为什么漏洞挖掘这么难?

王若愚
Arizona State University

自我介绍

- 亚利桑那州立大学（Arizona State University）助理教授
- angr 的创始人之一
- CTF 赛棍，Shellphish 现任队长
  - 专做逆向题
- DARPA CGC 决赛第三名

基础知识

声明

- 主要的讨论目标是二进制程序
  - 编译后的可执行文件
  - 没有源码，没有调试符号
- 示例大部分是源码
  - 因为大概没有人想读汇编代码
- “Evaluating Fuzz Testing [CCS 18]”

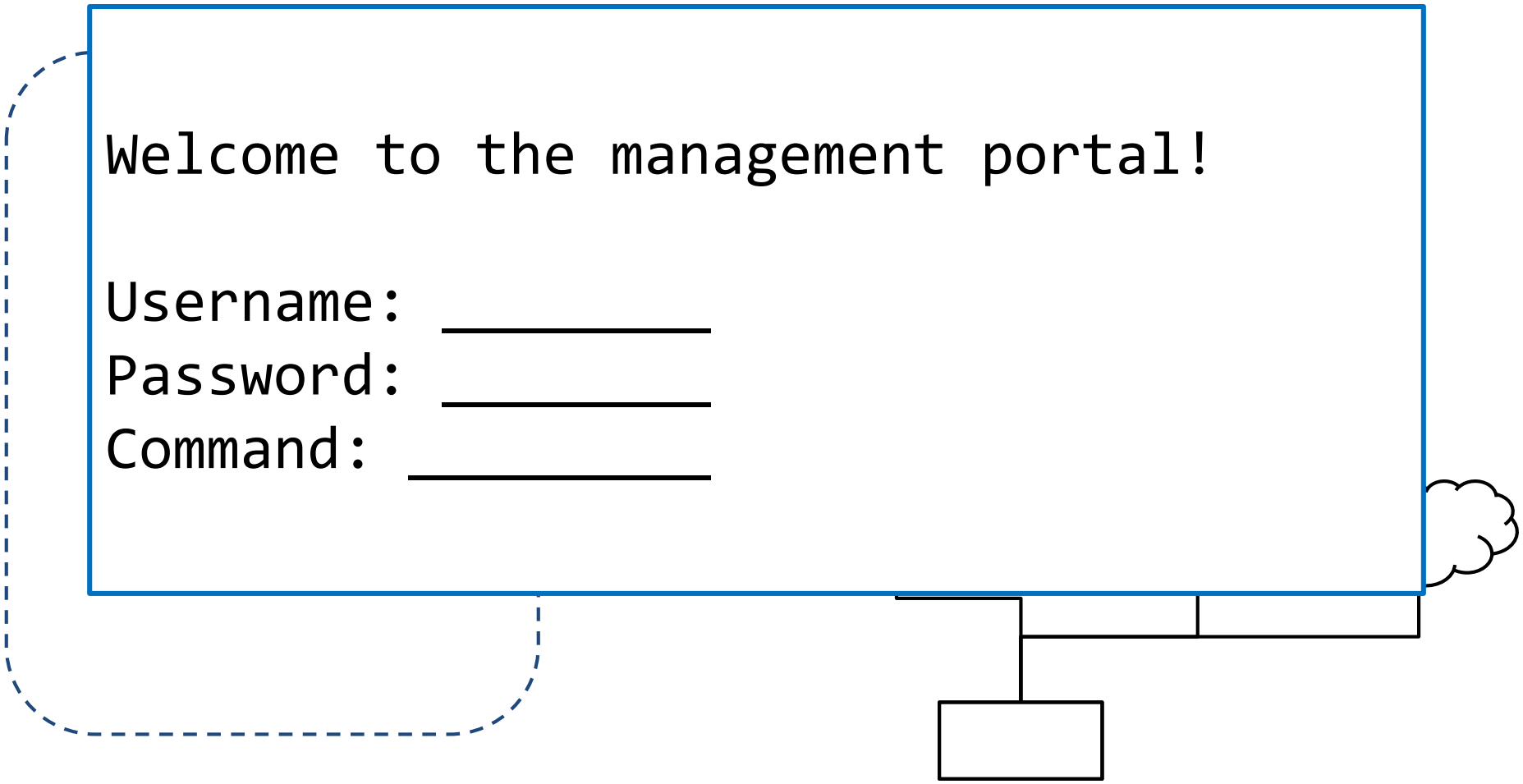
漏洞示例

- CVE-2014-1266 – Apple “goto fail”

函数 SSLVerifySignedServerKeyExchange():

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;

 ... other checks ...

 fail:
    ... buffer frees (cleanups) ...
    return err;
```

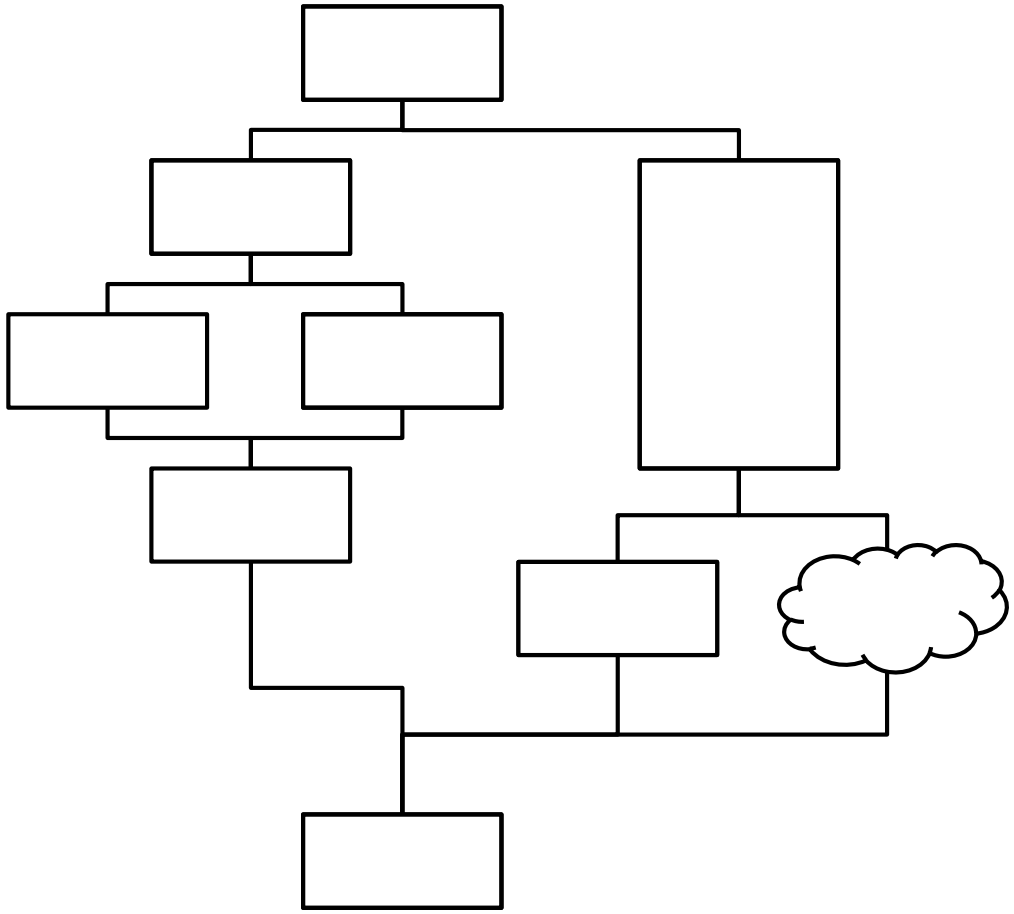
什么是 fuzzing?


Welcome to the management portal!
Username:
Password:
Command:

什么是 fuzzing?

测试输入

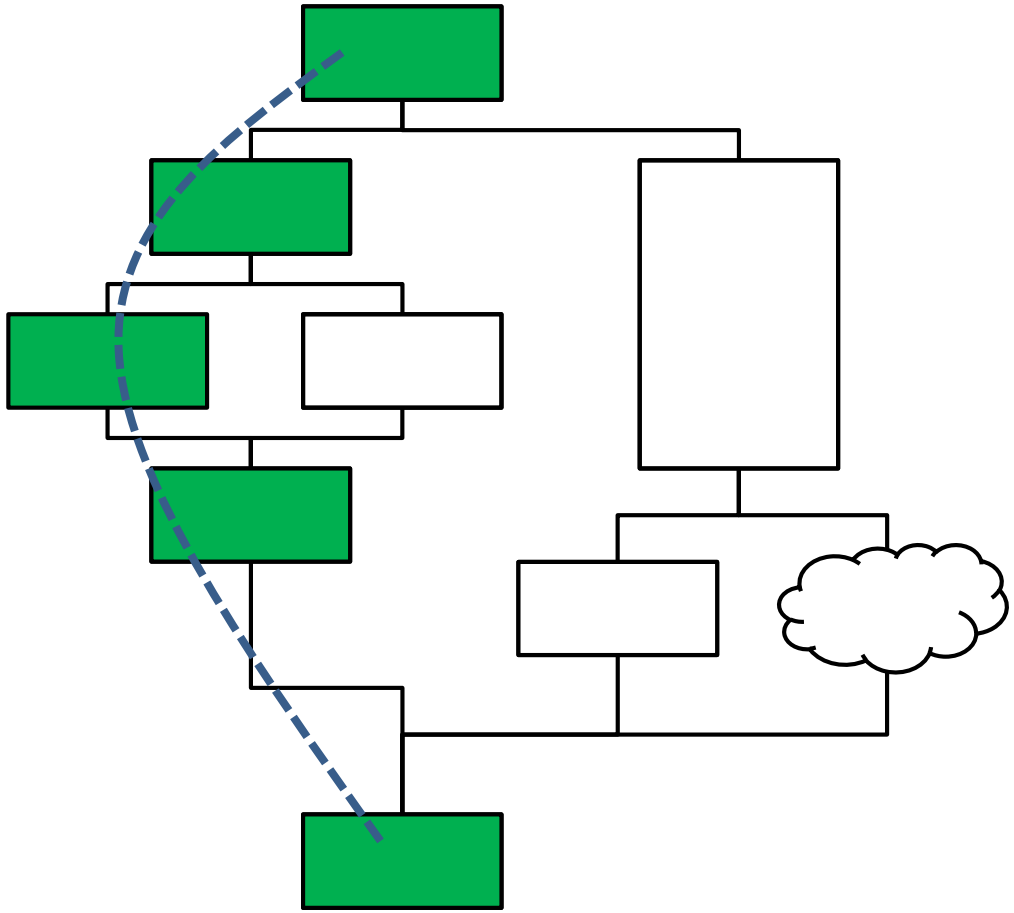
```
test:aaaaa:?
```



什么是 fuzzing?

测试输入

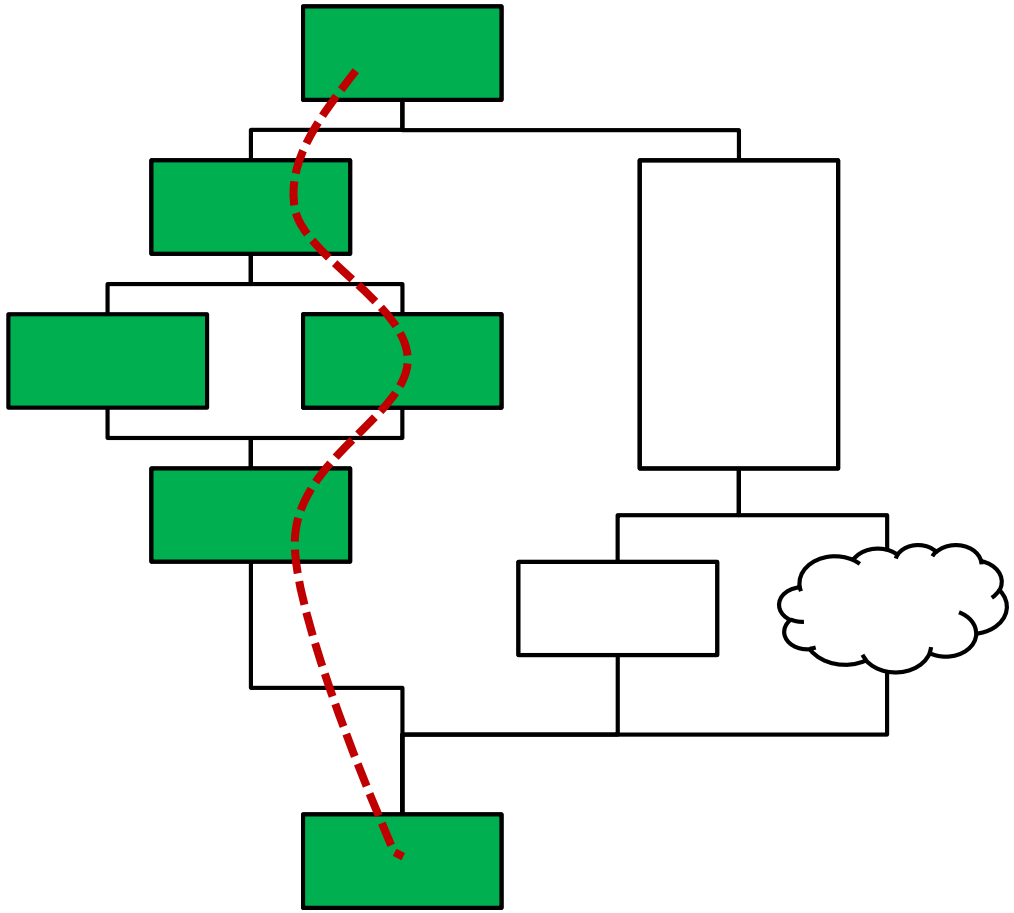
```
test:aaaaa:?

tes:bb:help
```



什么是 fuzzing?

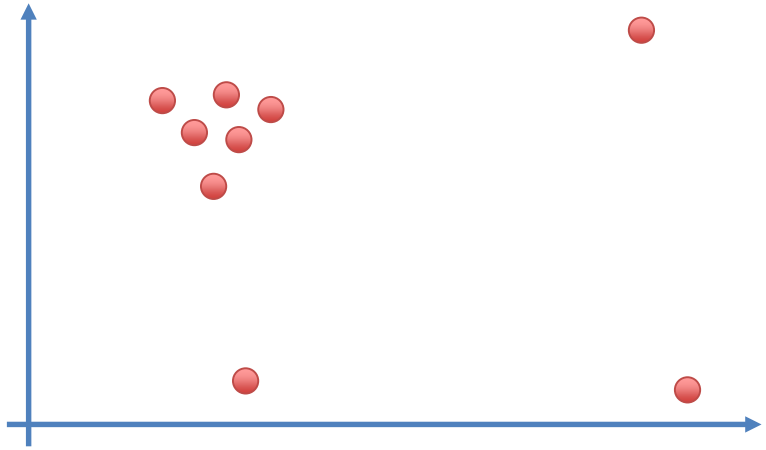
测试输入

```
test:aaaaa:?
tes:bb:help
aes:bc:heOp
```



Fuzzing 有效性的保证

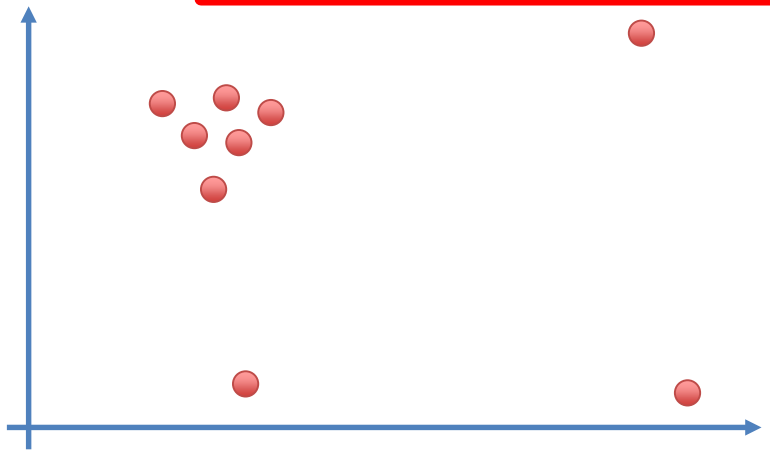
- 通过提供不同的输入，覆盖尽可能多的程序状态，尽早触发问题



- 高效率的输入变异
- 低开销的反馈：由程序覆盖率作为指引

Fuzzing ~~有效性的保证~~

- 通过提供不同的输入，覆盖尽可能多的程序状态，尽早触发问题



- 高效率的输入变异
- 低开销的反馈：由程序覆盖率作为指引

程序覆盖率和程序状态空间

```
int main()
{
    int win_count = 0;
    int scores[256] = {0};
    int i = 0;
    while (1) {
        int score = play();
        win_count += score > 0? 1: 0;
        scores[i++] = score;
    }
    return 0;
}
```

如何发现漏洞?

我们是如何挖洞的?

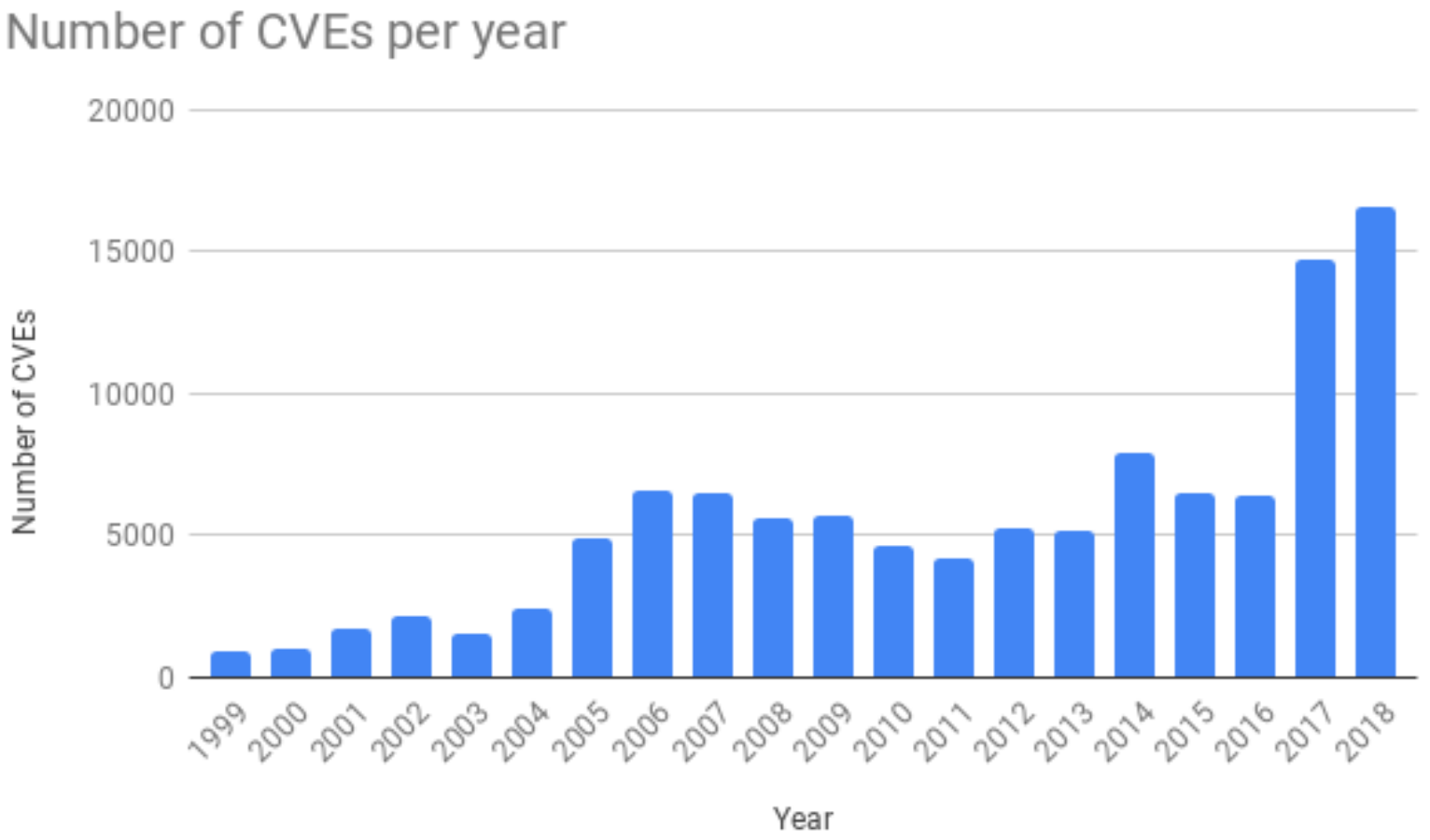
人工/手动

Fuzzing

符号执行

静态分析

漏洞报告数量突增



来源： https://cvedetails.com

Fuzzers

名称	需要源码?	时间	开源?	优势
Spike			是	
Peach			是/否	
radamsa		2011	是	
Sulley		2012	是	
AFL	是/否	2014	是	速度快，配置少，通用性好，极其稳定、极其健壮
libFuzzer	是	2016	是	

Fuzzers

名称	需要源码?	时间	开源?	优势
Spike			是	
Peach			是/否	
radamsa		2011	是	
Sulley		2012	是	
AFL	是/否	2014	是	速度快，配置少，通用性好，极其稳定、极其健壮
libFuzzer	是	2016	是	

我们是如何挖洞的?

人工/手动

Fuzzing

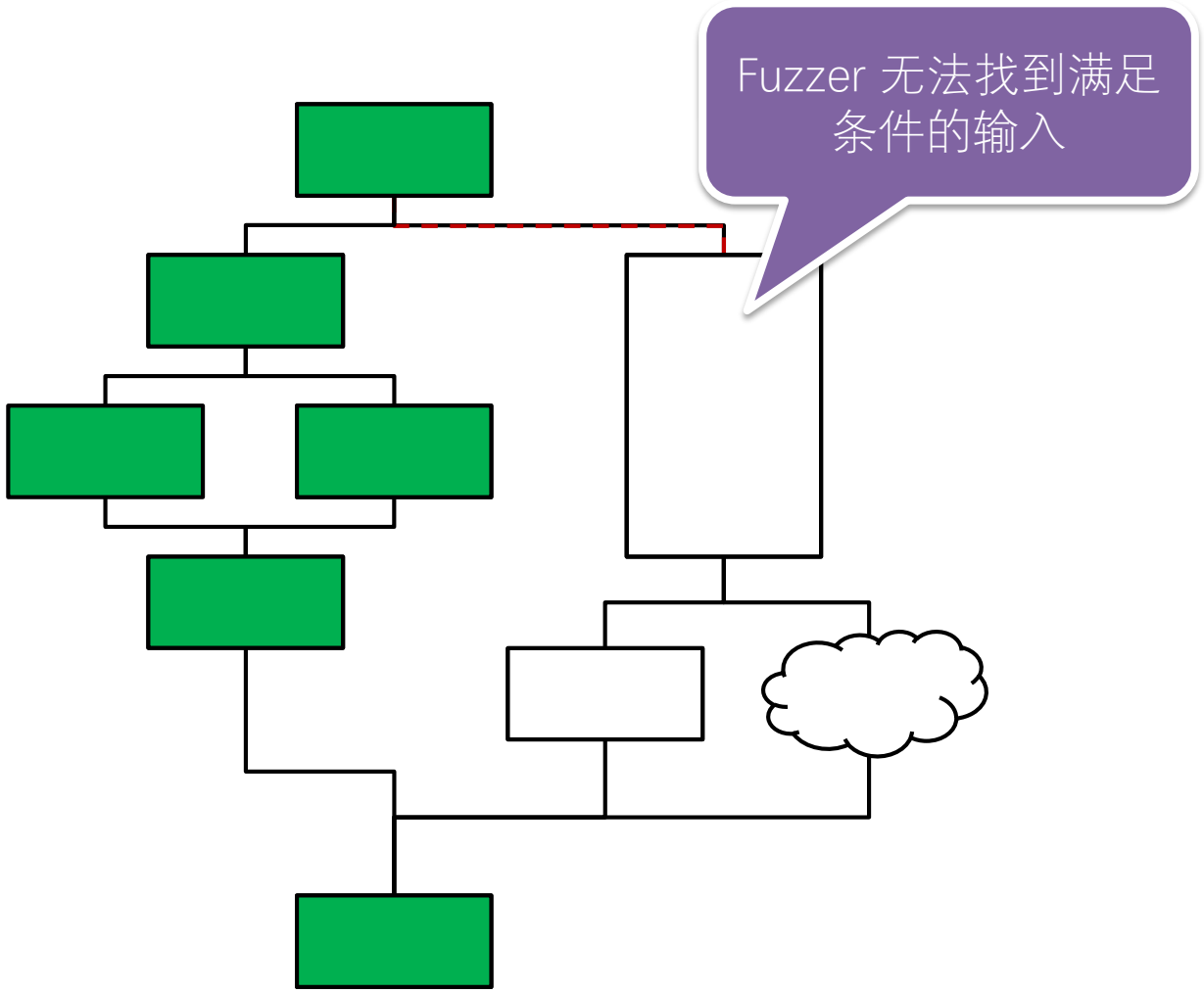
符号执行

静态分析

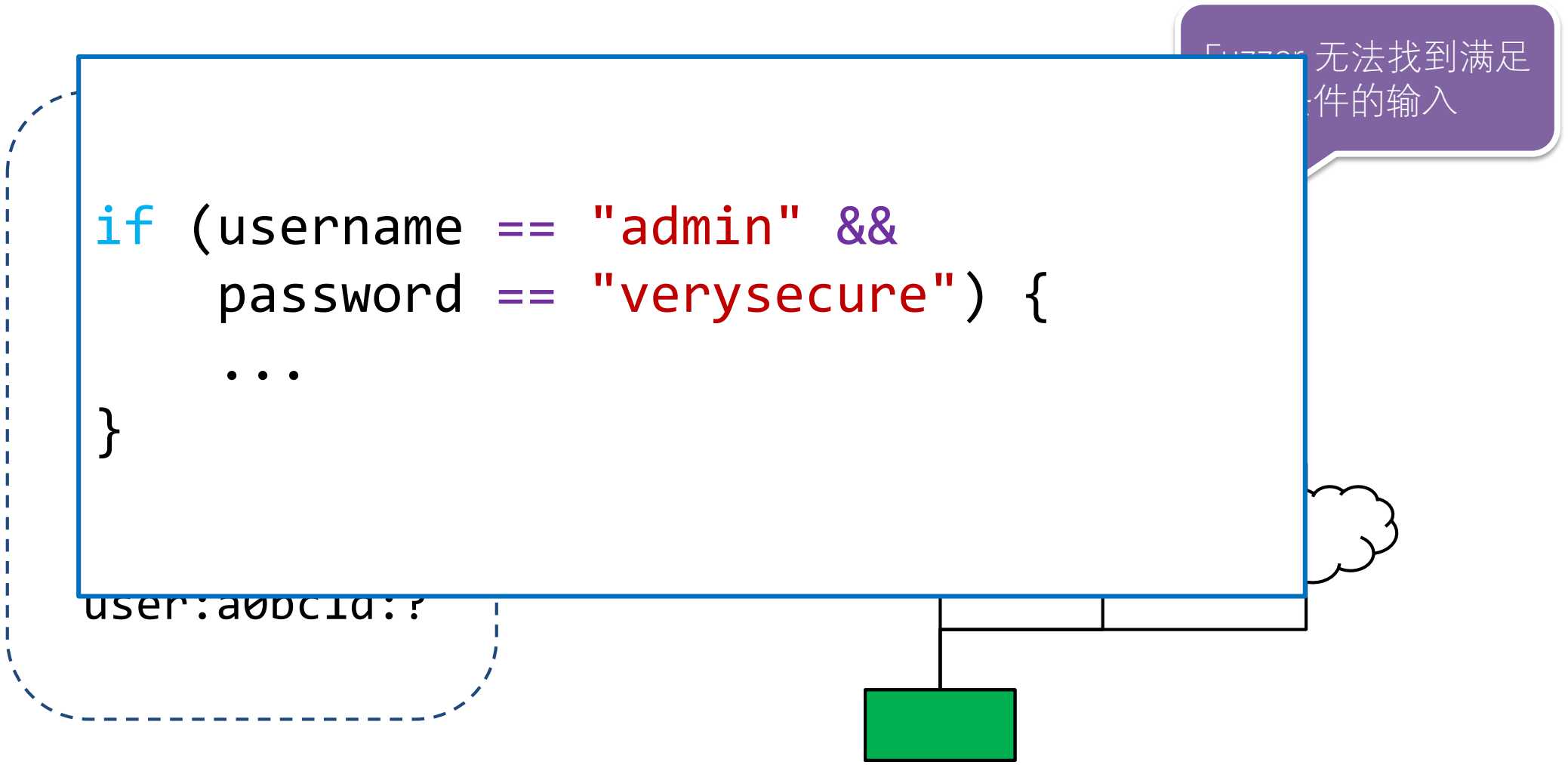
什么是符号执行?

测试输入

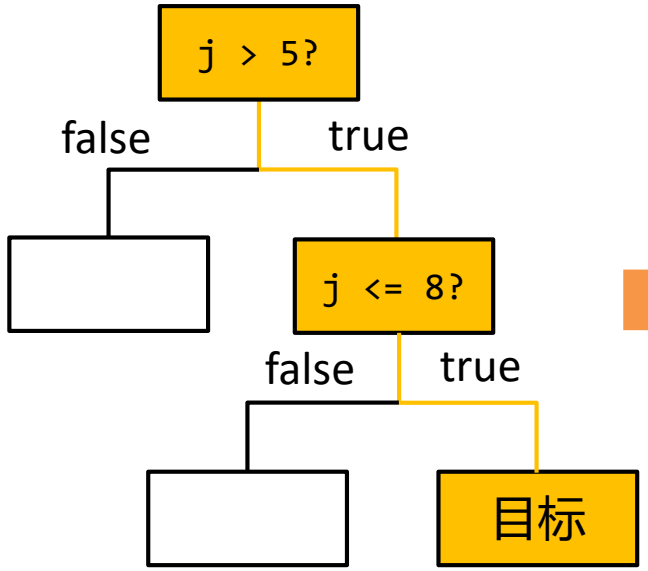
```
test:aaaaa:?
tes:bb:help
admin:00000:a
%^$@*9!
user:a0bc1d:?
```



什么是符号执行?



约束收集与求解



沿着一条执行路径收集约束信息

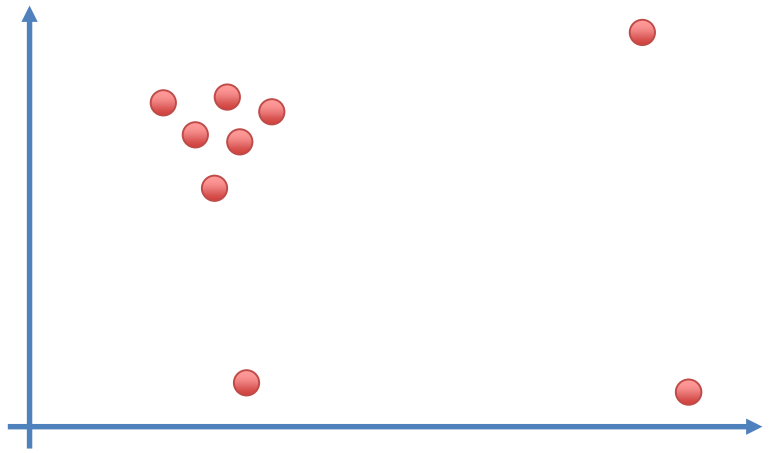
```
j > 5
j <= 8
j != 45
...
```

约束集

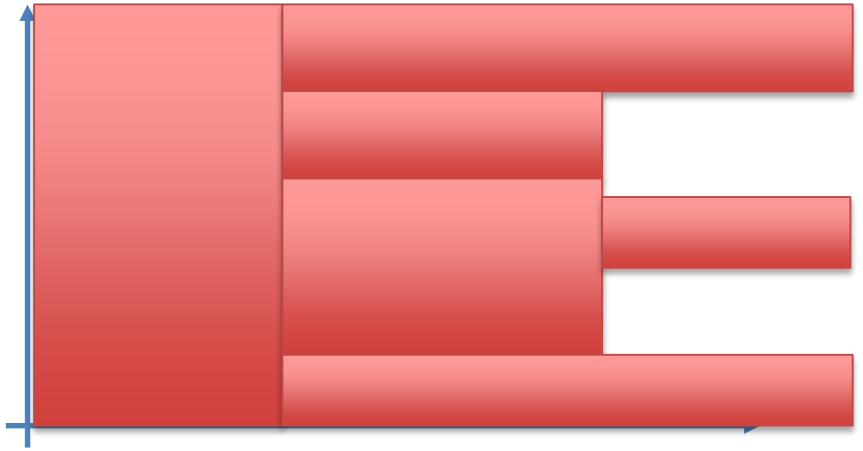
使用 SMT 求解器对约束集进行求解

```
j = 6
  or
j = 7
  or
j = 8
```

获取解（程序输入）



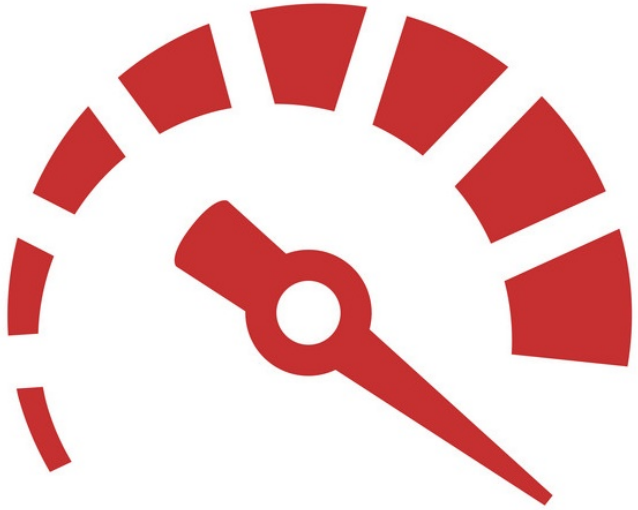
Fuzzing



符号执行

是不是不用 fuzzing 了？

- 符号执行如此强大，我们还要 fuzzing 干什么？



速度

具体分析

Fuzzing 和符号执行的速度差异

/usr/bin/readelf (amd64, Ubuntu 18.04 LTS)
测试环境：我的笔记本电脑
目标命令：readelf –S
种子输入：/bin/ls
angr 模式：符号追踪（symbolic/concolic tracing）

$2x = 3$
假设 x 的大小为 16 bits

测试项目	执行时间	执行次数	平均每秒执行次数
AFL + Qemu	120 秒	22.9k	180.0000
angr (CPython)	3580 秒	1	0.0002
angr (PyPy)	1142 秒	1	0.0008
angr (CPython,Unicorn)	626 秒	1	0.0016
angr (PyPy, Unicorn)	249 秒	1	0.0040

题外话：如何评估 fuzzer 的性能？

- 参考论文 Evaluating Fuzz Testing [CCS 18]

为什么 fuzzing 这么快?

- Fuzzing 不是一直都这样快的

```
Speed. It's genuinely hard to compete with brute force when your "smart"
   approach is resource-intensive. If your instrumentation makes it 10x more
   likely to find a bug, but runs 100x slower, your users are getting a bad
   deal.

   To avoid starting with a handicap, afl-fuzz is meant to let you fuzz most of
   the intended targets at roughly their native speed - so even if it doesn't
   add value, you do not lose much.
```

来源：http://lcamtuf.coredump.cx/afl/historical_notes.txt

Fuzzing 的适用条件

- 目标程序的执行速度接近于本地执行的速度
  - 同指令集或 JIT 往往是必要的
- 高质量的种子输入
- 速度极快的反馈机制
  - 程序崩溃
  - 内存/资源泄漏
  - 对于无法迅速反馈的问题（例如大多数逻辑漏洞）的效果较差

我们还要符号执行干什么?

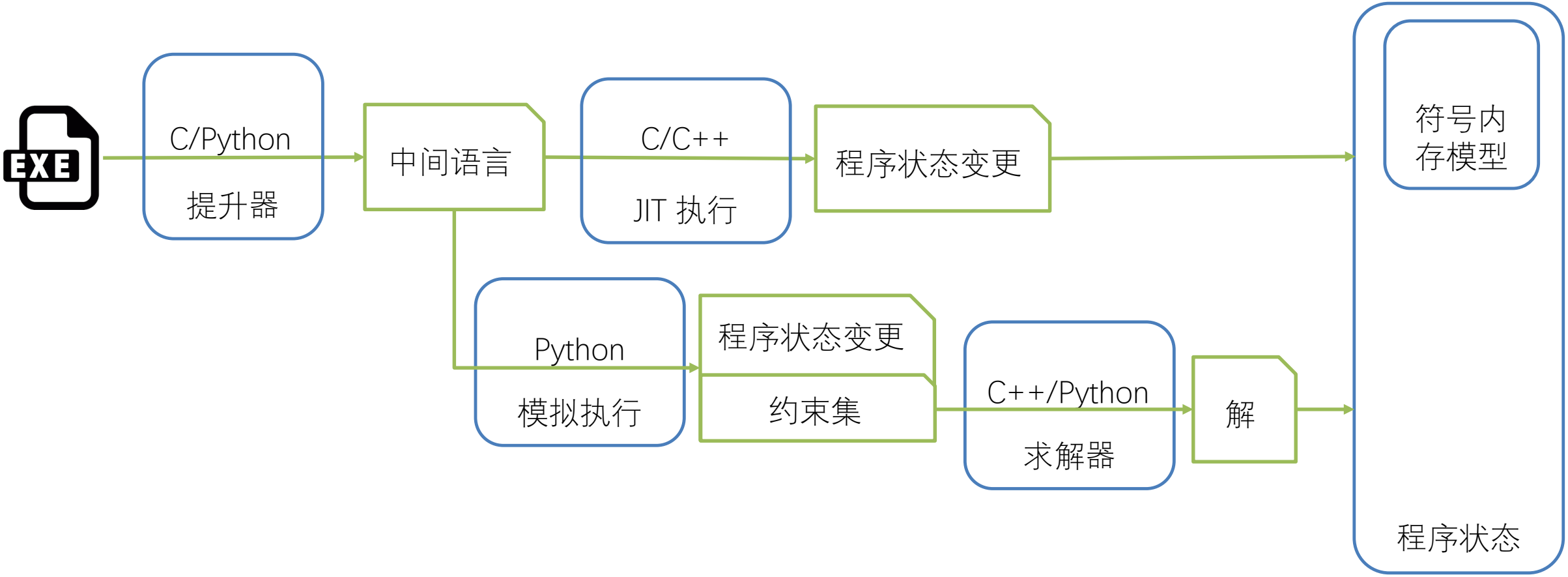
- 符号执行还是有很多优点的
  - 模拟执行
    - 收集执行条件，可以找到不存在“问题反馈”（比如无程序崩溃）的问题
    - 轻松地进行跨指令集和跨平台分析
    - 执行与分析程序抽象
    - 分析不完整的程序
    - 在原始环境不可用时进行分析
  - 符号求解
    - 从约束条件出发求解（针对性强）
    - 同时分析多条路径与多种输入
    - （某些情况下）可以保证测试的完备性

设计一个更快的符号执行平台

为什么符号执行这么慢?

- 根本原因
  - 模拟执行 vs 本地执行
  - 约束求解
  - 其它无法避免的开销
- 其它原因
  - 设计问题
  - 实现问题
  - 优化不足

angr 符号执行引擎架构



从模拟执行到本地执行

- 基本思路
  - 在CPU上执行尽可能多的指令
  - 遇到符号变量时切换回模拟执行环境
  - angr + Unicorn Engine
- 更进一步
  - 符号变量的复制操作可以在本地执行环境中完成
- 再进一步
  - 符号变量的基本运算也可以在本地执行环境中完成
- 还有吗？

减小 Python 的开销

- angr 选用 Python 是因为
  - 容易上手
  - 开发速度快，难度低
  - 灵活
  - 跨平台
- Python (CPython) 的性能不佳
  - CPython 运行时太慢
  - 标准实现是解释器（PyPy 是一个很好的尝试，但并不够）
  - 语言过于灵活，优化空间小
  - 各种开销（比如 boxing）属于历史包袱，难以移除
  - 多线程/多进程支持不适合密集运算场景

减小 Python 的开销

- 基本思路
  - 从 CPython 切换到 PyPy
  - 先做性能测试，基于性能测试优化实现
    - cProfile + snakeviz
    - vmprof
    - 常见优化：__slots__，用 type() 替换 isinstance()，不要用 Exception 实现跳出循环
  - 烂代码是第一大性能杀手

烂代码 A

```
class SimState:
    @property
    def arch(self):
        if self._is_java_jni_project:
            return self._arch['soot'] if self.ip_is_soot_addr else self._arch['vex']
        else:
            return self._arch
```

`SimState.arch` 被频繁调用，导致在某次符号追踪的性能测试中占用了 5% 的运行时间。

烂代码 B

```
class StateOption:
    def one_type(self):
        if len(self.types) == 1:
            return next(iter(self.types))

        return None
```

StateOption.one_type() 被频繁调用，导致在某次符号执行的性能测试中占用了 7% 的运行时间。

烂代码 C

```
# Prioritize returning functions
for func_addr in self._returning_functions:
    if func_addr not in self._jobs:
        continue
```

在较大的二进制程序中，_returning_functions 可能有超过 50 万项，而 _jobs 往往只有几百项。

```
# Prioritize returning functions
for func_addr in self._jobs:
    if func_addr not in self._returning_functions:
        continue
```

仅靠优化 Python 代码实现的性能提升

- angr/angr PR #1459
  - 通过性能测试，优化了许多较差的实现
  - 测试基准：perf_concrete_execution.py + CPython + cProfile

ID	提交 hash	测试耗时	累积性能提升
1	5a762e7	128.4	0.0%
2	5380542	123.7	3.7%
3	2000b7b	118.5	7.7%
4	e64c48a	86.1	32.9%
5	ab9625b	48.0	62.6%

减小 Python 的开销

- 基本思路
  - 从 CPython 切换到 PyPy
  - 先做性能测试，基于性能测试优化实现
  - 开源库也不一定是性能最优的

Unicorn Engine 问题修复

- 内存映射对象泄漏（GitHub PR #655）
  - 单次执行时，程序的运行速度越来越慢
  - 经过性能分析发现，Unicorn Engine 使用 Qemu 中的设备内存映射机制实现了用户态的内存页面映射机制
  - 所有的映射都被加到了一个单链表中
  - 解除内存映射时，映射对象（MemoryRegion）并没有从这个单链表中移除
  - 因此未来的内存映射越来越慢

Z3 Python 绑定

- 重复的类型检查（angr/claripy GitHub PR #119）
  - Z3 提供了 Python API
  - 但是公开的 Python API 比较上层，包含很多额外的检查
  - 其中一个类型检查在性能测试时占用了 15% 的执行时间
  - 我们发现这个检查大部分情况下没有意义，对于 claripy 尤其没有意义

减小 Python 的开销

- 基本思路
  - 从 CPython 切换到 PyPy
  - 先做性能测试，基于性能测试优化实现
  - 缓存常见的对象和值
  - 将密集运算转移到 C/C++ 实现
    - 指令提升与基本块构造
    - 本地代码执行（Unicorn-engine）
    - 基本代数运算
    - 代数抽象层
    - 寄存器与内存模型

优化符号内存模型

- 在页面的粒度上实现 copy-on-write
- 针对实值（concrete values）的访问优化
  - 大部分内存读写是针对实值的
  - 完全没必要经过符号内存模型或代数抽象层的处理
  - 速度提升：~25%
- 使用更优的符号内存模型
  - MemSight [ASE 17] 提供了一个更优的模型

约束求解优化

- 减小开销
  - 降低构造约束、传递约束的开销
- 优化约束
  - 利用程序分析中的信息构造更优的约束
  - 尽可能在 Python 中完成化简
  - 平衡 ITE 树
- 优化求解过程
  - 创建独立约束集，分别求解
  - 结合其它 theories（例如 floating point theory 与 string theory）

学术研究的价值

AFL 的优势

- AFL 可能是目前最好的通用 fuzzer，但它并不是学术界的产物
- 在漏洞挖掘上，学术研究的价值是什么呢？

学术研究对 AFL 的提升

- 通用性能提升
  - 用符号执行增强 fuzzing：Driller [NDSS 2016]
  - 进一步提升符号执行和 fuzzing 结合的效果：QSYM [USENIX 2018]
  - 无需符号执行：Angora [Oakland 2018]
  - 无需符号执行：REDQUEEN [NDSS 2019]
- 增加适用的目标
  - 文件系统：Two-dimensional input space exploration [Oakland 2019]

学术研究对符号执行的提升

- 提升分析速度
- 提高分析质量
- 利用更多的信息
- 增加可分析的目标

- angr 提供了一个比较好的基础，但是远远不够
  - 不够健壮
  - 性能太差

总结

- Fuzzing 绝不是终点，也不是最终解决方案
- 基于符号执行的技术有更大的潜力
  - 工程上，缺少一个像 AFL 一样极易用、极健壮、性能极好的解决方案
  - 学术上，缺少对主要问题的全面认知
- 漏洞挖掘的未来由你来创造！

我们的研究风格

- 开放研究
  - 大部分工程上的进展都在 GitHub 上开源
  - 所有的研究成果都会在论文发表后开放

- 合作风格
  - 教授与学生紧密合作：你是我们的学生，也是我们的同事
  - 自由度高：你可以研究任何你感兴趣的问题


EXIT
1:17 AM


EXIT
2:17 AM

招聘实习生

- 目的
  - 接触团队
  - 体验科研生活
  - 与 CTF 强队一起比赛
- 地点
  - 亚利桑那州凤凰城 SEFCOM@ASU
- 时间
  - 三个月至一年均可
- 带薪吗？
  - 当然

Gail-Joon Ahn

Adam Doupe

Yan Shoshitaishvili

Tiffany Bao

fishw@asu.edu