Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心

大模型驱动的软件漏洞PoC自动生成方法

硕士研究生 王怡男

2026年08月30日

问题回溯

- 问题回溯
  - 算法原理图解释不足
  - 方法讲解偏流程化，缺少直击创新的讲述
  - 部分细节内容解释过多
- 相关内容
  - 2026.03.15 杨语航《基于智能体的自动化漏洞重现方法》

内容提要

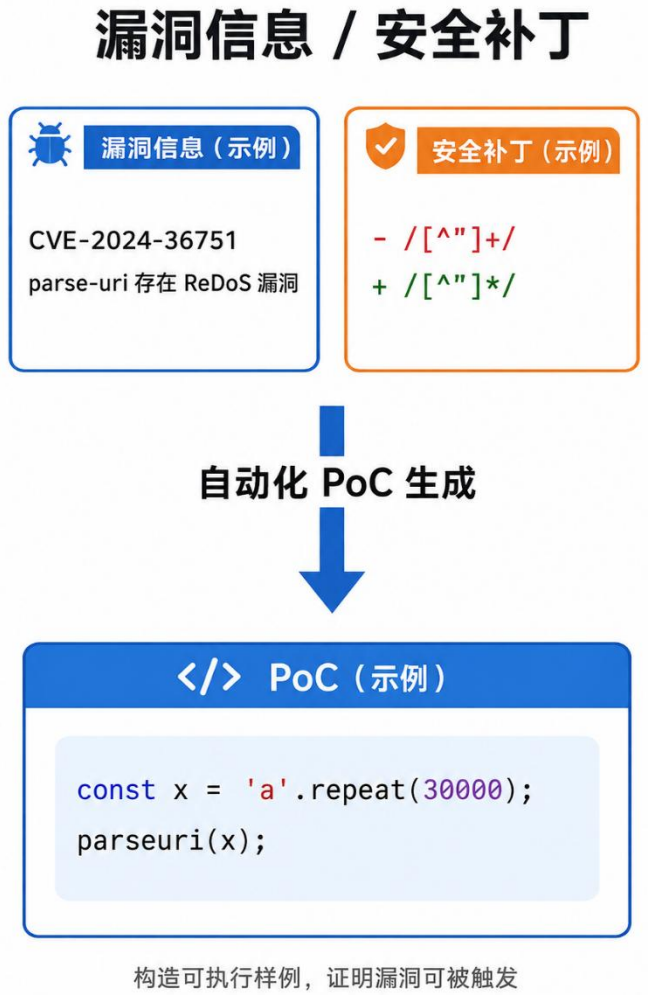
- 预期收获
- 题目内涵解析
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - PoCGen
  - PoCEvolve
- 特点总结与工作展望
- 参考文献

预期收获

- 预期收获
  - 1. 了解软件漏洞PoC自动生成任务的基本概念
  - 2. 理解LLM、静态分析与动态验证在PoC自动生成中的作用及互补关系
  - 3. 理解领域内对提示词优化的核心思路
  - 4. 了解现有LLM-based PoC自动生成方法的局限及未来发展方向

题目内涵解析 基本概念

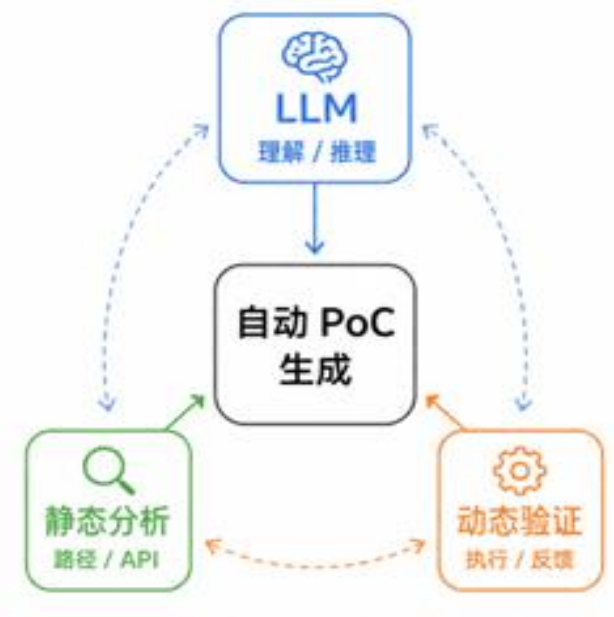
- 软件漏洞（Software Vulnerability）
  - 程序中可被攻击者利用的缺陷
    - 越权、信息泄露、代码执行等
  - 相比普通功能错误，更强调安全视角
- PoC（Proof-of-Concept）
  - 用于证明漏洞真实可触发的最小化样例
  - 输入文件、脚本、测试程序等
  - 默认漏洞已知
- PoC自动生成
  - 根据漏洞相关信息，自动构造触发漏洞的PoC
  - 常见输入：漏洞报告、代码库、漏洞补丁、运行反馈
  - 输出：一个可执行、可验证的漏洞触发样例



题目内涵解析 Why LLM?

- 传统PoC生成方法
  - 对形式化分析依赖强，分析成本高
    - 需要明确污点源（source）、敏感操作点（sink）、路径条件
  - 难以利用非结构化漏洞知识
    - 漏洞报告、补丁说明、API用法通常是自然语言
  - 失败反馈利用有限
    - 只知道“没触发”，但很难知道“为什么没触发”

- LLM-based方法
  - 能理解自然语言漏洞描述与补丁语义
  - 能推断vulnerable API、payload结构与调用方式
  - 能结合运行反馈进行迭代改进



研究背景与意义

- 研究背景
  - 软件漏洞数量持续增长，漏洞处理压力不断增大
    - 2026年上半年已公开约3.5万个CVE，同比增长约49.5%
  - 大量漏洞报告缺乏可执行的PoC
    - 漏洞报告通常仅提供非形式化、甚至不完整的漏洞描述
  - 人工构造PoC成本较高
    - 安全补丁与详细漏洞信息存在披露时间差
- 研究意义
  - 为漏洞提供可执行、可验证的证据
  - 辅助漏洞分析、修复与补丁验证
  - 支持漏洞回归测试与大规模自动化处置
    - 将生成的PoC纳入测试集，防止漏洞再次出现

研究历史与现状 大模型驱动的软件漏洞PoC自动生成方法

PoCGen：针对漏洞报告信息模糊、仅使用LLM会缺少精确程序上下文的问题，将大模型、静态污点分析与动态执行反馈结合：从漏洞报告定位Vulnerable API和Taint Path，生成候选PoC，并依据运行错误、覆盖率和调试信息持续优化。在SecBench.js的560个漏洞上成功生成395个有效PoC（71%）

PoCEvolve：进一步放宽PoCGen对详细漏洞报告的依赖，仅根据漏洞修复Commit与代码仓库重构漏洞类型、Vulnerable API和漏洞描述，并利用多次失败PoC中的漏洞上下文持续演化Prompt生成策略。在仅提供安全补丁的场景下，相比PoCGen，使用GPT-4o-mini和Qwen3.7-Plus时成功率分别相对提升20.7%和23.8%

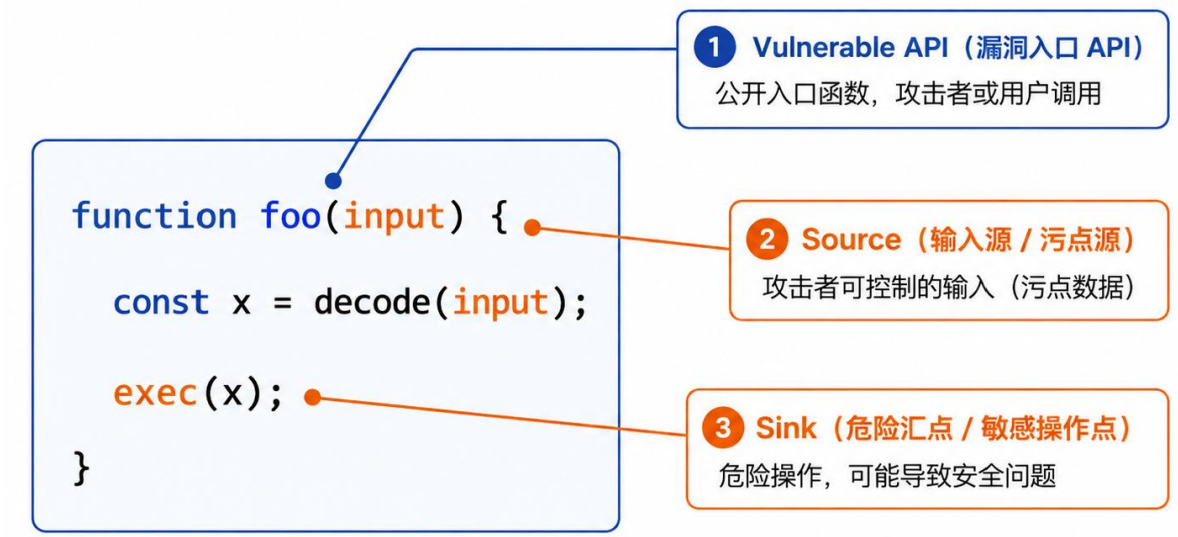
2025 2026 2026

FaultLine：针对通用LLM Agent难以理解复杂数据流和控制流的问题，引入分层程序推理：先追踪外部Source到漏洞Sink的数据流，再推理沿途分支条件，最后在反馈循环中生成PoV。方法不依赖特定语言的静态分析工具，可跨Java、C、C++使用；在100个真实漏洞上成功生成16个PoV

DrillAgent：针对现有LLM方法缺乏真实运行状态支撑的问题，将PoV生成建模为“假设—验证—优化”闭环，通过代码插桩获取细粒度执行信息，并将低层执行轨迹转化为LLM可理解的源码级约束，持续修正触发输入。在SEC-bench上成功验证55/190个PoV（28.9%），相比此前18%的最好结果，已解决CVE数量提升52.8%

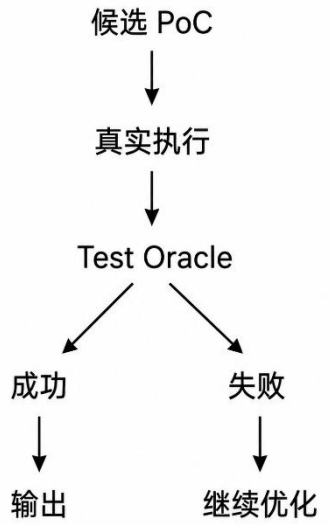
知识基础 漏洞触发路径与污点分析

- 漏洞触发
  - 攻击者输入沿程序路径到达敏感操作
    - Vulnerable API：攻击者或下游程序可调用的公开函数/方法
    - Source：攻击者可控数据进入程序的位置
    - Sink：可能产生安全影响的敏感操作
    - Taint Path（污点路径）：不可信数据由Source流向Sink的代码路径
- 污点分析
  - 数据是否来自不可信输入
  - 经过哪些变量/函数
  - 是否进入危险操作



知识基础 PoC的执行与验证

- 候选PoC ≠ 有效PoC
  - PoC必须能够实际执行
  - 必须真正通过漏洞路径触发预期安全行为
  - 需要设计Test Oracle（测试判定机制）
    - PoC执行并不能说明漏洞已触发
    - 一些漏洞类型和Test Oracle的例子



漏洞类型	解释	Test Oracle
Path Traversal（路径穿越）	恶意路径绕过目录限制，访问越权文件	是否访问越权文件
Command Injection（命令注入）	外部输入被拼接进系统命令	是否执行目标命令
Code Injection（代码注入）	恶意输入被程序解释并作为代码执行	是否实现任意代码执行
Prototype Pollution（原型污染）	修改对象原型，影响其他对象行为	是否污染目标属性
ReDoS（正则表达式拒绝服务攻击）	特殊输入导致正则表达式产生极高计算开销	是否造成异常执行延迟

PoCGen: Generating Proof-of-Concept Exploits for Vulnerabilities in Npm Packages

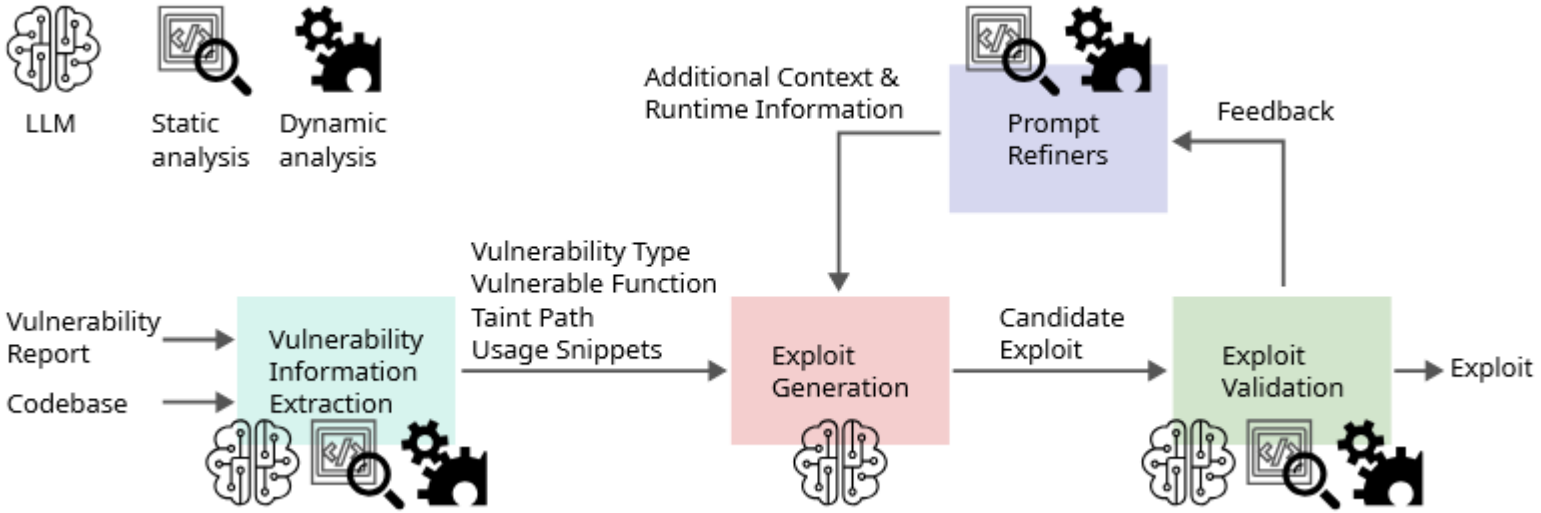
算法原理 TIPO

T	目标	自动生成并验证可执行PoC
I	输入	1个漏洞报告（自然语言描述）、1个对应漏洞版本的代码库
P	处理	1. 从报告与代码中提取漏洞类型、Vulnerable API、Taint Path、API用法 2. 结合上述信息，由LLM生成候选PoC 3. 真实执行候选PoC，并通过Test Oracle验证 4. 失败时加入静态/动态反馈迭代优化Prompt与PoC
O	输出	1个经过动态验证、能够真实触发目标漏洞的JavaScript PoC

P	问题	漏洞报告常模糊、不完整且缺少PoC； 传统程序分析难以利用自然语言漏洞知识，仅使用LLM又缺乏程序分析支撑
C	条件	需要漏洞报告与漏洞代码库；面向JavaScript/npm，支持5类典型漏洞
D	难点	从模糊报告中确定漏洞入口与触发路径；利用有限运行反馈不断修正PoC
L	水平	FSE 2026（CCF A）

算法原理 创新说明

- **PoCGen**
  - 首次将LLM、静态分析与动态分析系统结合，用于漏洞PoC的自动生成与验证
    - 从漏洞报告+漏洞代码库出发，构建闭环
      - 漏洞信息提取—PoC生成—动态验证—反馈优化
    - 根据漏洞报告与代码自动提取漏洞类型、污点路径等漏洞相关信息
    - 对候选PoC执行与漏洞类型相关的Test Oracle验证
      - 失败后利用静态信息和运行时信息不断优化Prompt，直到生成有效PoC或耗尽预算



算法原理 漏洞信息提取

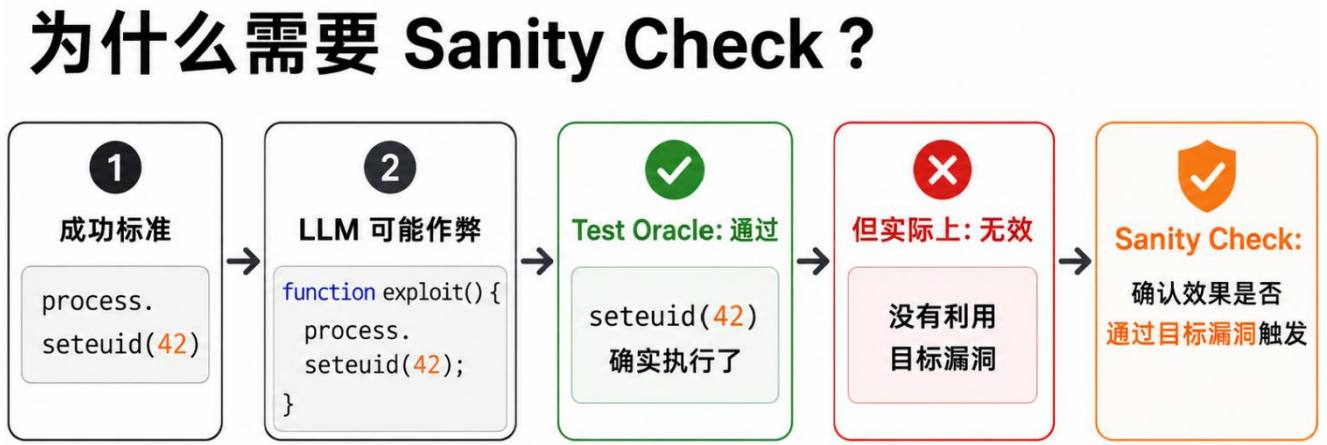
- 漏洞信息提取
  - 从漏洞报告+代码库中提取PoC生成所需的关键上下文
  - 综合利用LLM、静态分析与动态分析，依次获取4类信息
    - 漏洞类型（Vulnerability Type）
      - LLM从漏洞报告中提取
    - 漏洞入口函数（Vulnerable API）
      - 动态加载npm包
      - LLM排序
    - 漏洞传播路径（Taint Path）
      - 静态污点分析为主
      - 找不到则PoC生成失败
    - 函数使用示例（Usage Snippets）
      - 静态分析+LLM

````
Vulnerable method `import` of class `Environment` located in djv/lib/djv.js:
```js
import(config) { // tainted: "config"
  const item=JSON.parse(config) // tainted: "config"
  let restoreData=item // tainted: "item"
  if (item.name && item.fn && item.schema) {
    restoreData={
      [item.name]: item,
    }
  }
  Object.keys(restoreData).forEach((key)=>{ // tainted: "restoreData"
    const {name, schema, fn: source}=restoreData[key] // tainted: "restoreData"
    const fn=restore(source, schema, this.options) // tainted: "source"
    this.resolved[name]={
      name,
      schema,
```
Call to `restore`:
```js
function restore(source, schema, {inner}={}) { // tainted: "source"
  const tpl=new Function("schema", source)(schema) // tainted: "source"
  if (!inner) {
```
````

算法原理 PoC生成及验证

- PoC生成
  - 结构化上下文约束
    - 组织上一阶段提取到的信息
    - 加入PoC模板与相似漏洞PoC
    - 为不同漏洞指定明确触发
- PoC验证
  - 真实执行候选PoC，使用漏洞类型对应的Test Oracle判断安全效果
  - Sanity Check
    - LLM可能“作弊”式生成

```
`{{vulnerableFunction}}` is vulnerable to {{vulnerabilityType}}.
## Vulnerability Description:
```
{{vulnerabilityDescription}}
```
{{> usageSnippets }}
{{> exploitSkeleton }}
## Task:
Create an exploit that uses the {{vulnerabilityType}} in `{{vulnerableFunction}}` to
{{vulnerabilityTrigger}}.
Respond with the full exploit code and explain why it works.
If there is an exception thrown, do not try to handle it and pass it on.
Enclose the exploit code in backticks and define the exploit within a function named `exploit`.
{{> similarExploits }}
## Source code:
{{taintPathSnippets}}
```



算法原理 提示词改进

- 提示词改进
  - 目标：候选PoC验证失败后，根据失败原因补充针对性信息，重新生成PoC
  - 上下文改进：补充缺失的代码上下文
    - 函数体补充器：补充Taint Path所涉及函数的完整函数体
    - 缺失声明补充器：按LLM请求补充相关变量、函数的定义
  - 运行时改进：利用候选PoC的真实执行结果
    - Error、Coverage、关键表达式的实际运行值、Sink实际接收到的参数值
  - 改进循环：选择更有价值的改进方向持续探索
    - 使用优先队列管理不同Prompt，根据Taint Path覆盖进展+新错误信息进行评分
    - 删除已被模型正确掌握、无需重复提供的上下文，控制Prompt长度
    - 直到生成有效PoC或达到预算上限

算法原理 提示词改进

```
1  prompts ← PriorityQueue({0, getSeedPrompt()})
2  usedRefiners ← {}
3  seenExploits ← {}
4  while prompts ≠ {} and |usedRefiners| < 30 do
5      prompt ← top(prompts)
6      prompts ← prompts \ {prompt}
7      prompts ← prompts ∪ {prompts with more info}
8      exploit ← generateExploit(prompt)      // via LLM
9      if exploit ∉ seenExploits then
10         result ← runAndValidate(exploit)
11         if result.success then
12             return exploit
14         correctlyUsedInfo ← getCorrectlyUsedInfo(prompt, result) // Information in the prompt
           ↪ that the LLM used correctly
15         prompts ← prompts ∪ {(
16           result.taintSteps + result.newErrors,
17           prompt + result.errors - correctlyUsedInfo
18         )}
19         usedRefiners ← usedRefiners ∪ {used refiners}
20         seenExploits ← seenExploits ∪ {exploit}
21     end if
22 endwhile
23 return failure
```

初始化：建立提示词队列、
已用改进器集合与已见PoC集合

第一阶段：从优先队列中取出
当前提示词，并补充更多信息

第二阶段：调用LLM生成
候选PoC，并去除重复候选

第三阶段：执行并验证
候选PoC，若成功则直接返回

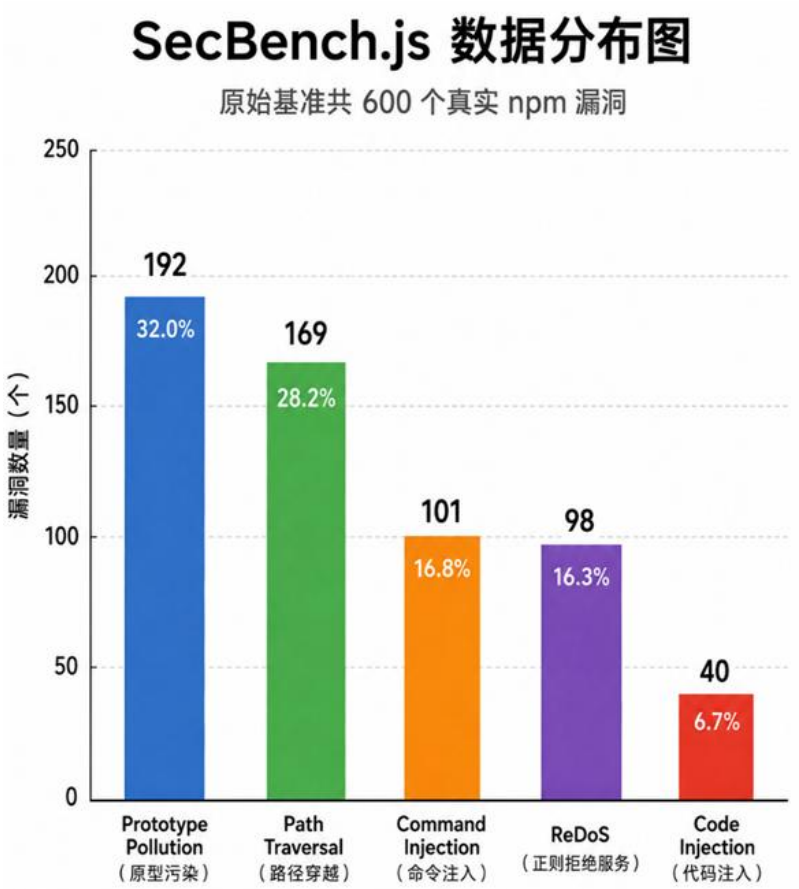
第四阶段：提取正确利用的信息，
依据覆盖进展与新增错误构造
新提示词并评分

第五阶段：更新已用改进器
与已见PoC集合

终止条件：队列耗尽或
达到预算上限后返回失败

实验设计 数据资源

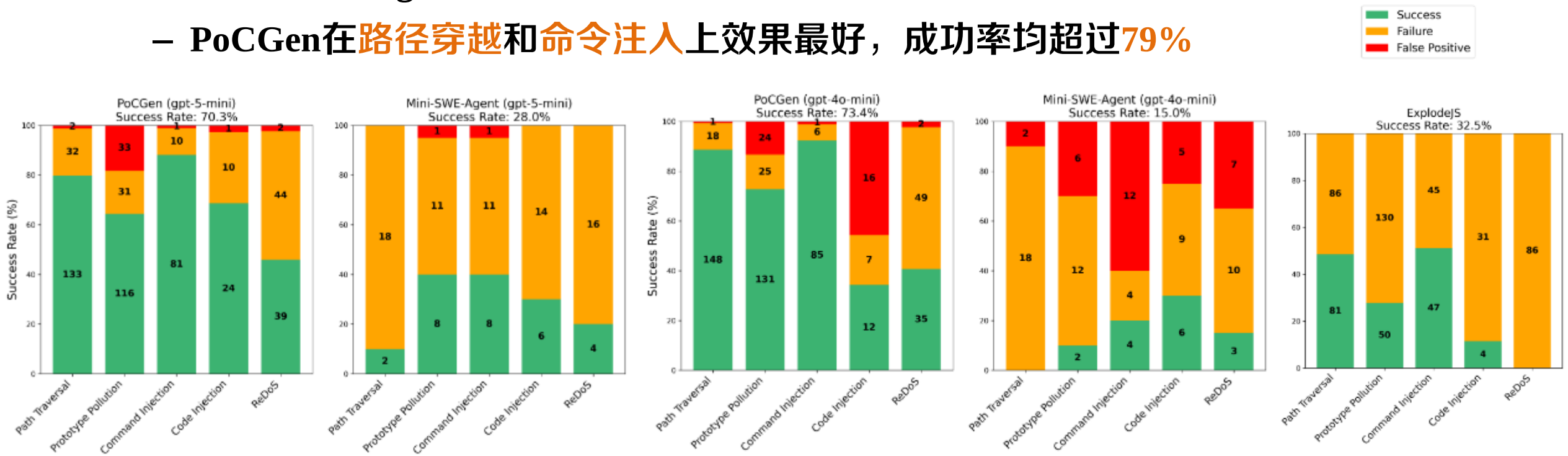
- 数据资源
  - SecBench.js：包含600个真实npm软件包漏洞
    - 覆盖共5类漏洞，最终使用560个漏洞
  - GitHub Security Advisories（GSA）近期漏洞集
    - 筛选属于前述5类的漏洞，得到126个近期真实漏洞
- 评估标准
  - *Success Rate*：成功生成有效PoC的漏洞比例
  - *Failed Attempts*：未成功生成有效PoC
  - *False Positives*：通过Test Oracle，但实际未触发漏洞
  - 重复运行3次，取平均结果
- 对比方法
  - Explode.js：传统程序分析方法
  - Mini-SWE-Agent：通用LLM Agent方法



漏洞数量（个）	192	169	101	98	40
占比（%）	32.0%	28.2%	16.8%	16.3%	6.7%

实验设计 实验结果

- 对比实验
  - PoCGen在 SecBench.js的560个漏洞上，平均成功生成395个有效PoC，成功率70.3%；Explode.js为182/560，32.5%，PoCGen提升37.8%
  - Mini-SWE-Agent在随机抽取的100个漏洞上平均成功率28%
  - PoCGen在路径穿越和命令注入上效果最好，成功率均超过79%



实验设计 实验结果

- 消融实验
  - Taint Path（漏洞传播路径）是贡献最大的部分，消融掉之后成功率下降50%
  - Usage Snippets（函数使用示例）贡献度第二，消融掉之后成功率下降36%
  - 只有27%的成功PoC是首轮直接生成的
    - 47%需要2-10次提示词改进
    - 成功PoC平均需要4轮提示词改进
    - 提示词改进是取得高成功率的重要原因

Configuration	Valid Exploits	Success Rate
PoCGen	**395**	**71%**
noTaintPath	115	21%
noUsageSnippet	196	35%
noFewShot	375	67%
noDebugger	377	67%
noErrorRefiner	376	67%

实验设计 实验结果

- 实用性实验
  - 每次PoC生成平均耗时11min，成功案例平均7min
  - 平均一次生成成本约$0.025，成功案例平均仅约$0.009
- 泛化性实验
  - PoCGen在GSA上成功生成76/126个有效PoC，成功率60.3%
  - 其中6个漏洞原报告无PoC
    - 作者补充提交后，5个已被接受

```
const tarfs = require("tar-fs");
const fs = require("fs");
const path = require("path");

async function exploit() {
    // Create a writable stream to extract the tar content
    const extractStream = tarfs.extract('/', {
        // We can ignore the file type checks to allow the extraction of the malicious file
        ignore: (name) => false,
    });

    // Create a tar stream
    const tarStream = tarfs.pack().on('error', (err) => {
        throw err;
    });

    // Append the malicious entry to the tar stream
    tarStream.entry({ name: '/flag.txt', mode: 0o644 }, Buffer.from('This is a flag!'));

    // Finalize the tar stream
    tarStream.finalize();

    // Pipe the tar stream into the extract stream
    tarStream.pipe(extractStream);
}

// Call the exploit function
exploit();
```

算法总结

- 算法贡献
  - 融合LLM、静态分析与动态分析，构建自动化PoC生成与验证框架
    - LLM理解漏洞语义与生成代码，程序分析提供确定性的代码路径和执行信息
  - 提出“生成—执行—验证—提示词改进”闭环
- 算法局限
  - 依赖已公开的漏洞报告
    - 必须获得漏洞自然语言描述及漏洞代码库后才能开始分析
    - 对于补丁已公开、详细漏洞信息尚未披露的场景无法直接应用
  - 适用范围有限
    - 当前实现仅面向JavaScript/npm
    - 主要支持5类漏洞

算法原理 PoCEvolve

POCEVOLVE: Generating Proof-of-Concept Exploits from Security Patches with Vulnerability-Aware Prompt Evolution

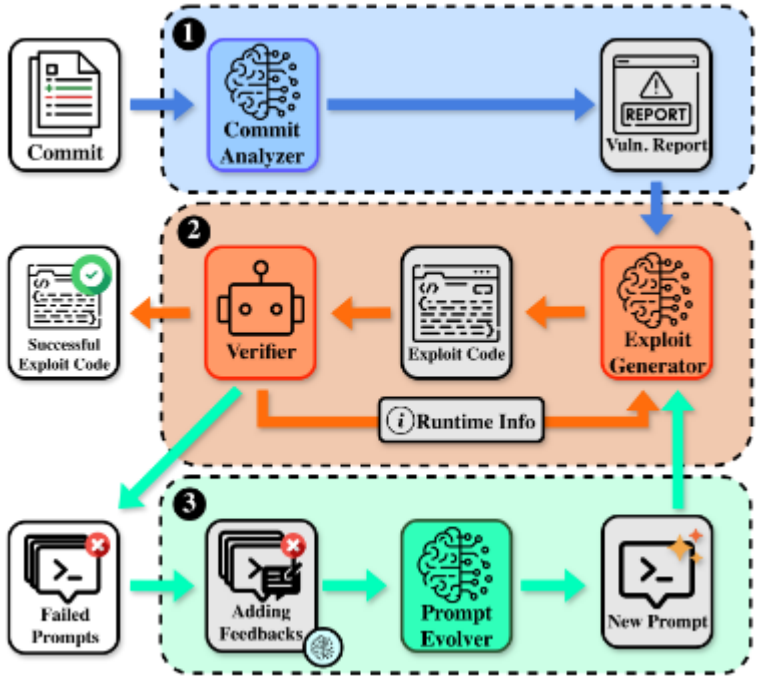
算法原理 TIPO

T	目标	自动生成并验证可执行PoC
I	输入	1个漏洞修复提交（含提交信息与代码变更）、1个对应漏洞版本的代码库
P	处理	1. 从补丁反推漏洞类型、Vulnerable API与漏洞描述 2. 生成候选PoC并真实执行验证 3. 分析多次失败案例，从多维漏洞上下文评分、反馈并演化Prompt，重新生成
O	输出	1个经过动态验证、能够真实触发目标漏洞的JavaScript PoC

P	问题	安全补丁常早于详细漏洞信息公开； 现有PoC生成方法依赖漏洞报告，难以处理Patch-only场景
C	条件	需要漏洞修复Commit+可访问代码仓库；面向JavaScript/npm
D	难点	从代码变更中反推漏洞语义与入口； 补丁不直接描述漏洞触发方式； 失败PoC中多种上下文信息难以有效利用
L	水平	arXiv，2026

算法原理 创新说明

- PoCEvolve
  - 首次面向Patch-only场景
    - 在无漏洞报告条件下，从漏洞修复提交直接生成可执行PoC
  - 提交分析：从提交信息、代码变更与仓库信息中重构漏洞类型、Vulnerable API与漏洞描述
  - 模块化PoC生成器及验证器
    - 复用PoCGen已有实现
  - 提示词演化：综合多次失败尝试，从多维漏洞上下文中评估有效信息，持续演化生成Prompt

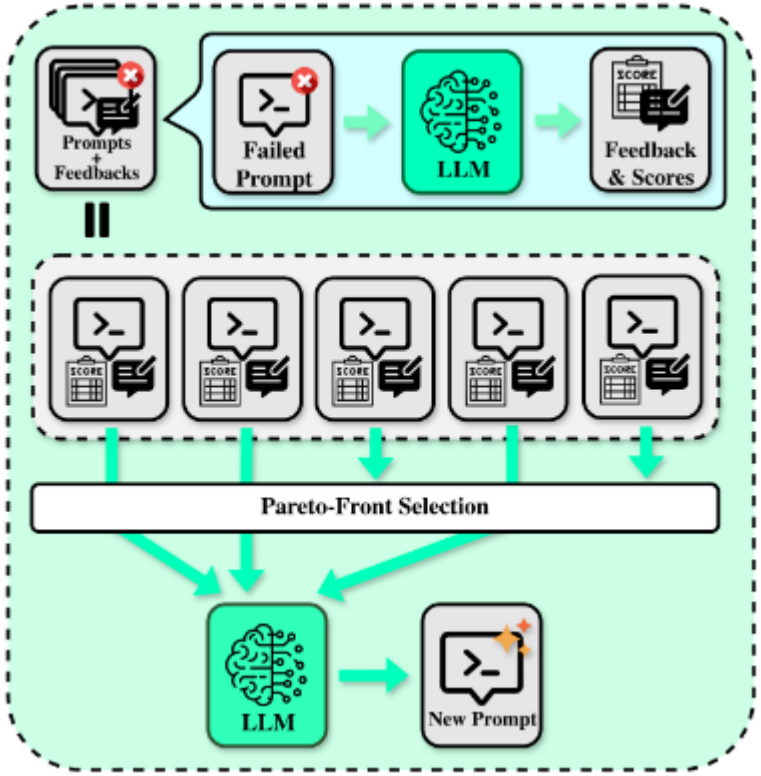


算法原理 漏洞修复提交分析

- 漏洞修复提交分析
  - 目标：从漏洞修复提交中恢复PoC生成所需的漏洞信息，替代真实漏洞报告
  - LLM输入
    - Package信息、漏洞版本、Commit Message、代码变更
  - 分析补丁中的修改位置与代码语义，重构3类核心信息
    - 漏洞类型、Vulnerable API、自然语言漏洞描述
  - 将结果组织为结构化JSON漏洞报告
- PoC生成及验证
  - 复用PoCGen的已有模块

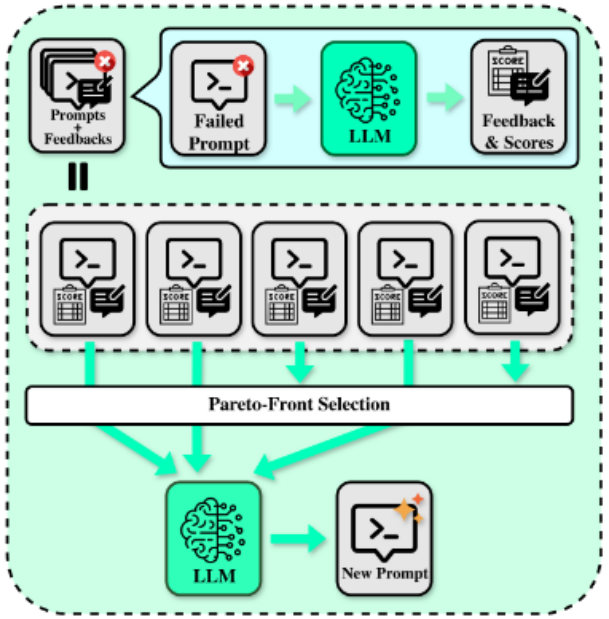
算法原理 提示词演化——失败候选评估

- 失败候选收集
  - 将多轮生成但未通过Test Oracle的Prompt、PoC、 LLM Response保留下来
  - 从失败候选中选取5个语义多样的Prompt，覆盖不同失败模式并减少重复分析
- 漏洞上下文多维评估
  - 目标：评价该信息对解释当前失败、指导下一轮Prompt修改是否有价值
  - LLM从8个维度评估每个失败候选
    - Vulnerable API、漏洞描述、函数使用示例、PoC模板、相似漏洞PoC、代码覆盖信息、漏洞传播路径、调试器输出
  - 每个维度赋予[0, 1]的有用性评分+简短理由



算法原理 提示词演化——Pareto选择与演化

- 候选状态构建
  - 结合补丁、执行结果和上下文评分，由LLM分析失败原因及生成Prompt修改建议
    - 修改建议：对Prompt内容执行增加、删除或替换等修改
  - 将Prompt+PoC+验证结果+评分+LLM分析结果及修改建议，构成候选状态
- Pareto选择
  - Pareto前沿选择：保留在8个维度上未被其他候选全面支配的候选状态
    - 目标：兼顾不同生成策略
- 提示词演化
  - 综合当前Prompt与Pareto前沿上的多组经验，归纳共同失败模式并合成新的Prompt
  - 新Prompt再次进入PoC生成、验证、评分、选择、演化的闭环，实验中最多演化5轮



算法原理 Pareto选择示例说明

Algorithm 2 Pareto Candidate Selection

Require: Candidate states $\mathcal{A} = \{(p, e, r, s, f)\}$, vulnerability-context criteria $\mathcal{K}$
Ensure: Pareto-front candidate set Q

```
if A = ∅ then
    return ∅
end if
Q ← ∅
for all a_i ∈ A do
    dominated ← FALSE
    for all a_j ∈ A do
        if a_i ≠ a_j and ∀k ∈ K : a_j.s[d] ≥ a_i.s[d] and ∃k ∈ K :
a_j.s[d] > a_i.s[d] then
            dominated ← TRUE
            break
        end if
    end for
    if ¬dominated then
        Q ← Q ∪ {a_i}
    end if
end for
return Q
```

Pareto Front 示例（仅看两个维度）

Prompt	Vulnerable API 信息质量	Debugger Output 信息质量
A	0.9	0.3
B	0.7	0.8
C	0.6	0.2

先看 C：A 全面优于 C

A = (0.9, 0.3)
C = (0.6, 0.2)

A 在两个维度上都不比 C 差，
而且两个维度都更高：
→ A 全面优于 C

因此：C 被 A 支配（dominated）
C 没有保留价值

再看 A 与 B：无法简单比较

A：API 很好，Debugger 较差
B：API 稍差，Debugger 很好

A 代表：“漏洞入口理解得非常好”
B 代表：“运行状态理解得非常好”

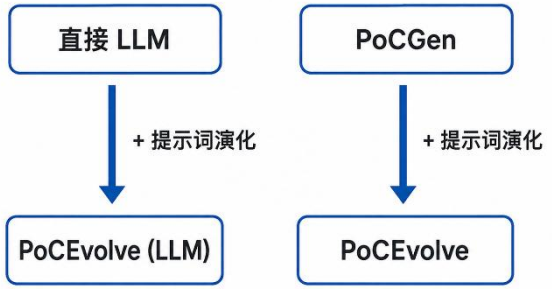
因此 A、B 都是 non-dominated candidates
共同组成 Pareto Front（帕累托前沿）

规则：若存在另一个候选 aj，使它在所有维度上都 ≥ 当前候选 ai，且至少一个维度严格更高，则 ai 被支配并淘汰。最后只保留没有被任何其他候选支配的候选。

实验设计 数据资源

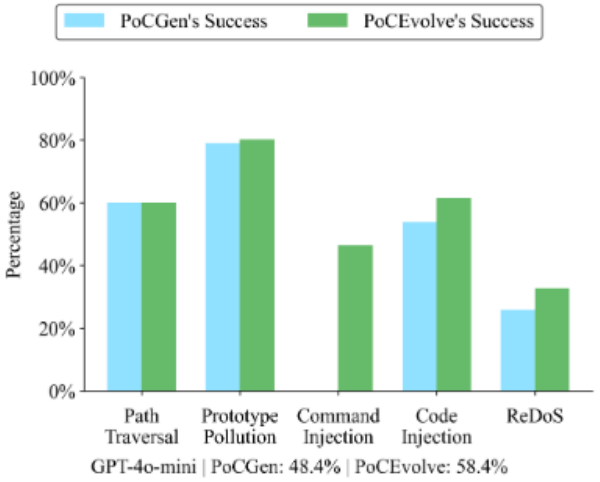
- 数据资源
  - SecBench.js：论文采用的子集共559个JavaScript/npm漏洞
    - 覆盖共5类漏洞
  - SecBench.VFC.js
    - 筛选具有漏洞修复提交的漏洞，最终获得190个
- 评估标准
  - *Success Rate*：成功生成有效PoC的漏洞比例
  - *Cost*：平均LLM调用成本
  - *Runtime*：平均漏洞处理时间
- 对比方法
  - LLM：直接根据修复提交+仓库上下文生成PoC
  - PoCEvolve(LLM)：直接LLM生成器+提示词演化模块
  - PoCGen*：程序分析辅助的PoC生成器，输入所需的漏洞报告由补丁解析后生成

Vulnerability Type	SecBench.js			SecBench.VFC.js		
	GHSA	Snyk	Total	GHSA	Snyk	Total
Path Traversal	85	82	167	8	2	10
Prototype Pollution	136	44	180	61	20	81
Command Injection	66	24	90	24	4	28
Code Injection	26	10	36	9	4	13
ReDoS	59	27	86	40	18	58
Total	372	187	559	142	48	190

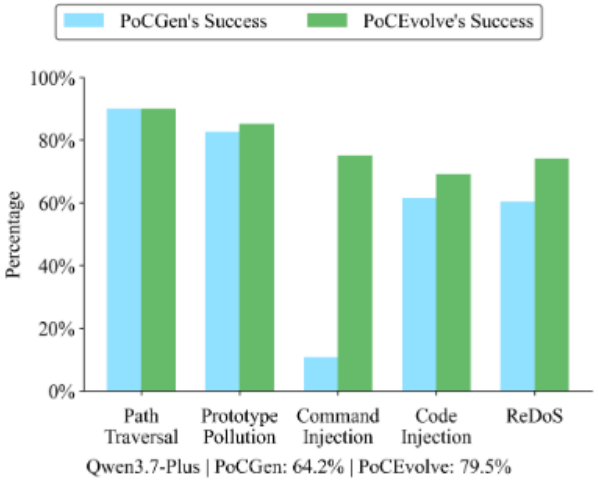


- RQ1：能否直接从漏洞修复提交生成PoC
  - GPT-4o-mini
    - LLM：19.5%→PoCEvolve (LLM)：28.4%
    - PoCGen：48.4%→PoCEvolve：58.4%
  - Qwen3.7-Plus
    - LLM：77.9%→PoCEvolve (LLM)：85.3%
    - PoCGen：64.2%→PoCEvolve：79.5%
  - 提示词演化能够稳定提升不同生成器与不同LLM的效果
  - 当底层模型足够强时，复杂的程序分析辅助框架未必一定优于直接给模型丰富上下文

Underlying LLM	Method	Success	Improvement
GPT-4o-mini	LLM	19.5%	–
	PoCEvolve (LLM)	28.4%	+45.9% over LLM
	PoCGen	48.4%	–
	PoCEvolve	**58.4%**	**+20.7% over PoCGen +200.0% over LLM**
Qwen3.7-Plus	LLM	77.9%	–
	PoCEvolve (LLM)	**85.3%**	**+9.5% over LLM**
	PoCGen	64.2%	–
	PoCEvolve	79.5%	+23.8% over PoCGen

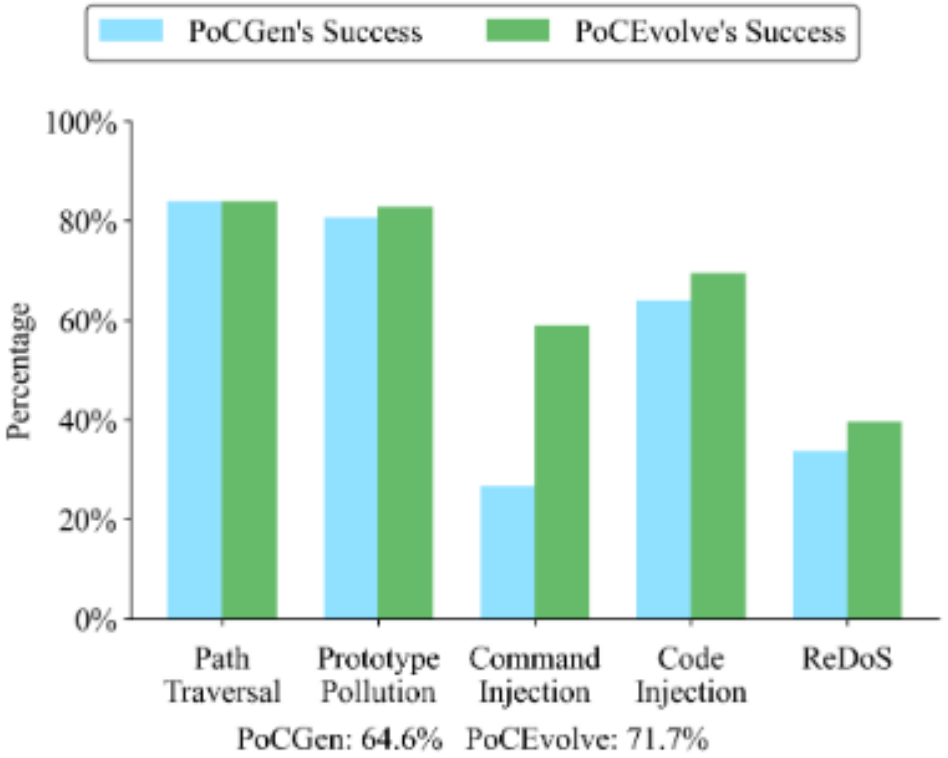


GPT-4o-mini | PoCGen: 48.4% | PoCEvolve: 58.4%



Qwen3.7-Plus | PoCGen: 64.2% | PoCEvolve: 79.5%

- RQ2：已有漏洞报告时，提示词演化是否仍有作用
  - 实验设置：不再进行漏洞修复提交分析，直接使用真实漏洞报告
  - PoCEvolve的成功率为71.7%，相对PoCGen提升11.1%
  - 即使漏洞信息已经很完整，PoCEvolve的提示词演化仍然能够从失败尝试中挖出PoCGen的提示词改进没有充分利用的信息



- RQ3：效果提升的代价是什么？
  - 成本花费：GPT-4o-mini下，PoCEvolve的平均花费约为PoCGen的2.45倍；Qwen3.7-Plus下约为3.51倍
  - 时间开销：GPT-4o-mini下，PoCEvolve的平均开销约为PoCGen的1.74倍；Qwen3.7-Plus下约为2.42倍
  - 提示词演化本身占据了大比例的成本与运行时间
    - 大量LLM调用都花在评分、生成反馈、Pareto筛选后的反思与Prompt合成上

底层模型	方法	成功率	平均成本 / 漏洞	平均时间 / 漏洞
GPT-4o-mini	PoCGen	48.4%	$0.0366	15m 08s
GPT-4o-mini	PoCEvolve	58.4%	$0.0895	26m 19s
Qwen3.7-Plus	PoCGen	64.2%	$0.0710	21m 49s
Qwen3.7-Plus	PoCEvolve	79.5%	$0.2492	52m 54s

算法总结

- 算法贡献
  - 提出Patch-only PoC Generation新场景
    - 在无详细漏洞报告条件下，仅依赖漏洞修复提交与代码仓库上下文生成PoC
  - 设计漏洞修复提交分析模块，从安全补丁自动重构PoC生成所需的漏洞知识
  - 提出漏洞感知的提示词演化机制
    - 从失败PoC中对多维漏洞上下文进行评分与反馈
    - 结合Pareto前沿选择+LLM反思合成，自动演化生成Prompt
- 算法局限
  - 提示词上下文持续膨胀，受LLM上下文窗口限制
  - 成本和运行时间显著增加
  - 当前验证范围仍局限于JavaScript/npm

特点总结与未来展望

特点总结与未来展望

- 特点总结
  - PoCGen && PocEvolve
    - LLM-based
      - 利用大模型进行漏洞语义理解与PoC代码生成
    - 程序分析+动态验证
      - 通过污点分析、调试信息、代码覆盖等提供确定性程序上下文
      - 使用Test Oracle判断生成结果是否真实触发目标漏洞
    - 反馈驱动的迭代生成
      - 根据执行反馈或多维评分、失败经验进行提示词迭代
- 未来展望
  - 扩展语言、软件生态与漏洞类型
  - 降低提示词演化的上下文与计算开销
  - 构建更通用、可靠的自动验证机制

参考文献

[1] Simsek D, Eghbali A, Pradel M. Pocgen: Generating proof-of-concept exploits for vulnerabilities in npm packages[J]. Proceedings of the ACM on Software Engineering, 2026, 3(FSE): 3887-3908.

[2] Tran D M, Widyasari R, Irsan I C, et al. PoCEvolve: Generating Proof-of-Concept Exploits from Security Patches with Vulnerability-Aware Prompt Evolution[J]. arXiv preprint arXiv:2607.22076, 2026.

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！