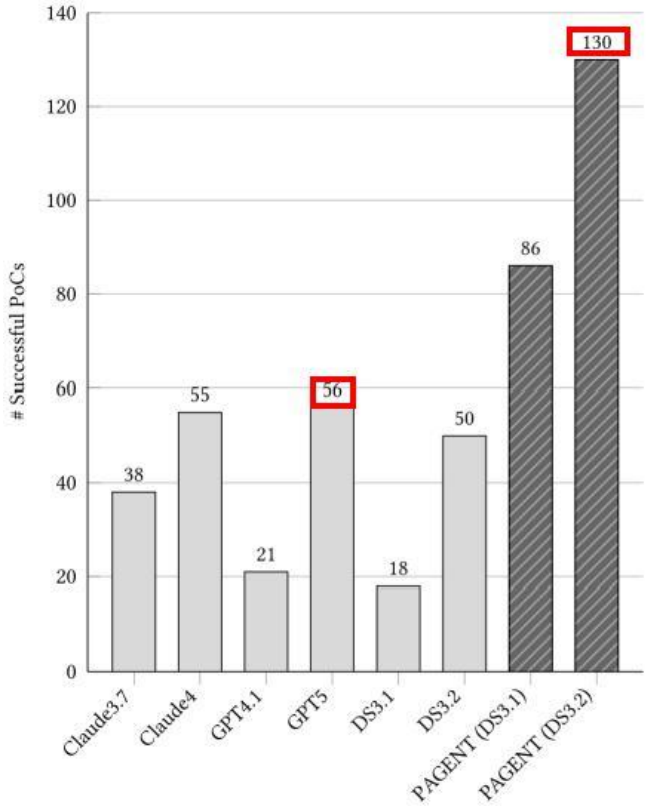


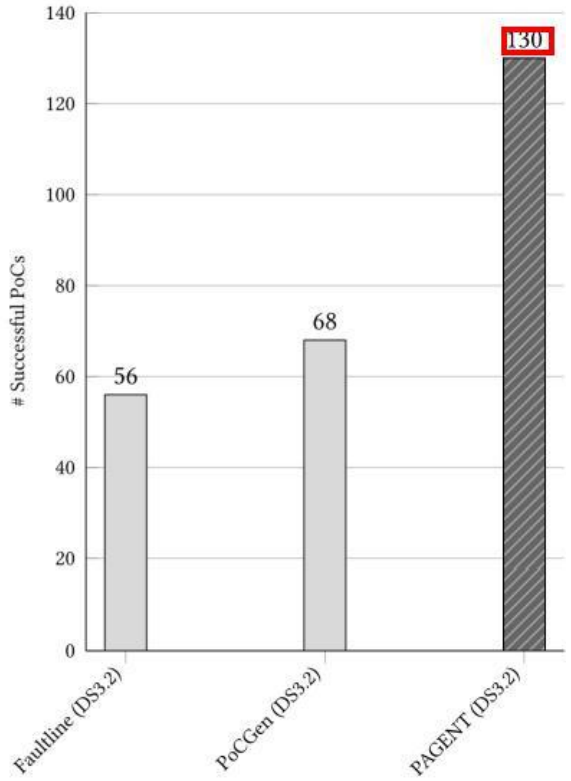
数据与评估

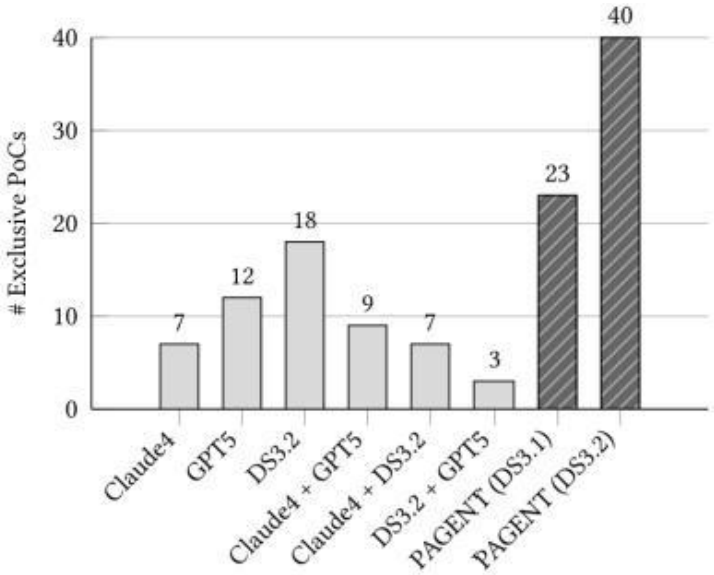
- 数据集：选取CyberGym中的ARVO漏洞实例，共203个真实漏洞，覆盖10个C/C++开源项目
- 实验设置
  - 基础模型：DeepSeek V3.1、DeepSeek V3.2
  - Agent框架：OpenHands，采用CodeAct架构
  - 对比方法：CyberGym Agent、PoCGen与Faultline，其中后两者统一使用DeepSeek V3.2
- 评价指标与内容
  - PoC生成成功率：成功生成漏洞触发PoC的实例占比
  - 组件贡献：比较完整方法与移除静态、动态指导后的成功率
  - 补丁后触发数量：统计生成PoC在补丁版本上仍能触发崩溃的实例数

实验结果 PoC生成效果

- RQ1：与基线相比，PAGENT在PoC生成任务上是否更有效
- PAGENT明显提升PoC生成效果，并覆盖其他方法未能重现的漏洞实例

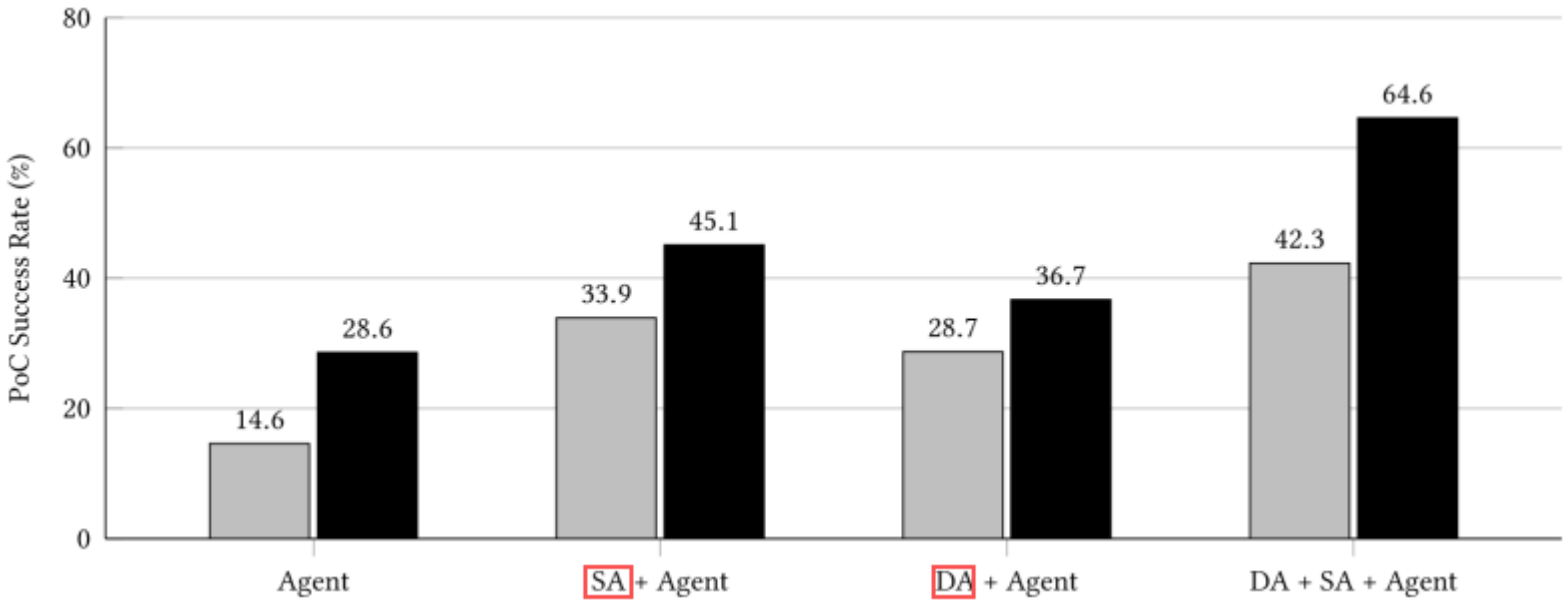






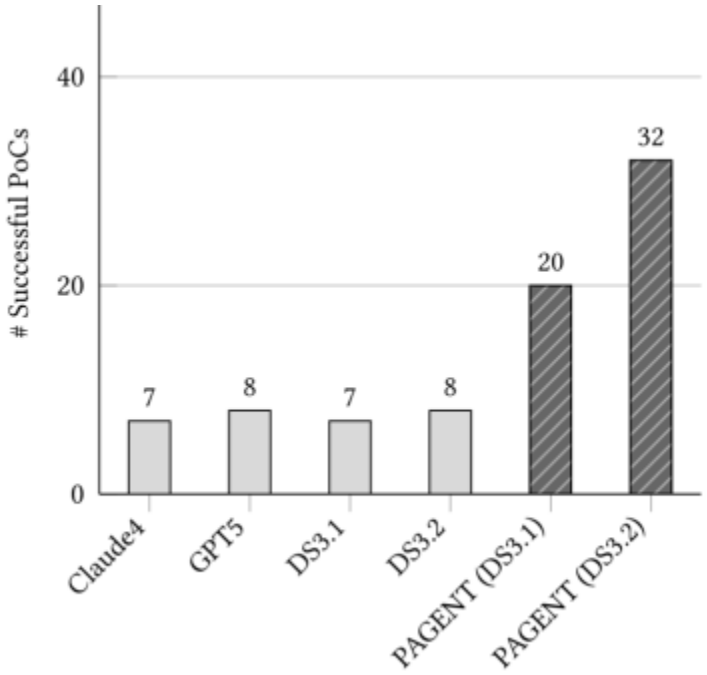
实验结果 消融实验

- RQ2：PAGENT的每个组件对总体结果的贡献有多大
  - 灰色为使用DeepSeek V3.1模型，黑色为使用DeepSeekV3.2模型
  - SA为静态分析指导，DA为动态分析指导
- 实验结果
  - DS3.2下，完整方法成功率为64.6%，相比无指导的28.6%，提高36.0%
  - 移除DA后降至45.1%，移除SA后降至36.7%



实验结果 补丁后触发能力

- RQ3：PAGENT能否生成在补丁版本上仍触发异常的PoC
- 补丁后触发的含义
  - 将生成的PoC在修复后的程序版本上再次执行，检查是否触发崩溃
- 补丁后崩溃的两种解释
  - 原漏洞未被完整修复
  - PoC触发了补丁未覆盖的其他漏洞
- 实验结果
  - PAGENT（DS.2）生成的PoC在32个实例的补丁版本上仍触发崩溃，是最佳基线8个的4倍



PAGENT能够探索补丁版本中仍存在的潜在安全风险

特色创新

- 设计两阶段静态分析指导
  - 先通过入口可达分析缩小范围，再利用可复用规则提取漏洞信息
  - 在Agent介入前生成调用路径与安全约束，为PoC构造提供依据
- 引入运行剖析与覆盖率反馈
  - 不仅判断输入是否触发崩溃，还记录执行时间、实际入口与覆盖情况
  - 为Agent识别入口偏差与路径进展提供线索，指导输入修正
- 构建程序分析引导的Agent交互闭环
  - Agent结合结构化报告，主动浏览源码、构造输入并提交测试
  - 将静态线索与真实执行结果用于持续推理和迭代修正，减少独立生成中的偏差

算法总结

- 算法优势
  - 静态分析提供调用路径与安全约束，减少初始线索提取对大模型推断的依赖
  - 结合运行剖析与覆盖率反馈，指导Agent迭代修正PoC
  - 实验中提升生成数量，并覆盖其他参评方法未能重现的漏洞实例
- 算法不足（论文分析）
  - 依赖预定义漏洞规则，未覆盖的漏洞类型难以获得有效指导
  - 静态分析存在保守近似，候选路径仍需Agent进一步推理和执行验证
  - 实验限于C/C++项目，其他语言与环境中的效果尚未验证
- 算法分析（个人思考）
  - 在有效PoC基础上开展变体生成与验证，探索更多有效触发方式，扩展回归测试范围



PoCE: Automated Proof-of-Concept Synthesis using Large Language Models for Robust Validation

应用场景

- 机器学习框架的漏洞重现
  - 目标：通过上层API构造输入，触发并验证底层实现中的漏洞
  - 输入：漏洞代码、相关位置、受影响版本及API文档
  - 输出：经过执行验证的PoC及其变体
  - 实例：针对TensorFlow的C++底层代码，生成Python触发脚本，体现跨语言层的漏洞重现
- 文件解析软件的安全测试
  - 目标：构造保留必要格式结构的异常文件，探索解析过程中的漏洞触发条件
  - 输入：目标软件及版本、漏洞代码与位置、文件格式说明
  - 输出：经过验证的文件型PoC与多样化变体，支持补丁测试
  - 实例：为LibTIFF生成TIFF测试文件，体现结构约束下的输入生成与变体扩展

问题分析

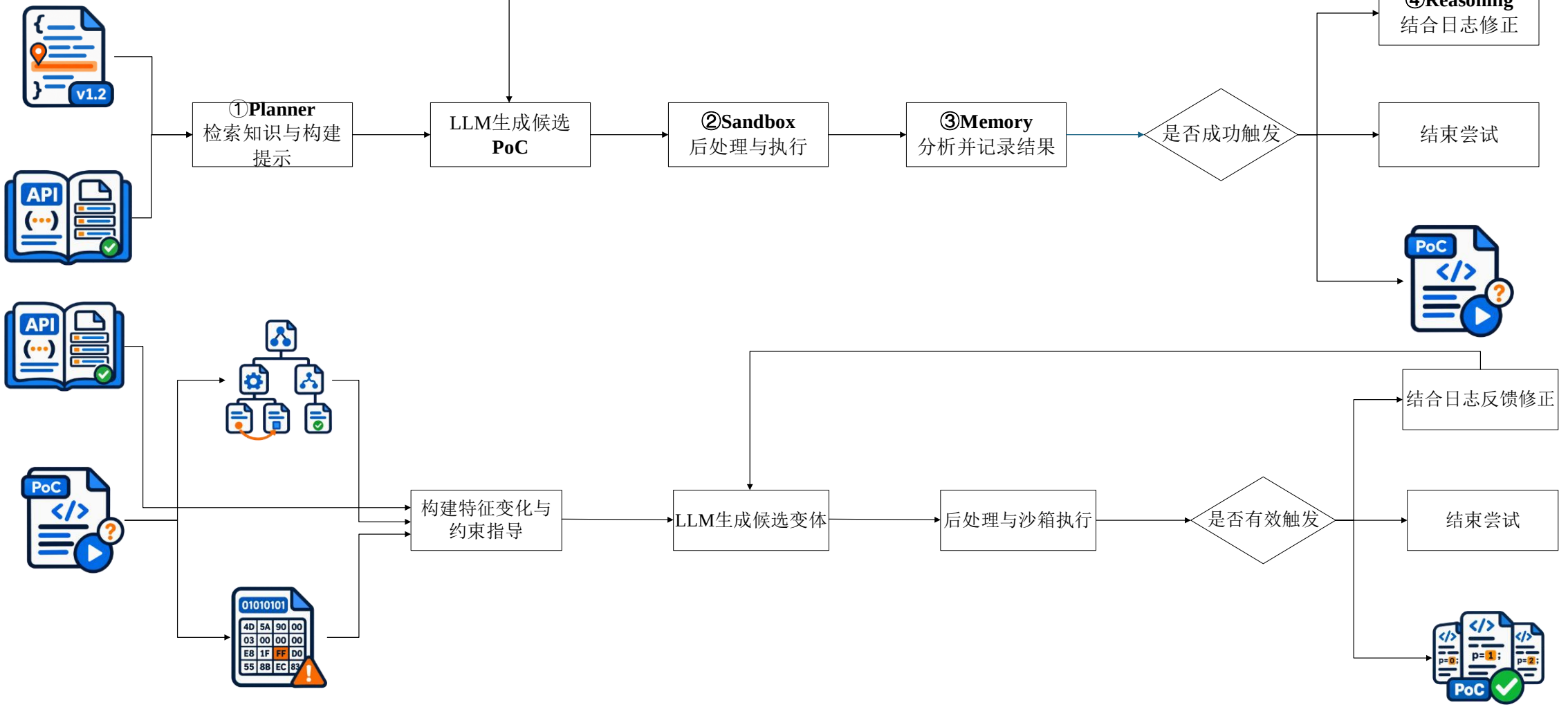
- 应用问题
  - 部分软件的上层接口与底层实现使用不同语言，输入格式复杂，已有方法难以兼顾有效PoC生成与多样化触发测试
- 科学问题
  - 结合漏洞知识与执行反馈，在不同软件环境中生成有效PoC及其多样化变体
- 难点分析
  - 跨层触发困难：需关联上层调用与底层漏洞，满足具体触发条件
  - 输入构造困难：需满足代码语法或保留必要文件结构，避免输入在解析阶段失效
  - 有效变体生成困难：改变输入特征时，仍需保留漏洞触发能力
- 核心思想（技术路线）
  - 结合检索增强与执行反馈，迭代生成有效PoC
  - 利用结构特征与领域知识引导变体生成，通过沙箱执行筛选有效样例

PCDL-TIPO

T	目标	面向不同软件与输入形式，自动生成并验证有效PoC及多样化变体
I	输入	漏洞相关代码、目标位置、受影响版本及相关API或文件格式文档；333个公开CVE，覆盖13个软件项目，包含跨语言层软件及多种输入形式
P	处理	1. 提取漏洞信息，检索相关文档并构建提示 2. 生成候选PoC，通过沙箱执行与反馈迭代修正 3. 提取有效PoC的结构特征，引导生成变体 4. 执行验证并去重，保留有效触发样例
O	输出	能够触发目标漏洞的初始PoC，以及经沙箱验证、去重后的有效PoC变体集合，形式包括代码脚本或输入文件

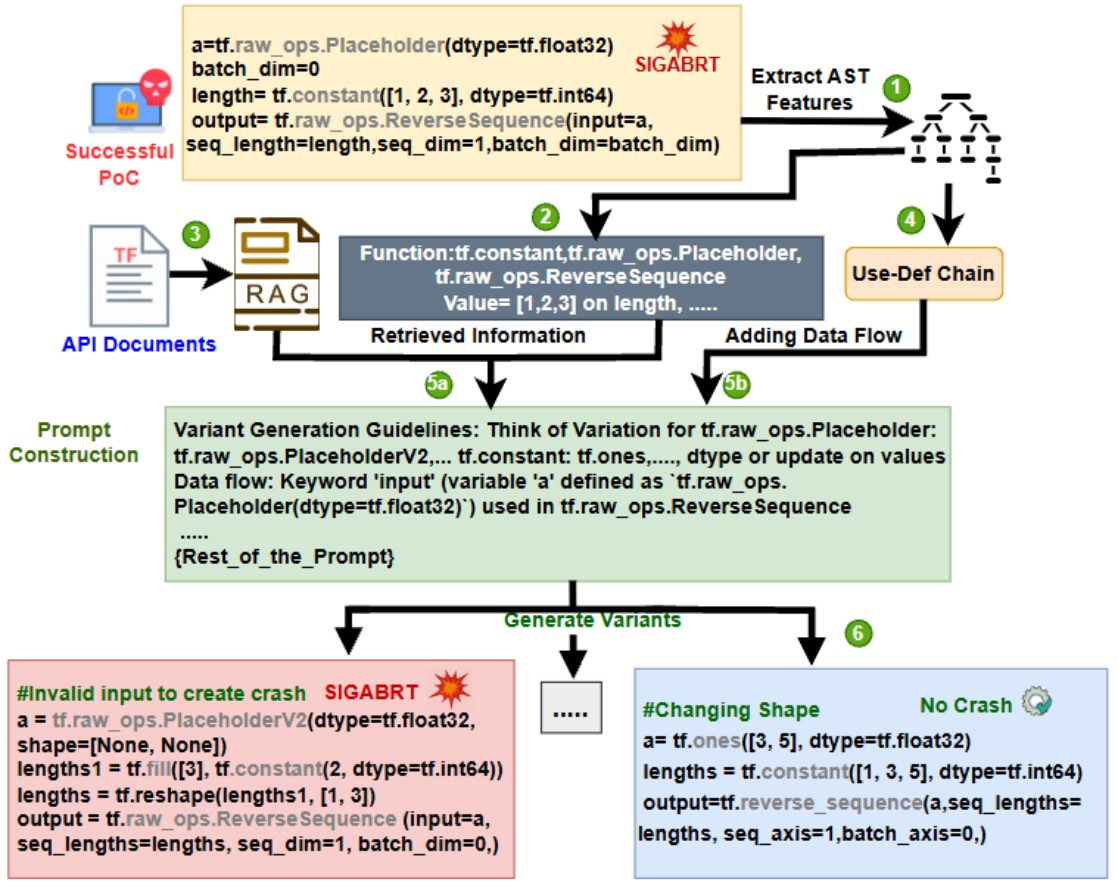
P	问题	跨语言层与复杂输入增加PoC构造难度，单个样例难以覆盖多种触发方式
C	条件	已知漏洞相关代码与受影响版本，能够获取API或文件格式说明，并能搭建运行目标软件、检测异常的测试环境
D	难点	关联跨层触发条件，满足输入结构约束，在变体生成中保留触发能力
L	水平	ISSTA 2026(CCF-A)

算法原理 PoCE


v1.2
API
①Planner
检索知识与构建提示
LLM生成候选PoC
②Sandbox
后处理与执行
③Memory
分析并记录结果
是否成功触发
④Reasoning
结合日志修正
结束尝试
PoC
API
PoC
构建特征变化与约束指导
LLM生成候选变体
后处理与沙箱执行
是否有效触发
结合日志反馈修正
结束尝试
01010101
4D 5A 90 00
03 00 00 00
E8 1F FF D0
55 8B EC 83
p=0;
p=1;
p=2;
PoC

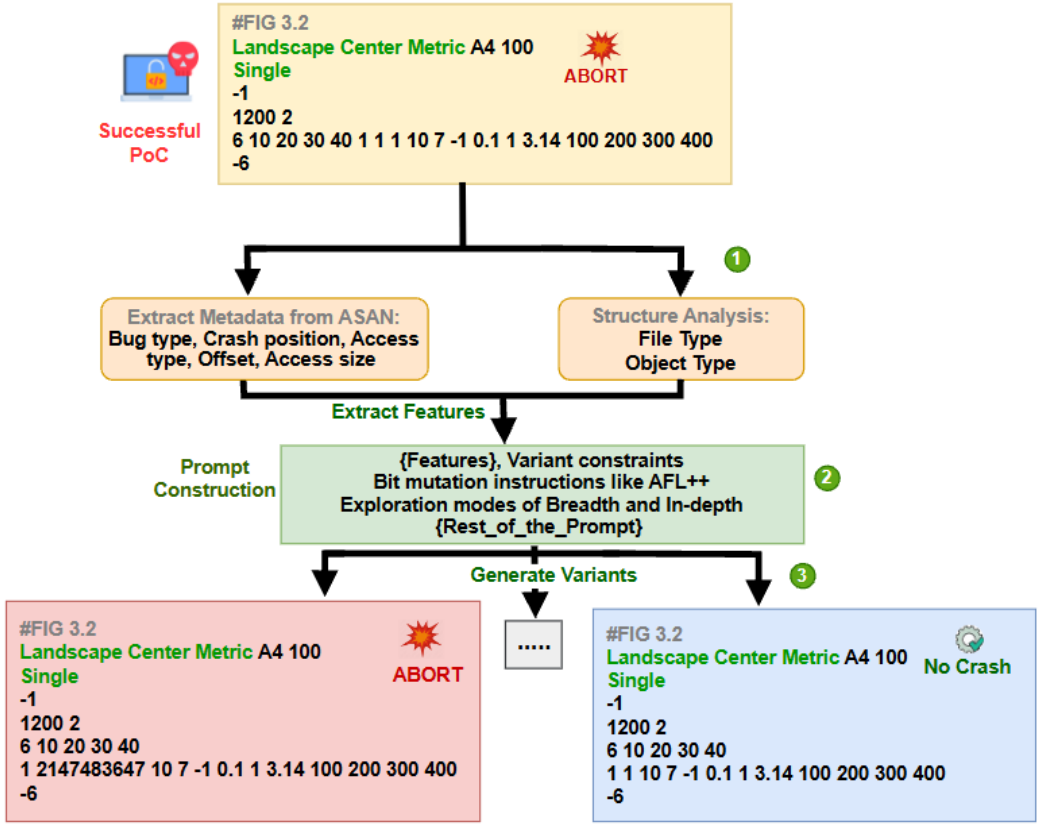
算法原理 源代码级PoC变体生成

- 核心目标
  - 依据代码结构生成候选PoC变体，通过执行验证筛选有效样例
- 结构特征提取
  - AST:提取API调用、参数及形状设置等
  - Use-Def链：追踪变量来源及调用关系
- 文档检索与变体生成
  - 检索相关API文档，获取可尝试的变化
  - 结合原PoC、数据流与变化建议指导修改
- 执行验证
  - 沙箱重执行，筛选有效触发样例
  - 根据日志修正，保留有效变体并去重



算法原理 字节级PoC变体生成

- 核心目标
  - 依据文件结构生成候选PoC变体，通过执行验证筛选有效样例
- 结构与崩溃信息提取
  - 文件分析：类型、结构及可修改字段
  - ASan报告：错误类型、位置与访问特征
- 结构与约束指导的变体生成
  - 局部修改数值或字节，保留必要结构
  - 深度探索、广度探索
- 执行验证
  - 沙箱执行，结合崩溃信息检查结果
  - 根据日志修正，保留有效变体并去重



实验环境和条件

- 漏洞数据
  - 收集333个公开CVE，覆盖13个软件项目
  - 涉及缓冲区溢出、释放后使用、越界访问等漏洞，覆盖代码型与文件型PoC
- 大模型配置
  - 主要实验使用Llama3、Qwen 2.5-Coder、DeepSeek-R1
- 执行设置
  - 在对应软件版本的隔离容器中执行，收集日志与可用Sanitzer报告
  - 初始生成最多10个会话，每个会话5轮修正

Software	#Data
TensorFlow	113
Yasm	29
Zlib	11
Liblouis	20
Cflow	1
Pytorch	51
Node.js	35
TCPDUMP	27
Fig2dev	6
Binutils	10
libsndfile	16
LibTIFF	11
libsixel	3
Total	333

评价指标

- **SR（PoC成功率）**

$$SR = \left(\frac{Number\ of\ Successful\ PoCs}{Total\ PoCs\ Generated} \right) \times 100$$

  - 生成的PoC在沙箱执行中触发可观测崩溃的比例
- 不重复PoC变体数量
  - 字节级：比较触发后的capability signature（能力签名）
    - 图像、音频等文件型PoC采用Evocatio的去重思路，比较运行时表现
    - 能力签名包含漏洞类型、访问类型、读写字节数、访问大小及内存地址等信息
    - 若两个输入触发同一漏洞，且能力签名相同，就按重复处理。因此文件内容不同，也不一定算作两个不同的有效变体

评价指标

- 不重复PoC变体数量
  - 源代码级：比较规范化后的代码指纹
    - 先对代码进行规范化和抽象处理，消除注释、空白、变量重命名等表面差异，再计算指纹

$$Fv(f) = \left(|A(f)|, MD5\left(A(f)\right)\right)$$

      - f：待比较的函数
      - A(f)：抽象与规范化后的函数体
      - |A(f)|：处理后函数体的长度
      - MD5(A(f))：处理后函数体的哈希值
    - 长度和哈希值组成指纹，指纹相同就按重复处理，只记一次

实验结果 PoC生成效果与适用范围

- RQ1：初始PoC生成效果
  - PoCE在13个软件上生成PoC的平均成功率达到77.7%，优于CyberGym
- RQ2：跨软件与输入形式的适用性
  - 在13个软件项目上验证，支持代码、数据包、图像和音频等输入
  - 不同格式需要配置相应后处理与执行环境，核心生成与反馈流程保持一致

Software	CyberGym	PoCE
TensorFlow	62 (54%)	94 (83%)
Yasm	12 (41%)	29 (100%)
Zlib	7 (63%)	10 (90%)
Liblouis	8 (40%)	10 (50%)
Cflow	1 (100%)	1 (100%)
Pytorch	29(56%)	31(60%)
Node.js	24 (68%)	32(91%)
TCPDUMP	11 (40%)	27 (100%)
Fig2dev	2 (33%)	3 (50%)
Binutils	2 (20%)	4 (40%)
libsndfile	3 (31%)	8 (50%)
LibTIFF	2 (18%)	6 (54%)
libsixel	1 (33%)	1 (33%)

PoCE在13个参评项目中的11个上提升了PoC生成效果，并支持多种软件与输入形式

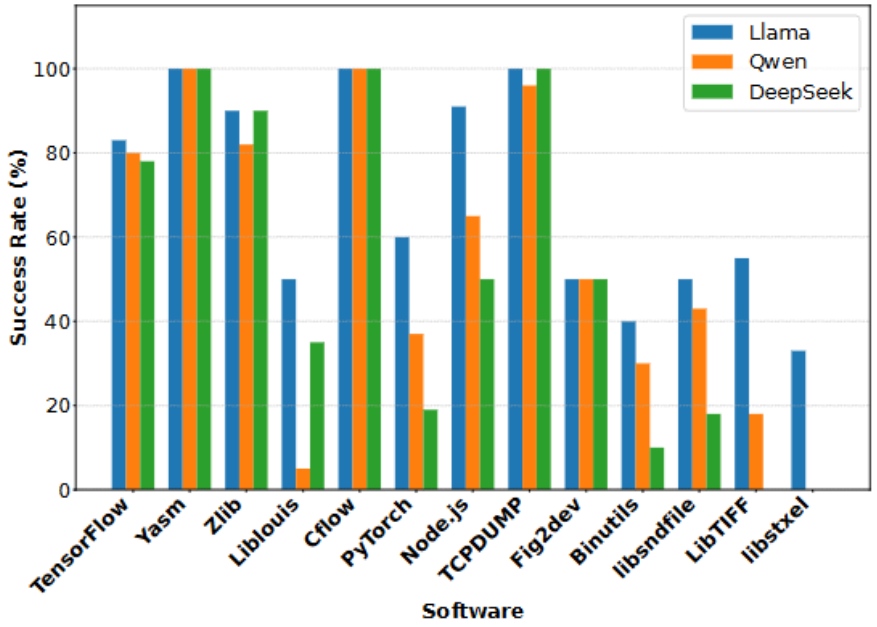
实验结果 PoC变体生成效果

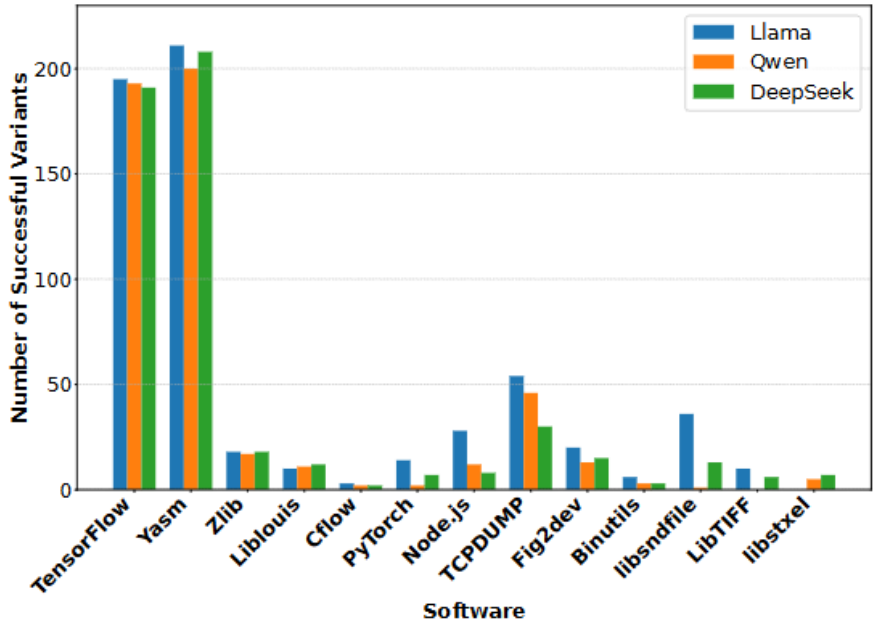
- RQ3：PoCE能否生成更丰富的有效PoC变体
  - 整体表现：PoCE在13个项目中的12个生成最多的不重复有效变体
  - 相比LLM方法：Tensorflow生成196个，相比AID的107个，相对提升约83%
  - 与其他变体生成方法相比：Yasm生成212个，高于Sleuth的194个；libsndfile生成28个，高于Redqueen的22个
  - 相比直接提示生成：libsndfile由4个增至28个，达到7倍
  - libsixel生成8个，略低于Weizz的9个，说明PoCE并非在所有项目上取得最优结果

Software	Naive	Sleuth	afl-cexp	Evocatio	Weizz	Redqueen	AID	PoCE
TensorFlow	109	N/A	N/A	N/A	N/A	N/A	107	**196**
Yasm	126	194	10	23	87	8	165	**212**
Zlib	6	N/A	N/A	N/A	N/A	N/A	8	**18**
Liblouis	6	6	4	8	4	0	6	**11**
Cflow	2	2	1	0	N/A	0	1	**4**
Pytorch	6	N/A	N/A	N/A	N/A	N/A	6	**14**
Node.js	22	N/A	N/A	N/A	N/A	N/A	25	**36**
TCPDUMP	43	N/A	N/A	N/A	0	N/A	43	**54**
Fig2dev	0	0	0	1	17	0	0	**20**
Binutils	1	5	3	4	0	0	1	**7**
libsndfile	4	2	5	2	0	22	2	**28**
LibTIFF	0	2	1	7	0	5	0	**11**
libsixel	2	0	0	4	**9**	5	2	8

实验结果 模型表现与计算开销

- **RQ4：不同大模型对生成效果的影响**
  - 三款模型均能在多数参评软件上生成有效PoC及变体，Llama在多数项目上表现较好
- **RQ7：PoC与变体生成耗时**
  - 对于初始PoC和变体生成，执行时间取决于数据数量以及处理该软件的复杂度





Software	#Data	PoC (hr)	#PoC	Variant (hr)
TensorFlow	113	7.49	94	6.9
Yasm	29	2.72	29	6.68
Zlib	11	0.96	10	1.12
Liblouis	20	8.26	10	4.1
Cflow	1	0.19	1	1.25
Pytorch	51	12.8	31	5.9
Node.js	35	6.79	32	7.7
TCPDUMP	27	6.34	27	7.83
Fig2dev	6	0.78	3	0.62
Binutils	10	0.85	4	1.83
libsndfile	16	1.3	8	1.2
LibTIFF	11	2.15	6	3.68
libsixel	3	0.39	1	0.46

实验结果 跨层与新漏洞探索

- RQ5：已知漏洞在最新版本上是否可触发
  - 零日PoC：针对已知漏洞构造并且在实验时的软件最新版本中仍能触发异常的样例，不等同于新发现的漏洞
- RQ6：跨层软件中的生成与探索能力
  - PoCE支持跨语言层的PoC生成，并在TensorFlow探索中发现了26个此前未公开漏洞
  - TensorFlow探索过程
    - 以1179个内核实现文件作为对象，将涉及输入或内存操作的语句作为候选目标
    - 生成上层触发PoC，并通过执行检查底层行为
    - 在26个文件上成功生成PoC

Software	Method	Llama	Gemini	Claude
TensorFlow	CyberGym	3	5	8
	PoCE	10	14	15
Yasm	CyberGym	6	9	7
	PoCE	29	29	29
Zlib	CyberGym	0	1	2
	PoCE	2	4	5
Liblouis	CyberGym	1	2	4
	PoCE	1	3	4
Cflow	CyberGym	0	1	0
	PoCE	0	1	0
Pytorch	CyberGym	0	0	0
	PoCE	5	6	6
Node.js	CyberGym	0	2	2
	PoCE	1	2	2
TCPDUMP	CyberGym	0	0	0
	PoCE	0	0	0
Fig2dev	CyberGym	0	0	0
	PoCE	0	0	0
Binutils	CyberGym	0	1	1
	PoCE	1	2	2
libsndfile	CyberGym	1	3	4
	PoCE	2	4	5
LibTIFF	CyberGym	0	0	0
	PoCE	0	0	0
libsixel	CyberGym	0	0	0
	PoCE	0	0	0

实验结果 消融实验

- RQ8：PoCE的关键组件对PoC生成的有效性
  - RAG与日志分析均有助于PoC生成，其中移除日志分析造成的成功率下降更明显
- 移除RAG，生成成功率下降
  - Node.js成功率由91%降至31%
  - TCPDUMP由100%降至40.7%
  - Yasm与Cflow结果保持不变
- 移除日志分析，下降更明显
  - TenserFlow成功率由83%降至21.2%
  - 8个软件成功率降为0

Software	PoCE	W/O RAG	W/O Log
TensorFlow	94(83%)	87(77%)	24(21.2%)
Yasm	29(100%)	29(100%)	15(51.7%)
Zlib	7(63%)	4(36.4%)	0(0%)
Liblouis	10(50%)	4(20%)	1(5%)
Cflow	1(100%)	1(100%)	0(0%)
Pytorch	31(60%)	19(37%)	6(12%)
Node.JS	32(91%)	11(31%)	2(5%)
TCPDUMP	27(100%)	11(40.7%)	0(0%)
Fig2dev	3(50%)	1(16.7%)	0(0%)
Binutils	4(40%)	3(30%)	0(0%)
libsndfile	8(50%)	3(18%)	0(0%)
LibTIFF	6(54%)	3(27%)	0(0%)
libsixel	1 (33%)	0(0%)	0(0%)

特色创新

- 构建初始生成与变体扩展的一体化框架
  - 结合检索知识与执行反馈，迭代生成有效PoC
  - 将成功样例进一步用于变体扩展与验证，形成多样化测试集合
- 设计结构特征引导的变体生成机制
  - 源代码级结合AST与Use-Def链，指导API和参数变化
  - 字节级结合文件结构与崩溃信息，指导局部变异并探索不同触发方式
- 支持跨语言层与多种输入形式
  - 根据底层漏洞代码，构造通过上层API触发的PoC
  - 按输入形式适配后处理与执行环境，复用统一流程处理代码、图像和数据包等样例

算法总结

- 算法优势
  - 结合知识检索与执行反馈，提高初始PoC生成效果
  - 利用结构特征指导变异，生成更多不重复的有效触发样例
  - 支持跨语言层调用与多种输入形式，复用统一的生成验证流程
- 算法不足（论文分析）
  - 环境适配仍需投入：不同软件需要配置后处理逻辑与执行环境，尚非完全通用
  - 迭代生成存在开销：多会话调用与日志反馈带来时间和计算成本，长日志与多轮历史增加上下文管理负担
- 算法分析（个人思考）
  - 加强目标漏洞一致性核验，避免将其他原因造成的崩溃计为有效触发
  - 结合路径与触发行为筛选变体，减少重复测试
  - 探索Agent驱动的自适应生成流程



特点总结与未来展望

特点总结与未来展望

- 特点总结
  - PAGENT：程序分析引导的PoC生成
    - 静态分析提取调用路径与安全约束，减少初始线索对模型推断的依赖
    - 结合运行剖析与覆盖率反馈，指导Agent迭代修正触发输入
  - PoCE：跨层PoC生成与多样化变体扩展
    - 结合检索知识与执行反馈，适配跨语言层及不同输入形式
    - 利用代码或文件结构指导变异，通过执行验证获得不重复的有效样例
- 未来展望
  - 加强目标漏洞归因与证据核验，区分目标漏洞触发、其他异常与证据不足
  - 结合执行路径与触发行为保留代表性变体，减少后续重复测试
  - 减少规则、环境及格式适配工作，提升方法的工程可用性

参考文献

[1] DESAI A, SHAFIUZZAMAN M, GUO W, et al. PAGENT: Program Analysis Guided LLM Agent for Proof-of-Concept Generation[J]. Proceedings of the ACM on Software Engineering, 2026, 3(ISSTA): 2475-2497.

[2] TITHY T D, RODOSHI L H, AZAHAR A R, et al. PoCE: Automated Proof-of-Concept Synthesis using Large Language Models for Robust Validation[J]. Proceedings of the ACM on Software Engineering, 2026, 3(ISSTA): 986-1009.

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。

