软件安全实验报告

姓名：郭子涵 学号：2312145 班级：信息安全、法学双学位班

1 实验名称

Angr应用示例

2 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** 以及怎么解决一些实际问题做一些探讨。

3 实验过程

3.1 安装Angr

在已经安装了python3的基础上，适用命令行安装angr :pip install angr.

```
  Running setup.py install for mulpyplexer ... done
Successfully installed GitPython-3.1.44 ailment-9.2.154 angr-9.2.154 archinfo-9.2.154 b
ackports-strenum-1.3.1 bitarray-3.4.0 bitstring-4.3.1 cachetools-5.5.2 capstone-5.0.3 c
art-1.2.3 claripy-9.2.154 cle-9.2.154 cxxheaderparser-1.5.0 gitdb-4.0.12 markdown-it-py
-3.0.0 mdurl-0.1.2 mulpyplexer-0.9 pefile-2024.8.26 protobuf-6.30.2 pycryptodome-3.22.0
 pydemumble-0.0.1 pydot-4.0.0 pyelftools-0.32 pyformlang-1.0.11 pyvex-9.2.154 rich-14.0
.0 smmap-5.0.2 sortedcontainers-2.4.0 unique-log-filter-0.1.0 z3-solver-4.13.0.0

[notice] A new release of pip is available: 23.0.1 -> 25.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

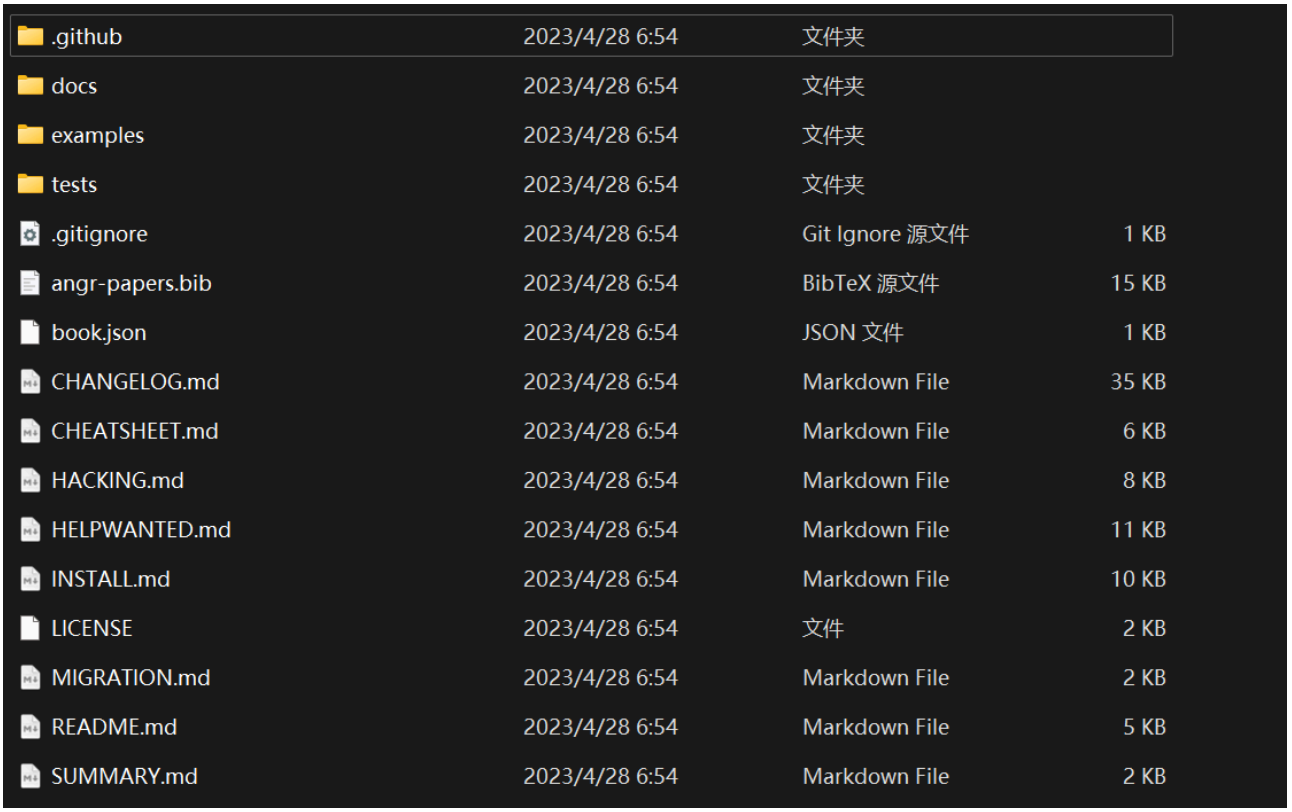
图1 安装angr

测试安装：输入命令python,进入python界面，然后输入import angr如下图2所示，证明安装成功。

```
D:\>python
Python 3.10.11 (tags/v3.10.11:7d4cc5a, Apr  5 2023, 00:38:17) [MSC v.1929 64 bit (AMD64
)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
>>>
```

图2 测试安装angr

将angr官方网站的https://github.com/angr/angr-doc里的所有文档以zip方式下载到本地。由图3所示，在docs文件夹中体况很多angr的用法，在examples文件夹中展示了Angr的用法，比如cmu_binary_bomb、simple_heap_overflow等二进制爆破、堆溢出等漏洞挖掘、软件分析的典型案例。



.github	2023/4/28 6:54	文件夹	
docs	2023/4/28 6:54	文件夹	
examples	2023/4/28 6:54	文件夹	
tests	2023/4/28 6:54	文件夹	
.gitignore	2023/4/28 6:54	Git Ignore 源文件	1 KB
angr-papers.bib	2023/4/28 6:54	BibTeX 源文件	15 KB
book.json	2023/4/28 6:54	JSON 文件	1 KB
CHANGELOG.md	2023/4/28 6:54	Markdown File	35 KB
CHEATSHEET.md	2023/4/28 6:54	Markdown File	6 KB
HACKING.md	2023/4/28 6:54	Markdown File	8 KB
HELPWANTED.md	2023/4/28 6:54	Markdown File	11 KB
INSTALL.md	2023/4/28 6:54	Markdown File	10 KB
LICENSE	2023/4/28 6:54	文件	2 KB
MIGRATION.md	2023/4/28 6:54	Markdown File	2 KB
README.md	2023/4/28 6:54	Markdown File	5 KB
SUMMARY.md	2023/4/28 6:54	Markdown File	2 KB

图3 angr-doc文件目录

4 angr示例--sym-write

4.1 issue.c源码

```c
#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;
}
```

4.2 第一种方法

solve.py源代码:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import angr
import claripy
def main():
    #1.新建一个工程，导入二进制文件，后面的选项是选择不自动加载依赖项，不自动加载依赖的库
    p = angr.Project('./issue', load_options={"auto_load_libs": False})
    #2.初始化一个模拟程序状态的SimState对象state,包含程序的内存、寄存器、文件系统数据符号信息等模拟运行时动态变化的数据。
    # blank_state():可通过给定参数addr的值指定程序起始运行地址
    # entry_state():指明程序在初始运行时的状态，默认从入口点执行
    # add_options获取一个独立的选项来添加到某个state中
    # SYMBOLIC_WRITE_ADDRESSES：允许通过具体化策略处理符号地址的写操作
    state = p.factory.entry_state(add_options={angr.options.SYMBOLIC_WRITE_ADDRESSES})
    # 3. 创建一个符号变量，这个符号变量以8位bitvector形式存在，名称为u
    u = claripy.BVS("u", 8)
    # 把符号变量保存到指定的地址中，这个地址是就是二进制文件中.bss段u的地址
    state.memory.store(0x804a021, u)
    # 4. 创建一个Simulation Manager对象，这个对象和我们的状态有关系
    sm = p.factory.simulation_manager(state)
    # 5. 使用explore函数进行状态搜寻，检查输出字符串是win还是lose
    # state.posix.dumps(1)获得所有标准输出
    # state.posix.dumps(0)获得所有标准输入
    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
    #  进行符号执行得到想要的状态，即得到满足correct条件且不满足wrong条件的state
    sm.explore(find=correct, avoid=wrong)

    # 也可以写成下面的形式，直接通过地址进行定位
    # sm.explore(find=0x80484e3, avoid=0x80484f5)

    # 获得到state之后，通过solver求解器，求解u的值
    # eval_upto(e, n, cast_to=None, **kwargs) 求解一个表达式指定个数个可能的求解方案
e - 表达式  n - 所需解决方案的数量
```

```
    # eval(e, **kwargs) 评估一个表达式以获得任何可能的解决方案。 e - 表达式
    # eval_one(e, **kwargs) 求解表达式以获得唯一可能的解决方案。 e - 表达式
    return sm.found[0].solver.eval_upto(u, 256)

def test():
    good = set()
    for u in range(256):
        bits = [0, 0]
        for i in range(8):
            bits[u&(1<<i)!=0] += 1
        if bits[0] == bits[1]:
            good.add(u)
    res = main()
    assert set(res) == good

if __name__ == '__main__':
    print(repr(main()))#repr()函数将对象类型转化为字符串类型
```

此示例是展示关于“符号写地址”的用法，程序的核心逻辑如下：

1. 对地址 `0x804a021` 进行写操作，写入的是一个 **8 位符号变量** `u`；
2. 然后对程序进行路径探索（`explore()`），寻找会输出 `b'win'` 的路径，并避开输出 `b'lose'` 的路径；
3. 最后将 `u` 的所有可能值（即使得程序输出 `win` 的 `u` 值）返回出来

在pycharm中运行结果：

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

```
D:\PycharmProjects\PythonProject\.venv\Scripts\python.exe D:\tools\angr-doc-master\example
WARNING  | 2025-05-08 10:48:08,191 | angr.storage.memory_mixins.default_filler_mixin | The
WARNING  | 2025-05-08 10:48:08,191 | angr.storage.memory_mixins.default_filler_mixin | ang
WARNING  | 2025-05-08 10:48:08,191 | angr.storage.memory_mixins.default_filler_mixin | 1)
WARNING  | 2025-05-08 10:48:08,191 | angr.storage.memory_mixins.default_filler_mixin | 2)
WARNING  | 2025-05-08 10:48:08,191 | angr.storage.memory_mixins.default_filler_mixin | 3)
WARNING  | 2025-05-08 10:48:08,191 | angr.storage.memory_mixins.default_filler_mixin | Fil
WARNING  | 2025-05-08 10:48:08,192 | angr.storage.memory_mixins.default_filler_mixin | Fil
[51, 57, 60, 240, 75, 139, 78, 197, 23, 142, 90, 29, 209, 154, 212, 99, 163, 102, 108, 166
```

图4 solve.py运行结果截图

其中：

u = 51 → 二进制为 00110011，包含四个 1 和四个 0

u = 60 → 00111100，也是四个 1 和四个 0.均正确！

5 第二种方法

solve2.py源代码:

```python
#!/usr/bin/env python
# coding=utf-8
import angr
import claripy

def hook_demo(state):
    state.regs.eax = 0

p = angr.Project("./issue", load_options={"auto_load_libs": False})
# hook函数：addr为待hook的地址
# hook为hook的处理函数，在执行到addr时，会执行这个函数，同时把当前的state对象作为参数传递过去
# length 为待hook指令的长度，在执行完 hook 函数以后，angr 需要根据 length 来跳过这条指令，执行下一条指令
# hook 0x08048485处的指令（xor eax,eax），等价于将eax设置为0

# hook并不会改变函数逻辑，只是更换实现方式，提升符号执行速度 ，起到复杂程序分析的替代功能
p.hook(addr=0x08048485, hook=hook_demo, length=2)
#程序的入口点指定起始地址
state = p.factory.blank_state(addr=0x0804846B, add_options={"SYMBOLIC_WRITE_ADDRESSES"})
#定义符号变量
u = claripy.BVS("u", 8)
state.memory.store(0x0804A021, u)
sm = p.factory.simulation_manager(state)
sm.explore(find=0x080484DB)
st = sm.found[0]
#打印一个结果
print(repr(st.solver.eval(u)))
```

原始程序中0x8048485处的指令入为xor eax,eax，更换实现方式为hook_demo,但是并没有改变原始程序的逻辑，只是提升符号执行的速度。

图5 原始程序在0x8048485处的指令

由图6可以看出，此时只有一个结果输出，$226 = (11100010)_2$,结果正确。

```
D:\PycharmProjects\PythonProject\.venv\Scripts\python.exe D:\PycharmProjects\PythonProject\solve2.py
WARNING  | 2025-05-08 11:03:23,473 | angr.storage.memory_mixins.default_filler_mixin | The program i
WARNING  | 2025-05-08 11:03:23,473 | angr.storage.memory_mixins.default_filler_mixin | angr will cop
WARNING  | 2025-05-08 11:03:23,473 | angr.storage.memory_mixins.default_filler_mixin | 1) setting a
WARNING  | 2025-05-08 11:03:23,473 | angr.storage.memory_mixins.default_filler_mixin | 2) adding the
WARNING  | 2025-05-08 11:03:23,473 | angr.storage.memory_mixins.default_filler_mixin | 3) adding the
WARNING  | 2025-05-08 11:03:23,473 | angr.storage.memory_mixins.default_filler_mixin | Filling memor
WARNING  | 2025-05-08 11:03:23,474 | angr.storage.memory_mixins.default_filler_mixin | Filling regis
226

进程已结束，退出代码为 0
```

图6 第二种方法运行结果截图

6 对比分析

上述代码与前面的解法有三处区别：

（1）采用了hook函数，将0x08048485处的长度为2的指令通过自定义的hook_demo进行替代，功能是一致的，原始xor eax, eax和state.regs.eax = 0是相同的作用。

hook 是对*“具体实现”*的替代，不是对*“逻辑”*的更改，在实际复杂程序中（如：printf, strcmp, libc函数、IO等），很多函数无法符号执行或太耗时，使用 hook 可以：简化分析；避免路径爆炸；提高分析速度；更易控制具体的行为（自定义返回值/状态等）

（2）进行符号执行得到想要的状态，有变化，变更为find=0x080484DB。因为源程序win和lose是互斥的，所以，只需要给定一个find条件即可。**从内容识别到地址识别**，性能更快：不再读取 IO 缓存、比较字符串；更明确：angr 不需要解释程序语义，只需要走到目标地址；易调试：能精确控制目标路径位置。

（3）最后，eval(u)替代了原来的eval_upto，将打印一个结果出来。

7 angr 实际应用探讨

7.1 angr的使用

angr是一个基于 Python 开发的多分析器平台，结合了静态分析、动态符号执行（symbolic execution）、路径敏感分析、约束求解等能力。它适用于分析各种架构的 ELF、PE 等格式的可执行文件，支持 Linux、Windows、ARM、x86、x64 等平台。

核心组件包括：

- **Project**：分析的核心对象，用于加载二进制程序。
- **State**：代表某一时刻程序执行状态（寄存器、内存、输入等）。
- **SimulationManager**：路径管理器，管理多个 state（执行路径）。
- **Exploration Techniques**：探索策略，用于控制路径执行逻辑。
- **Solver（claripy）**：符号约束求解器，用于处理符号变量的约束逻辑。

7.1.1 基本使用流程

1. 安装 angr：可以使用 pip 安装，建议使用虚拟环境，或者安装 angr 提供的 Docker 镜像进行隔离。

```
pip install angr
```

2.加载项目：

```
import angr

proj = angr.Project('./target_binary', auto_load_libs=False)
```

参数 `auto_load_libs=False` 可避免加载动态链接库，提高分析速度。

3.创建初始状态：

```
state = proj.factory.entry_state()  # 默认从入口点开始
```

也可以使用自定义地址或符号输入状态：

```
state = proj.factory.blank_state(addr=0x400610)
```

4. 定义符号变量

```
import claripy

sym_arg = claripy.BVS('input', 8 * 8)  # 8 字节符号变量
state.memory.store(0x601050, sym_arg)  # 将其写入内存地址
```

或者定义符号化 `stdin` 输入：

```
state = proj.factory.full_init_state(stdin=claripy.BVS('input', 8*20))
```

5. 执行路径探索

```
simgr = proj.factory.simulation_manager(state)

def is_successful(s):
    return b'Success' in s.posix.dumps(1)

simgr.explore(find=is_successful)
```

也可以指定 `find` 和 `avoid` 的地址：

```
simgr.explore(find=0x400700, avoid=0x400800)
```

6.约束求解

```
found = simgr.found[0]
solution = found.solver.eval(sym_arg, cast_to=bytes)
print("Solved input:", solution)
```

如果是多个符号值，可以使用 `eval_upto`：

```
results = found.solver.eval_upto(sym_arg, 5, cast_to=bytes)
```

7.1.2 常用功能模块

功能	模块
符号执行引擎	angr.SimulationManager, angr.State
约束求解	claripy
控制流图恢复	proj.analyses.CFG()
函数识别	proj.kb.functions
Hook 指令	proj.hook(addr, hook_func, length)
路径策略	simgr.use_technique(angr.exploration_techniques.DFS())

7.1.3 使用技巧

1. 使用 Hook 加速复杂调用

```
def dummy_printf(state):
    return

proj.hook(0x400800, dummy_printf, length=5)
```

跳过如 `printf`，`strcmp` 等调用，提升执行效率。

2. 使用路径策略控制执行顺序

```
from angr.exploration_techniques import DFS, LoopSeer

simgr.use_technique(DFS())        # 深度优先策略
simgr.use_technique(LoopSeer(5))  # 限制循环次数
```

3. 控制符号约束与路径爆炸

路径爆炸问题是符号执行的瓶颈，可以通过：

- 限制符号变量长度
- 使用 `LoopSeer` 控制分支深度
- 使用 `state.solver.simplify()` 简化表达式

- 针对性 `prune` 无关路径

7.2 angr的实际应用

angr 在多个实际场景中都具有极高的应用价值。其最根本的优势在于能够以state为核心，对执行路径进行枚举、约束求解和分支探索，因此在输入生成、安全验证、漏洞挖掘、逆向工程等多个方向上有着极为广泛的用途。

1. 在输入自动生成方面：angr 可用于自动化求解满足某段程序逻辑的输入，典型的例子如破解一个验证程序的密码逻辑，只需将输入定义为符号变量，再将程序输出中出现 `"Correct"` 或 `"Access granted"` 等提示字符串作为路径目标（`find`），angr 就能自动探索出到达这条路径所需的所有输入值。这种机制广泛应用于 CTF 比赛和安全课程练习中，可大大降低人工逆向分析的工作量。
2. 在漏洞路径分析方面：angr 能追踪程序中的条件判断，提取输入与状态间的约束关系，从而识别哪些输入会触发内存溢出、整数溢出、格式化字符串等安全漏洞。例如，当程序中存在一个写地址是输入控制的 `memcpy`，我们就可以通过符号化该输入地址并设置目标状态为某个非法写（如修改 GOT 表）的位置，从而自动生成可触发漏洞的精确输入。
3. 为了提升分析效率，angr 提供了 hook 机制，这对分析包含复杂库函数、系统调用或不可用符号的程序尤其重要，在本实验的第二种方法中也用到。通过在目标地址设置 hook，可以跳过分析实际逻辑复杂的代码，只模拟其行为，节省大量路径搜索资源。例如，可将 `printf`、`strcpy`、`malloc` 等系统调用以简化模型代替，从而集中资源在更关键的控制逻辑上。
4. 在程序逆向辅助方面：angr 的控制流图恢复和函数识别能力，能帮助逆向人员在无调试符号的二进制中识别函数边界、函数调用关系、基本块等静态结构，进而结合符号执行精确模拟程序行为。特别是在处理反调试、反虚拟机等保护机制时，angr 提供的路径跟踪工具能够揭示隐藏的程序分支，是现代逆向工程不可或缺的工具之一。
5. 除此之外，angr 在补丁比较、自动漏洞利用、模糊测试强化等场景中也有重要作用。通过比较补丁前后的二进制，在同样的符号输入下分析行为差异，可以精准评估补丁是否彻底修复漏洞。结合 AFL、LibFuzzer 等模糊测试工具，angr 还可以在模糊执行陷入路径瓶颈时接管符号执行任务，计算出能绕过条件检查的新输入，从而提升模糊测试的路径覆盖率与效率。

总之，angr 具备足够的灵活性与能力支撑真实世界中自动化安全审计、恶意代码分析、IoT 固件安全验证等高强度、高精度的工作需求，是现代二进制安全研究工具体系中的核心组件之一。

8 心得体会

本次实验围绕二进制分析框架 **angr** 展开，实验的主体任务是复现一个使用符号写地址的样例程序（sym-write）的两种解法。通过实际分析sym-write案例，深入探索了 angr 的使用方法、功能机制以及实际应用场景。

在分析过程中，SYMBOLIC_WRITE_ADDRESSES使 angr 能够正确处理内存写操作中目标地址为符号变量的情况。这类问题在一般二进制分析中较难捕捉，angr 的强大符号执行能力使得此类复杂路径问题得以求解。通过设置了 find 和 avoid 条件，用来控制符号执行引擎寻找“win”路径并避开“lose”路径。路径控制不仅提高了求解效率，也体现了 angr 在状态空间管理方面的灵活性。

在第二种求解方法中，使用了 angr 的 hook功能，对程序中某些指令进行了重定义。这种方式在处理实际程序中的系统调用或复杂函数（如 printf、malloc等）时非常实用。通过 hook，我们也可以跳过这些非关键逻辑，避免执行过程中的异常或路径爆炸，同时保留核心逻辑不变。本实验通过hook成功替代原始指令行为，提高了执行效率，验证了 angr 在动态插桩与行为模拟方面的强大能力。进一步，利用 claripy构建了符号变量 u，并将其写入指定内存地址，通过求解得到可使程序进入“win”路径的具体字节值。这种由输入反推出路径条件的能力，使 angr 在 crackme 分析、逆向工程、输入验证、漏洞利用等场景中尤为实用。

综上，angr 提供了一种以状态为中心的分析范式。与传统以指令为驱动的调试器或反汇编工具不同，angr 更关注路径、输入、状态之间的关系。这种“路径导向+符号求解”的分析模型，不仅适用于 crackme 和 CTF 场景，更在安全审计、自动化测试等领域具有广泛应用价值。尽管符号执行面临路径爆炸问题，但通过合理建模、hook 非关键逻辑、限制符号范围、裁剪路径状态，我们可以有效控制计算成本，得到有意义的输入条件。

通过本次实验，我深刻体会到 angr 的强大与灵活。它不仅是一个程序分析框架，更是一种通用的自动化逻辑验证工具。angr 与其它静态反汇编工具（如IDA）的配合使用，也将为逆向分析带来更多可能。