

《软件安全》实验报告

姓名：许洋 学号：2313721 班级：1070

实验名称：

Angr应用实例

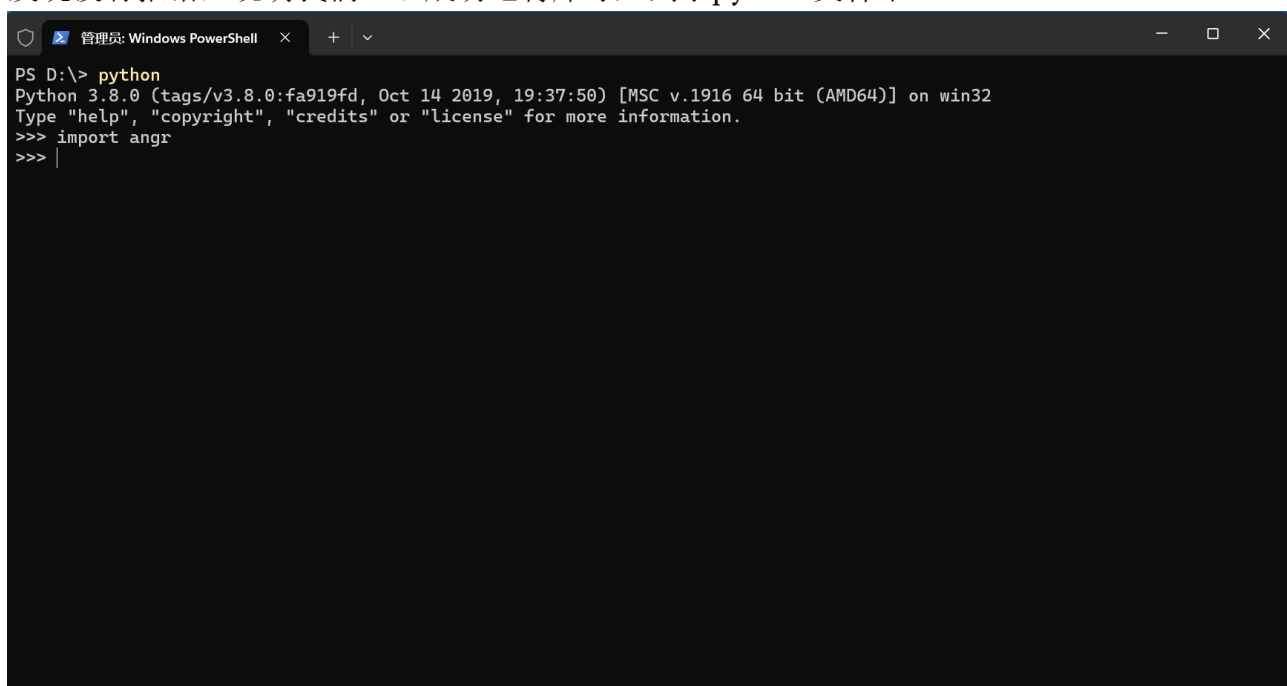
实验要求：

根据课本8.4.3章节，复现sym-write示例的两种angr求解方法，并就如何使用angr以及如何解决一些实际问题做一些探讨。

实验过程：

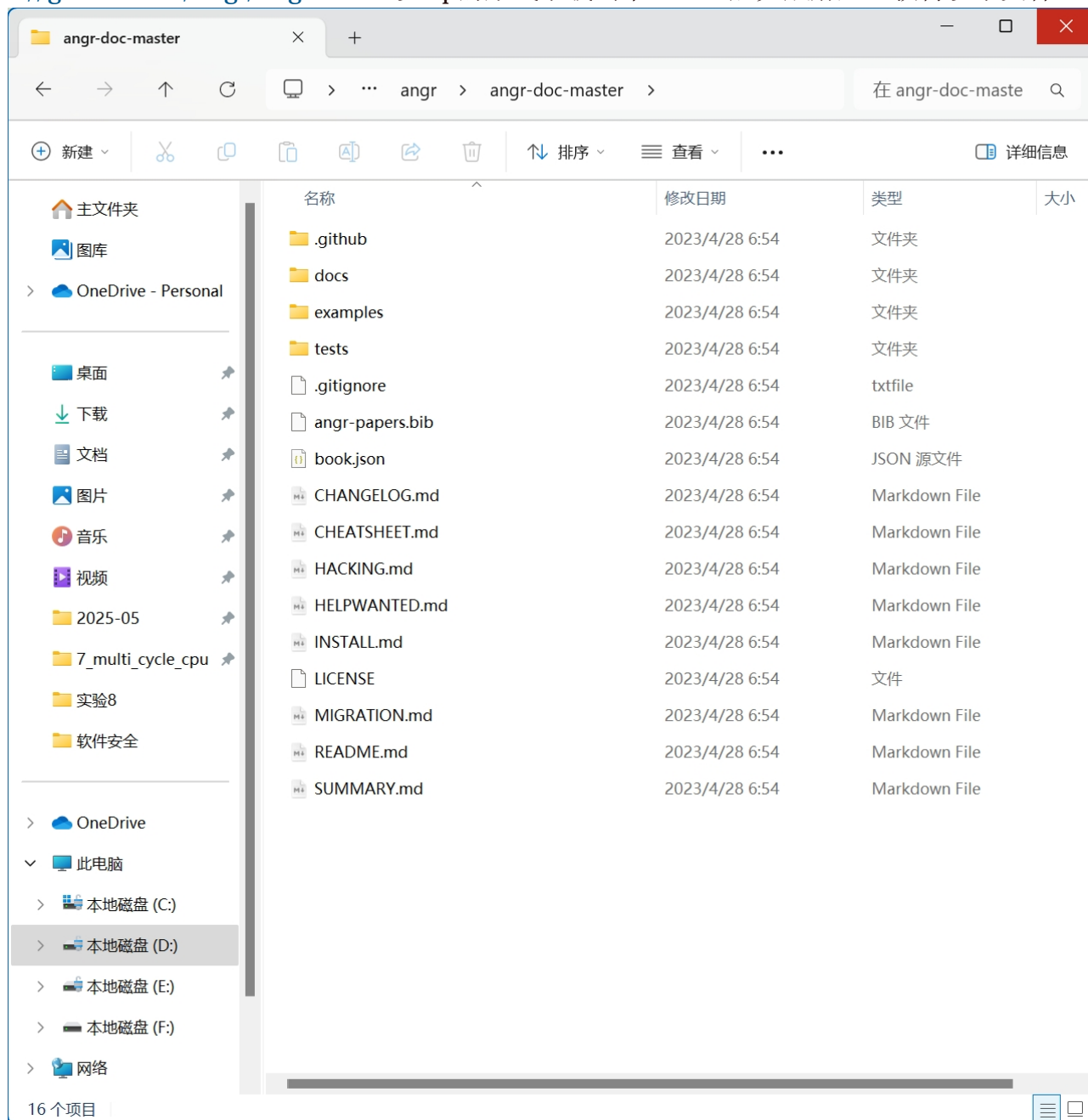
安装Python3和Angr

我的电脑中安装过python3.8，所以只需要在终端输入`pip install angr`，开始安装angr。为了确保我们成功的安装了angr，我们进行测试，启动python，输入`import angr`，发现没有报错，说明我们已经成功地将库导入到了python文件中。



```
PS D:\> python
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
>>> |
```

然后，我们下载angr的官方文档来获得实验所需要的样例。我们进入提供的网址<https://github.com/angr/angr-doc>，以zip的形式下载到本地，之后完成解压，获得以下文件：



复现sym-write的两种方法

一、问题描述与源码分析

我们面对的问题是：找到一个输入值 **u**，使得程序输出 "you win!"。该程序的源代码如下（位于 angr 文档示例中）：

issue.c源码：

```
#include <stdio.h>
char u=0;
```

```

int main(void)
{
    int i, bits[2]={0,0};
    for (i=0; i<8; i++) {
        bits[(u&(1<<i))!=0]++;
    }
    if (bits[0]==bits[1]) {
        printf("you win!");
    }
    else {
        printf("you lose!");
    }
    return 0;
}

```

程序统计变量 `u` 的 8 位中，每一位是 0 还是 1。若 0 和 1 的数量相等（即各有 4 个），则输出 "you win!"。因此，我们需要找出所有满足条件的 `u` 值（0~255 范围内的整数），其中二进制表示中有恰好 4 个 1 和 4 个 0。

二、使用 Angr 解题思路

为了自动化地求解这个问题，我们可以借助符号执行工具 `angr` 来进行逆向分析。

求解方法1

1. 构建项目并加载目标文件

首先，将上述 C 源码编译为可执行文件（如 `issue`），然后在 Python 中创建 `angr` 工程：

```

import angr
p = angr.Project('./issue', load_options={"auto_load_libs": False})

```

`auto_load_libs=False` 表示不自动加载系统库，避免不必要的干扰。

2. 初始化状态并设置符号变量

接下来初始化程序状态，并将变量 `u` 设置为符号变量：

```
state = p.factory.entry_state(add_options={
    angr.options.SYMBOLIC_WRITE_ADDRESSES})
```

通过反汇编工具可以查得，变量 `u` 存在于 `.bss` 段地址 `0x804a021` 处。我们在此处写入一个 8 位的符号变量：

```
u = claripy.BVS("u", 8)
state.memory.store(0x804a021, u)
```

3. 创建模拟管理器并探索路径

创建模拟管理器后，利用 `explore()` 方法进行状态搜索：

```
sm = p.factory.simulation_manager(state)
def correct(state):
    try:
        return b'win' in state.posix.dumps(1)
    except:
        return False
def wrong(state):
    try:
        return b'lose' in state.posix.dumps(1)
    except:
        return False
sm.explore(find=correct, avoid=wrong)
```

4. 获取并输出结果

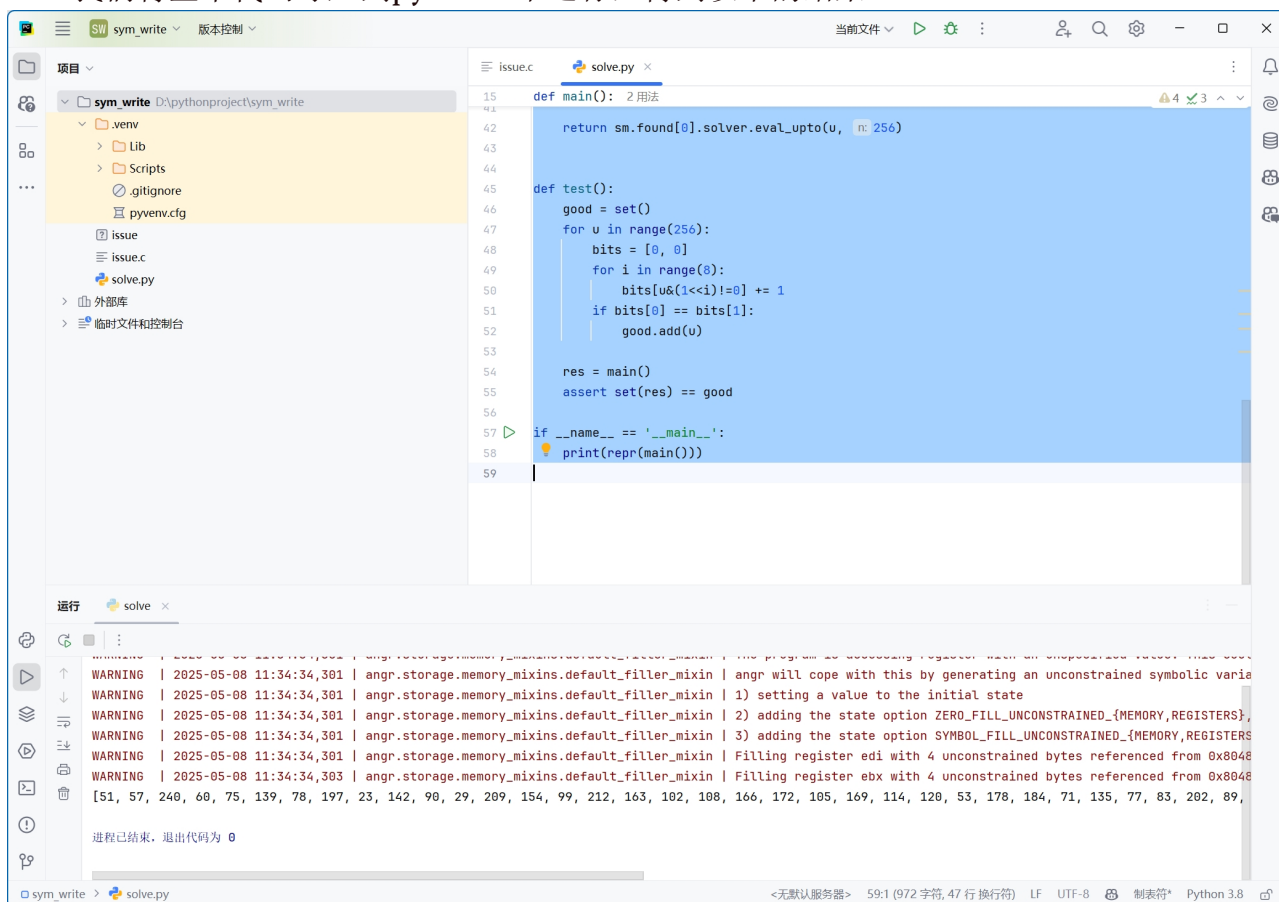
一旦找到符合条件的状态，就可以调用求解器获取 `u` 的可能取值：

```
result = sm.found[0].solver.eval_upto(u, 256)
print(repr(result))
```

其中：

- `eval_upto()` 用于获取最多 256 个可能解。
- `repr()` 将结果转换为字符串形式打印。

我们将整个代码导入到pycharm中运行，得到以下的结果：



可以发现我们得到了u的所有结果：

```
[51, 57, 240, 60, 75, 139, 78, 197, 23, 142, 90, 29, 209, 154, 99, 212, 163, 102, 108, 166, 172, 105, 169, 114, 120, 53, 178, 184, 71, 135, 77, 83, 202, 89, 147, 86, 153, 92, 150, 156, 106, 101, 141, 165, 43, 113, 232, 226, 177, 116, 46, 180, 45, 58, 198, 15, 201, 195, 85, 204, 30, 149, 210, 27, 216, 39, 225, 170, 228, 54]
```

以上的每一个解我们都可以带回到源程序中进行验证。

5. 验证结果

运行脚本后，得到一组满足条件的 `u` 值，例如 `u=51`（即二进制 `00110011`），确实有 4 个 1 和 4 个 0，符合题目要求。

求解方法2

第二种方法引入了 **hook** 技术，以提高符号执行效率，同时从特定地址开始执行。

1. Hook 特定指令

在地址 `0x08048485` 处是一条 `xor eax, eax` 指令，我们将其替换为自定义函数：

```
def hook_demo(state):  
    state.regs.eax = 0  
    p.hook(addr=0x08048485, hook=hook_demo, length=2)
```

这并不会改变程序逻辑，而是提供了一种更高效的模拟实现方式。

2. 从指定地址启动执行

使用 `blank_state()` 并指定起始地址：

```
state = p.factory.blank_state(addr=0x0804846B, add_options=  
    {"SYMBOLIC_WRITE_ADDRESSES"})
```

这样可以跳过无关的初始化代码，提升效率。

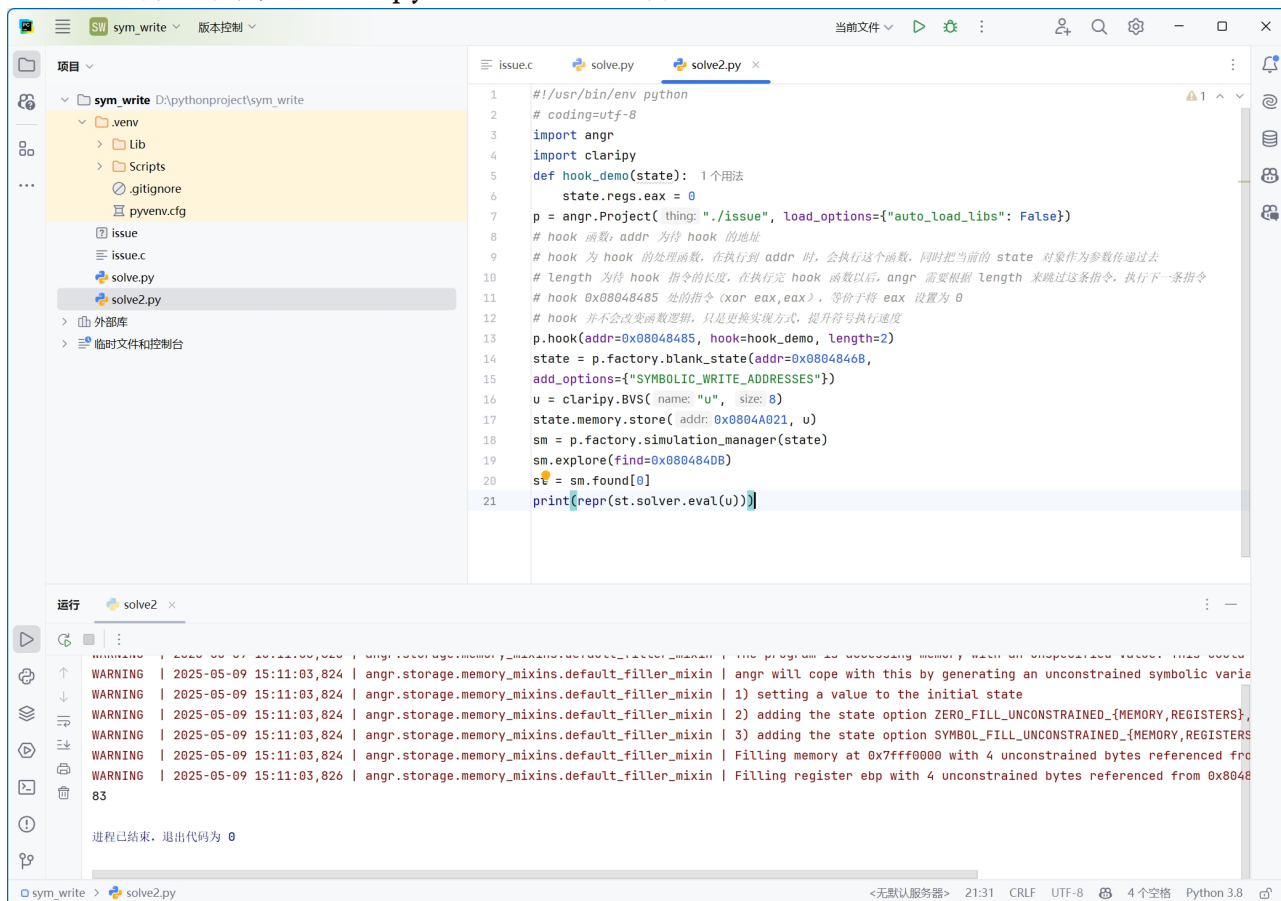
3. 执行探索并获取结果

只需寻找成功路径即可，无需同时避开失败路径：

```
sm.explore(find=0x080484DB)  
st = sm.found[0]  
print(repr(st.solver.eval(u)))
```

这里只返回一个符合条件的解。

我们将整个代码导入到pycharm中运行，得到以下的结果：



以上的一个解我们都可以带回到源程序中进行验证。

4. 最终效果与验证

运行改进后的脚本，得到一个具体的 `u` 值，例如 `u=83`，其二进制为 `01010011`，同样满足 4 个 1 和 4 个 0。

验证无误后，确认我们的 `angr` 分析正确解决了问题。

Anger 在实际问题中的应用

如何使用 `angr` 库

`Anger` 是一个基于 `Python` 的二进制分析框架，广泛用于符号执行、路径探索、逆向工程等场景。它可以帮助我们自动分析程序行为，寻找特定路径的输入，甚至发现潜在的安全漏洞。

要开始使用 `angr`，通常可以按照以下几个步骤进行：

1. 安装 `angr`

使用 `pip` 可以轻松安装 `angr`:

```
pip install angr
```

2. 导入库并加载目标程序

在 `Python` 脚本中导入 `angr`，并通过 `Project` 类加载你想要分析的二进制文件：

```
import angr
project = angr.Project("path/to/your/binary")
```

3. 设置初始状态与模拟管理器

创建程序入口点的状态对象，并用模拟管理器（`Simulation Manager`）来控制整个执行过程：

```
entry_state = project.factory.entry_state()
simgr = project.factory.simgr(entry_state)
```

4. 执行分析任务

你可以运行模拟器进行基本的执行，也可以使用 `explore()` 方法进行更智能的路径探索：

```
simgr.run() # 基础执行
simgr.explore(find=0x4005f6) # 寻找特定地址的目标路径
```

5. 处理结果并求解约束

如果找到了符合条件的路径状态，可以用求解器获取变量的具体值：

```
if simgr.found:
    solution_state = simgr.found[0]
    input = solution_state.posix.dumps(0)
    print("找到的输入是:", input.decode())
```

6. 分析输出信息

根据符号执行的结果，我们可以推导出程序的行为特征、漏洞触发条件，或者生成特定的输入数据。

Angr 在实际中的应用场景

Angr 不仅是一个理论工具，在实际安全研究和开发中也有非常广泛的应用，主要包括以下几个方面：

- **漏洞挖掘**

Angr 可以帮助我们自动化地分析二进制程序，识别如缓冲区溢出、格式化字符串、整数溢出等常见漏洞。通过探索不同的执行路径，可以快速定位存在风险的代码区域。

- **逆向工程辅助**

对于没有源码的程序，Angr 提供了强大的静态和动态分析能力，有助于理解程序逻辑、函数调用关系以及关键算法的实现方式。

- **加密算法逆向分析**

在面对自定义或混淆过的加密逻辑时，Angr 可以协助分析其输入输出之间的数学关系，帮助判断是否存在可被破解的弱点。

- **CTF 比赛实战**

在网络安全竞赛中，Angr 被大量用于解决逆向类题目。它可以快速找出程序所需的正确输入，从而绕过复杂的验证逻辑。

- **漏洞利用开发**

结合符号执行与路径约束，Angr 可以帮助研究人员构造精确的输入数据，以触发特定的漏洞路径，为后续的 exploit 开发提供支持。

- **模糊测试辅助**

Angr 可以与模糊测试工具结合使用，用于生成更有针对性的测试用例，提高测试覆盖率并发现更多潜在问题。

- **固件安全分析**

对嵌入式设备的固件进行深入分析时，Angr 可以帮助识别隐藏的后门、弱密码机制或其他安全隐患。

心得体会：

通过本次实验，我按照课本 8.4.3 节的内容，成功复现了 sym-write 示例中的两种 angr 求解方法。在实践过程中，我深入理解了 angr 在二进制分析中的应用方式，掌握了符号执行的基本流程，包括如何设置符号变量、构建模拟状态、探索程序路径以及进行约束求解。

第一种方法通过从程序入口开始执行，利用输出字符串判断程序是否达到目标路径，从而求解出满足条件的所有输入值。这种方法逻辑清晰，适合初学者理解和掌握 angr 的基本操作。第二种方法则采用了 hook 技术，并从指定地址开始执行，跳过了不必要的代码路径，提升了分析效率。这让我认识到，在面对复杂程序时，合理利用 angr 提供的各种功能可以显著优化分析性能。

通过本次实验，我深刻体会到 **angr** 在漏洞挖掘、逆向分析、CTF 解题等实际安全问题中的强大作用。它不仅可以帮助我们自动化地寻找特定路径的输入条件，还能辅助理解程序行为、发现潜在漏洞。未来我希望继续深入学习 **angr** 的高级功能，如结合模糊测试、动态插桩等技术，提高自己在软件安全领域的实战能力。