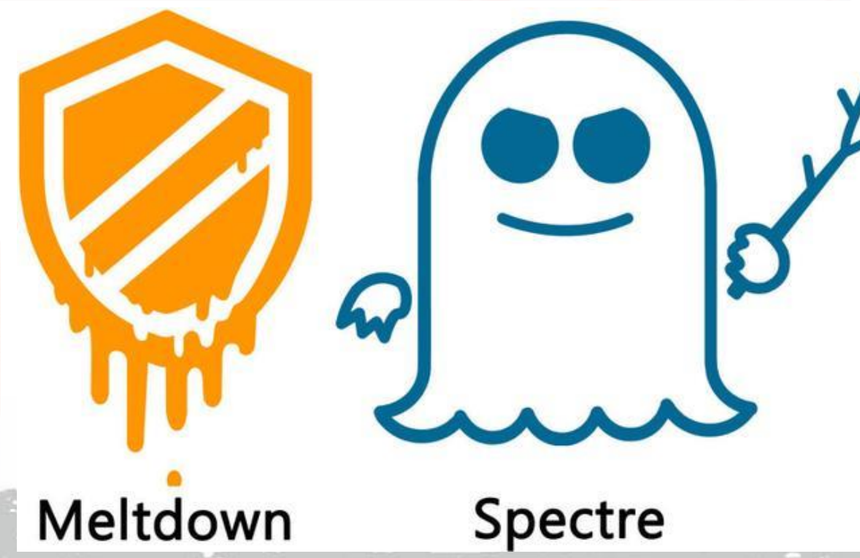
一种基于蒙特卡洛方法的混合式漏洞挖掘工具

赵磊
武汉大学网络安全学院

软件漏洞和漏洞利用

- 软件漏洞是诸多安全事件的源头，漏洞利用是主要的网络攻击手段



自动化、智能化网络攻防

- DARPA自动化网络攻防CGC
  - 自动化漏洞挖掘
  - 自动化漏洞利用
  - 自动化攻击抑制
  - 自动化漏洞修复

CYBER GRAND_CHALLENGE

自动化、智能化网络攻防

- 以漏洞为中心的网络攻防关键技术
  - 模糊测试
  - 符号执行
  - 动态污点分析

ForAllSecure

Pittsburgh, PA

Deep Red

Arlington, VA

TECHx

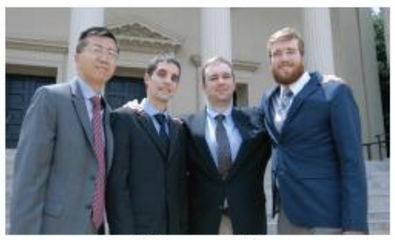
Charlottesville, VA

CYBER GRAND_CHALLENGE

Shellphish

Santa Barbara, CA

disekt



Athens, GA

Codejitsu

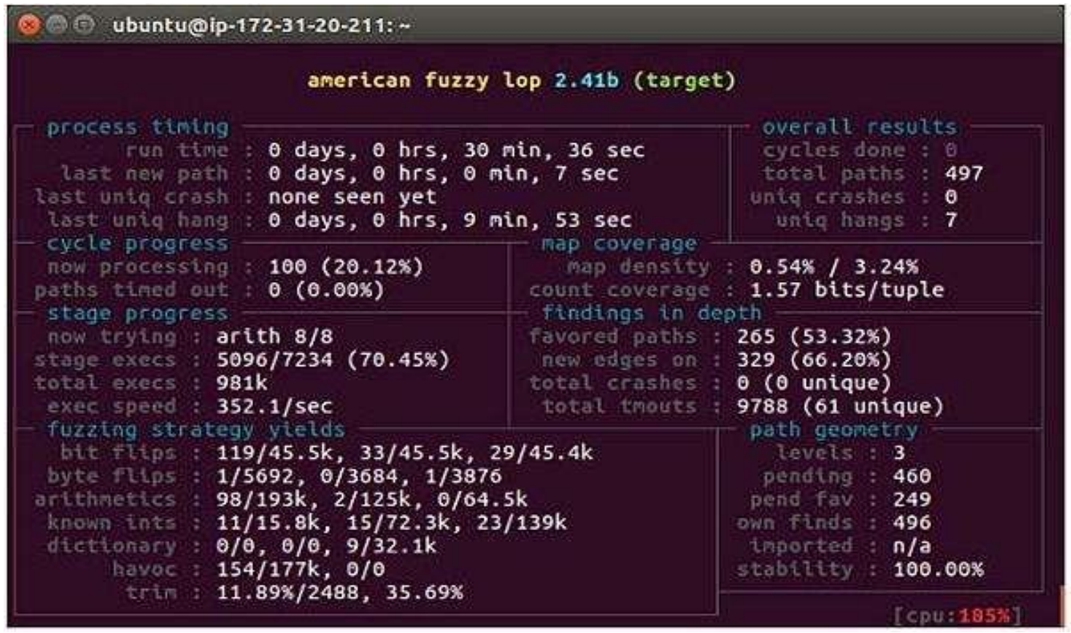
Berkeley, CA

CSDS

Moscow, ID

以Fuzzing为代表的动态分析

- Everyone is doing fuzzing 😂
  - 自2014年, 超过 60 top-tier papers (Security Top 4 + SE Top 4)

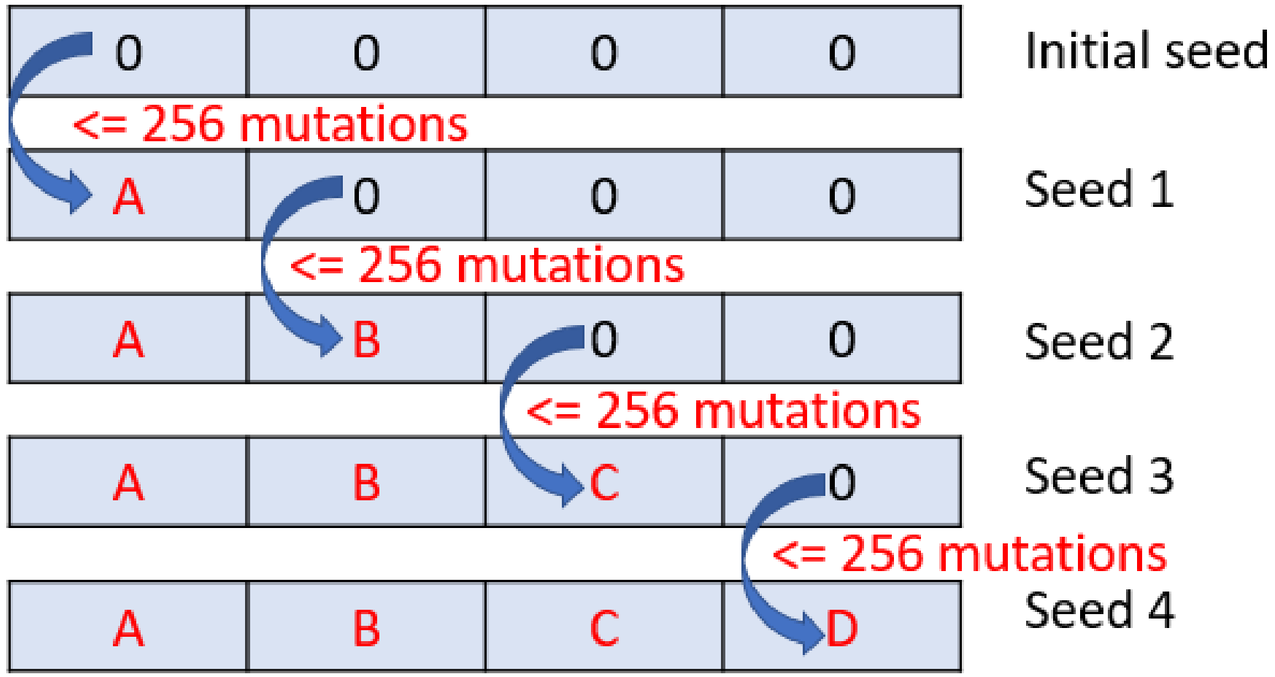


- 涌现出多个好用的工具
- AFL
- HonggFuzz

Grey-box Fuzzing

- 关键: 遍历尽可能多的程序状态

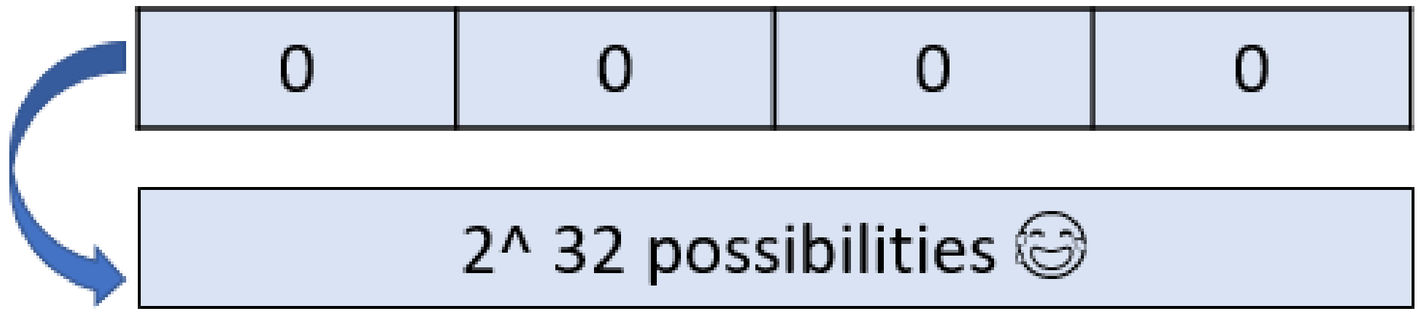
```
char buf[4];

input(buf, 4); //4-type inputs

if(buf[0] == 'A')

  if(buf[1] == 'B')

    if(buf[2] == 'C')

      if(buf[3] == 'D')

        bug();
```



Grey-box Fuzzing

- 难以遍历解空间小的状态

```
int magic;

input(&magic, 4); //32-bit integer

if (magic == 0x12345678)
      bug();
```



符号执行

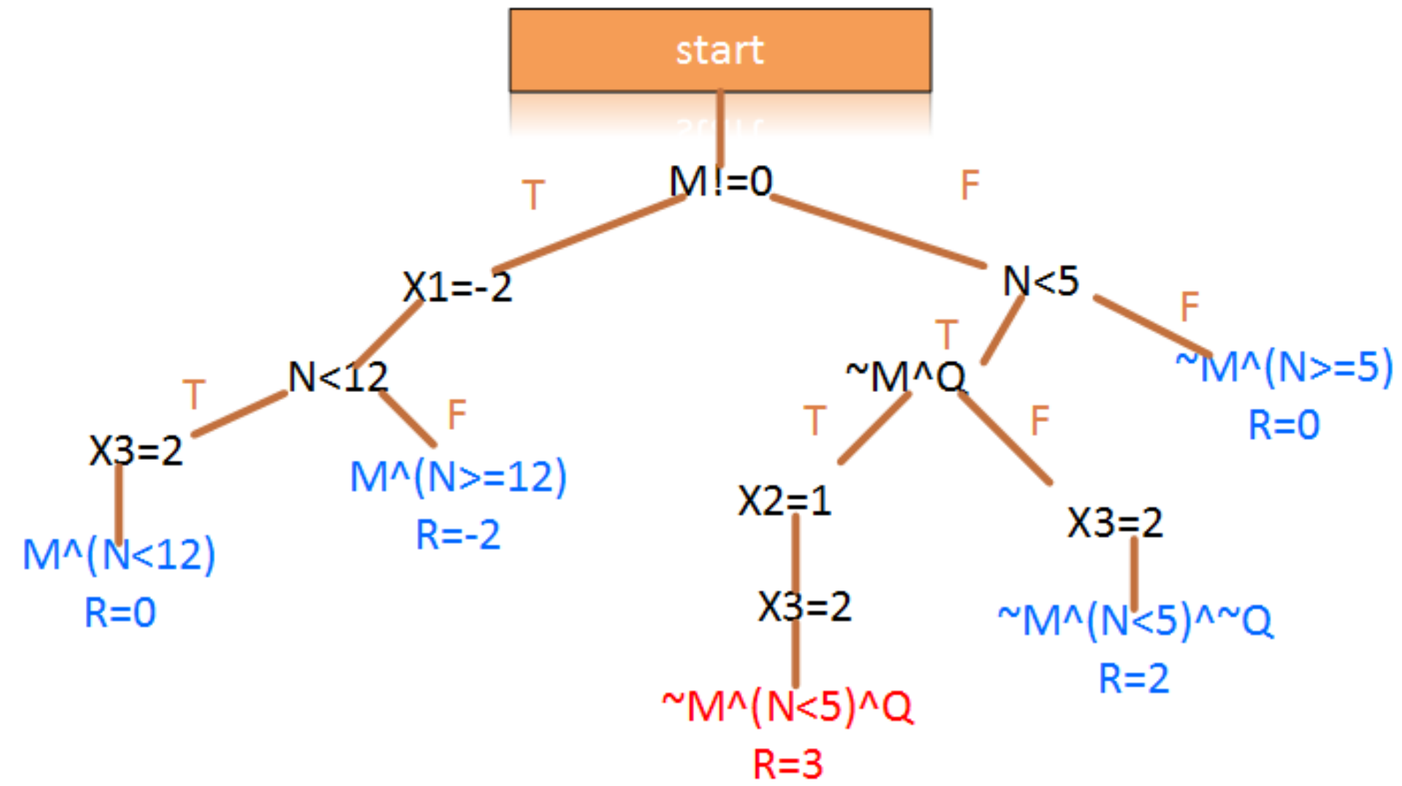
- 1975年就提出的一种程序分析技术
  - 可以系统化探索程序的各条执行路径
  - 近10几年得到巨大的发展
    - 硬件计算能力、更高效的约束求解算法
    - 动态符号执行技术
- 主要应用
  - 缺陷/漏洞检测（SAGE）
  - 测试用例生成（Pex、KLEE、SPF）

符号执行

- 符号执行的工作过程
  - 把程序的输入用符号来表示
  - 在程序中进行符号计算，变量都表示为符号表达式
  - 遇到分支，程序分解为两个路径，每条路径有一个布尔表达式表示的Path Condition（路径约束）
  - 这样不断分解路径，形成程序的符号执行树，代表程序的各条执行路径

符号执行

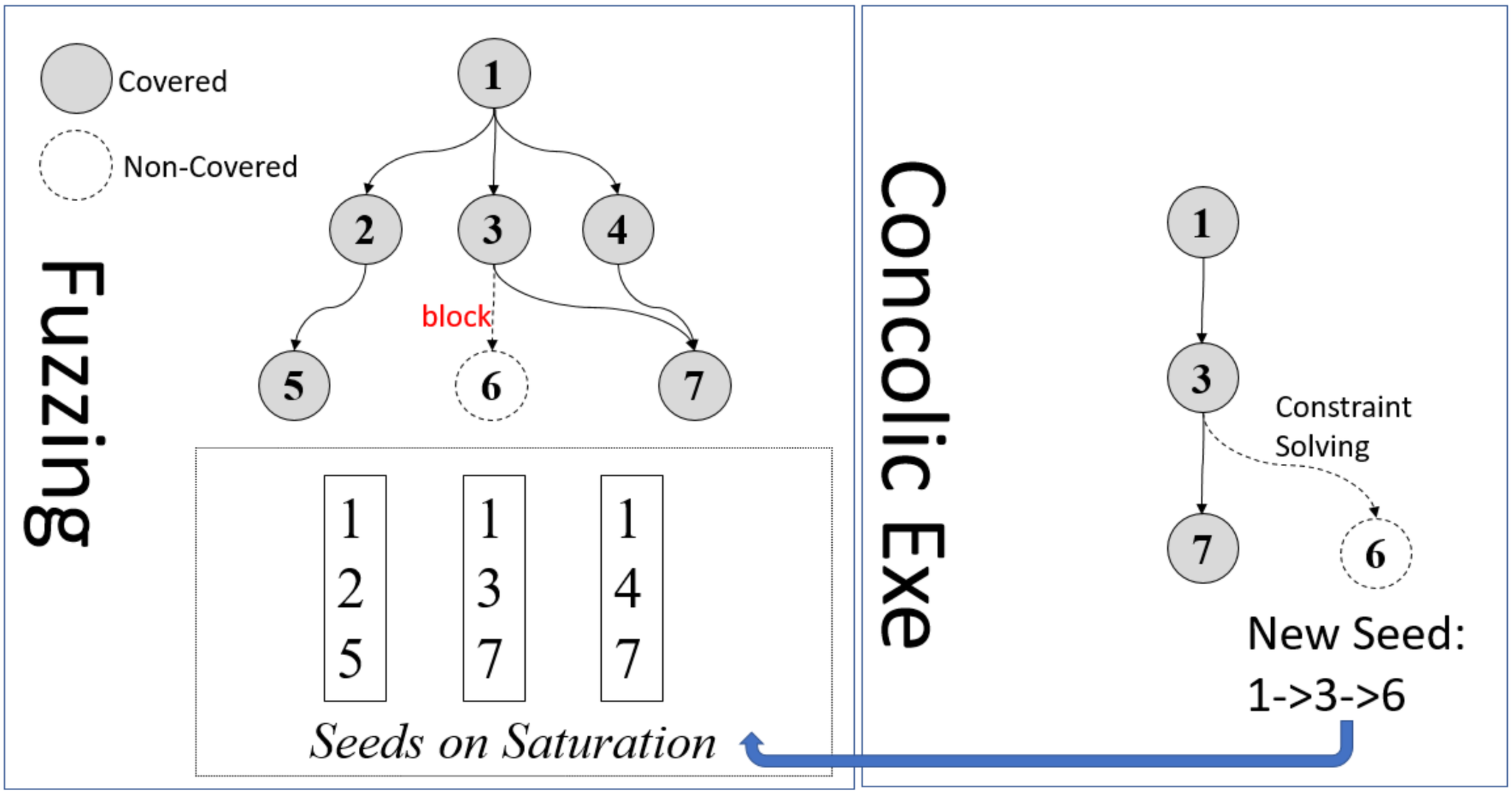
- 符号执行的工作过程
  - 设输入为X
  - 代码语句 = 方程
  - 符号执行 = 方程组生成
  - 约束求解 = 解方程



动态符号执行

- 优势
  - 定向遍历
  - 可生成实际的输入
- 不足
  - 路径爆炸
  - 性能损耗太大，特别是在高效的Fuzzing面前，无对比无伤害

Hybrid Fuzzing



Hybrid Fuzzing

- 模糊测试 + 符号执行
  - 理论上的优势互补
  - Fuzzing难以遍历解空间很小的状态，Concolic可以
  - Concolic很慢且路径爆炸，大部分的遍历活交给fuzzing

- 如何调度？如何分发遍历任务？

Hybrid Fuzzing

- 模糊测试 + 符号执行
  - 理论上的优势互补
  - Fuzzing难以遍历解空间很小的状态，Concolic可以
  - Concolic很慢且路径爆炸，大部分的遍历活交给fuzzing

有事没事多和Boss沟通；BOSS的嗅觉是很敏锐的

  - 如何调度？如何分发遍历任务？

Hybrid Fuzzing

- Towards Optimal Concolic Testing (2018)

- 需求分发策略
  - Hybrid Concolic Testing (2007)；Driller (2016)

Hybrid Fuzzing

- Towards Optimal Concolic Testing (2018)
  - 针对每个path，分析fuzzing和符号执行的代价
  - 总是选择最经济的那个
- 问题
  - 理论方法
    - 难以量化fuzzing和符号执行的代价
      - 符号执行的代价和目标程序、符号执行器、约束求解器等都有关
    - 量化方法依然重量级

Hybrid Fuzzing

- 需求分发策略
  - Hybrid Concolic Testing (2007)；Driller (2016)
  - 首先启动fuzz，监控fuzzing的状态
    - 当不能继续发现代码覆盖率时，fuzzing遇到困难了
    - 启动符号执行辅助

Hybrid Fuzzing

- 需求分发策略
  - Stuck是Fuzzzer的状态，并不能反应程序被测状态
    - Where is the stuck point
    - Fuzzer没有stuck, 程序的某些路径是否已经stuck
  - Stuck状态是个暂时的
    - 如何量化fuzz stuck
- 即便是找到stuck point，代价的比较依然存在
  - 高效率的轻量级尝试　vs. 笨重的精确计算

需求分发策略效果

- 118个程序中有49个程序的测试调用了concolic execution
- 对于纯fuzzing，其stuck 持续时间都很短，80%在100s以内

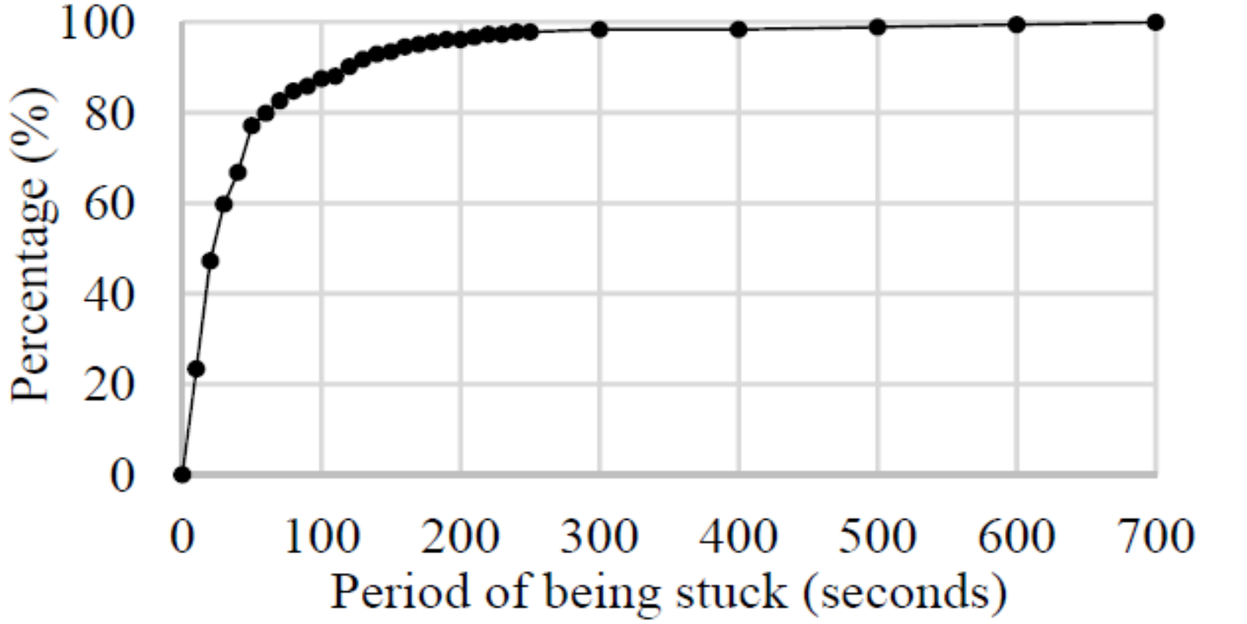
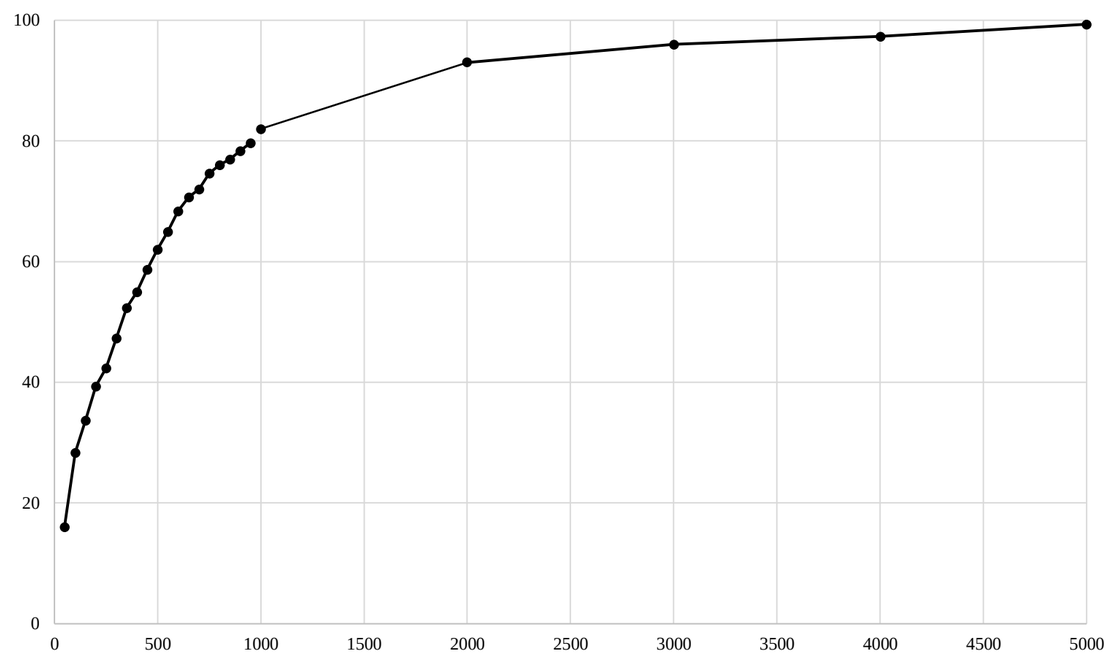


Fig. 1: The distribution of the stuck state durations



需求分发策略效果

- interesting inputs的数量和在有限时间内(12h) concolic execution能够完成的input数量大得多

- 7.1% (1709 /23915)

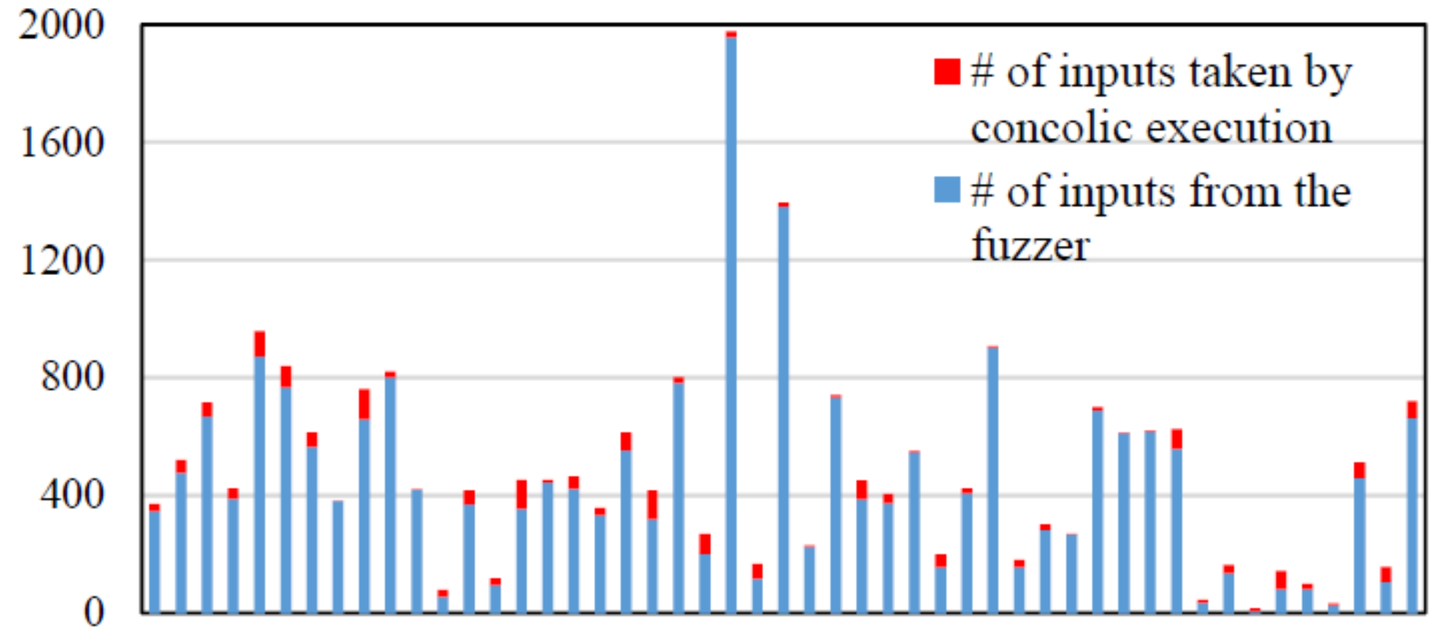


Fig. 2: The number of inputs retained by the fuzzer and the number of inputs taken by concolic execution.

需求分发策略效果

- interesting inputs的数量和在有限时间内(12h) concolic execution能够完成的input数量大得多

- 当前最快的QSYM (引自SAVIOR)

Let's say Tcpdump

- In 24 hours, AFL generates over 20K seeds.
- In 24 hours, QSYM, the fastest concolic executor, can replay around 600 seeds.

混合式漏洞挖掘

- 如何判别程序路径的解空间？
  - 精确的解空间估计依赖于精细化的程序分析
    - 概率符号执行、程序值集分析（VSA）
  - 重量级的方法可能会得不偿失
    - 精细化程序分析所导致的性能损耗得不偿失
  - 轻量级的粗略估计？

DigFuzz

- 一种基于蒙特卡洛方法的概率混合模糊测试

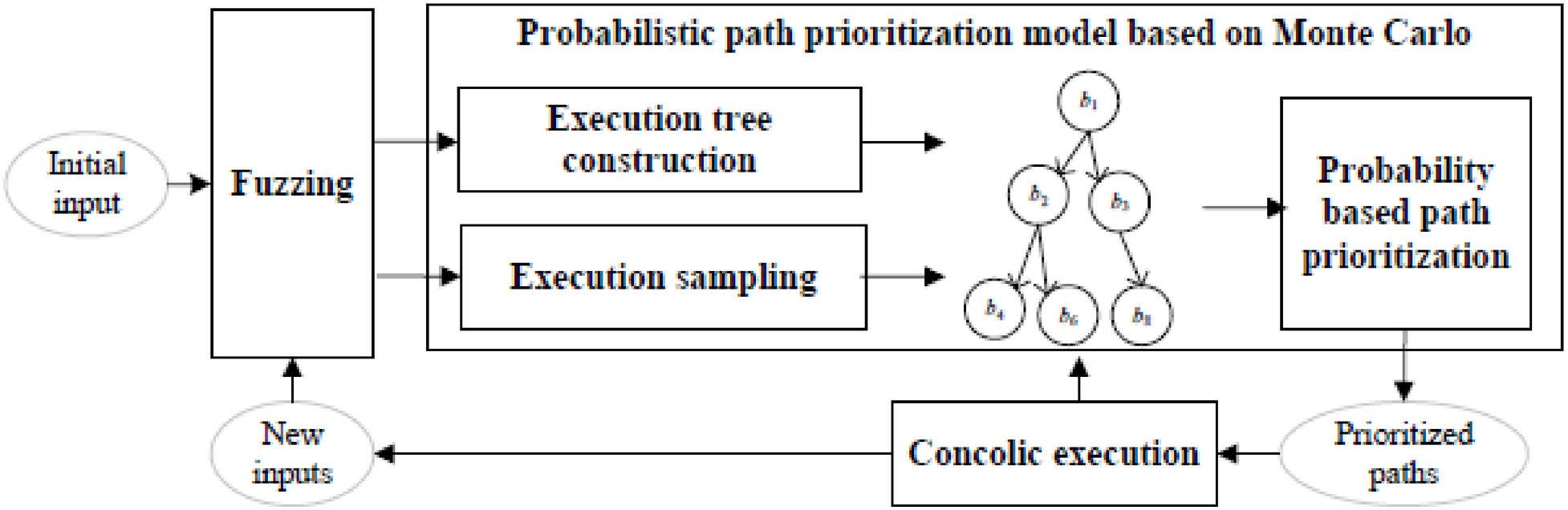


Fig. 3: Overview of DigFuzz

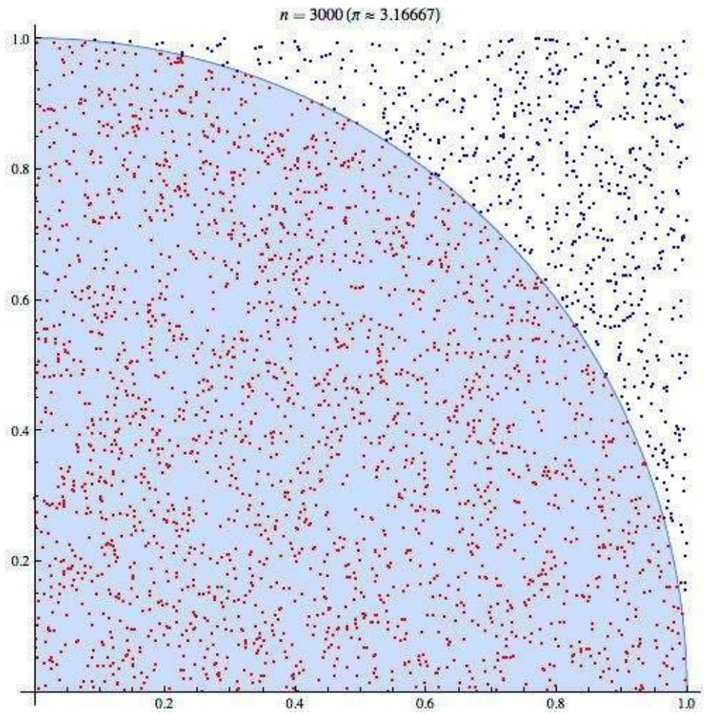
我们的方法：DigFuzz

- 一种基于蒙特卡洛方法的概率混合模糊测试
  - 基于蒙特卡洛方法度量路径的解空间
    - 随机取样命中的概率
  - 基于概率对missed paths排序
    - 在统一的概率估计模型下排序，而不是去直接比较fuzzing和concolic execution的代价
- 每次concolic execution 都选择概率最低的路径

我们的方法：DigFuzz

- 蒙特卡洛方法
  - 通过某种“试验”的方法，得到这种事件出现的频率，或者这个随机变数的平均值，并用它们作为问题的解。

蒙



DigFuzz

- 基于蒙特卡洛方法的概率估计
  - 将Fuzz随机测试抽象成为程序状态的随机取样
    - 解空间的值 -> 基于蒙特卡洛方法的估计
  - 监视Fuzz所产生的代码覆盖信息，估计每条路径的概率

DigFuzz

- 路径的概率估计
  - 直接估计路径概率需要路径的覆盖信息
  - 将路径概率抽象为马尔科夫过程
    - 先计算branch的概率

$$P(br_i) = \begin{cases} \frac{cov(br_i)}{cov(br_i)+cov(br_j)}, & cov(br_i) \neq 0 \\ \frac{3}{cov(br_j)}, & cov(br_i) = 0 \end{cases} \quad (1)$$

    - 依据路径覆盖的branch，估计整个路径的概率

$$P(path_j) = \prod \{P(br_i) \,|br_i \in path_j\} \quad (2)$$

Evaluation

- CQE样本集上的覆盖率
  - 样本特色
- Crash数量

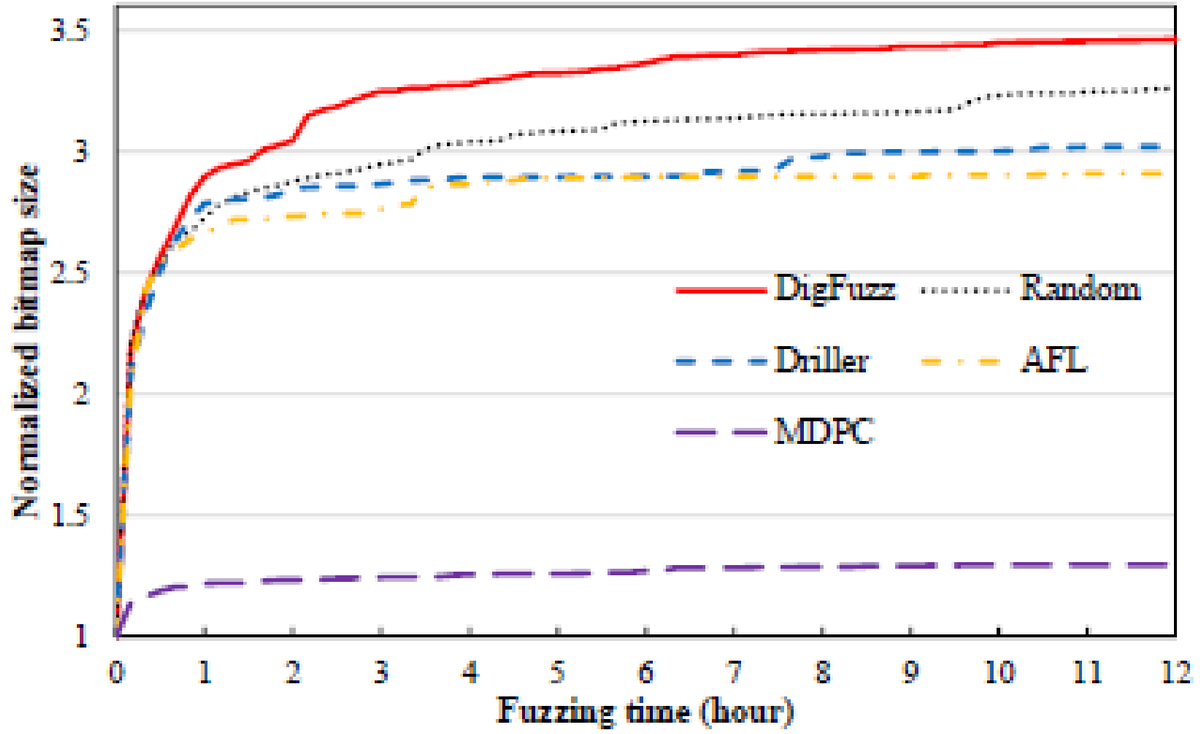


Fig. 6: Normalized bitmap size on CQE dataset

TABLE II: Number of discovered vulnerabilities

	$=3$	≥ 2	≥ 1
DigFuzz	73	77	81
Random	68	73	77
Driller	67	71	75
AFL	68	70	73
MDPC	29	29	31

Evaluation

- CQE样本集上的覆盖率
  - 样本特色
- Crash数量

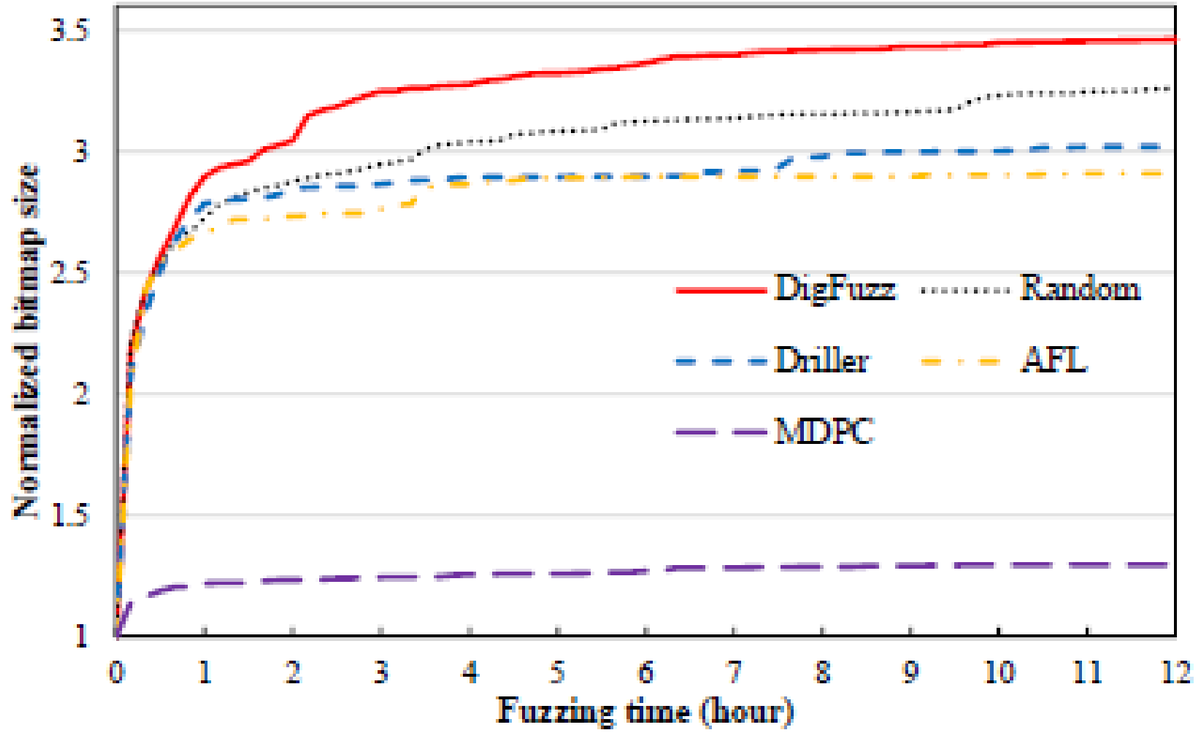


Fig. 6: Normalized bitmap size on CQE dataset

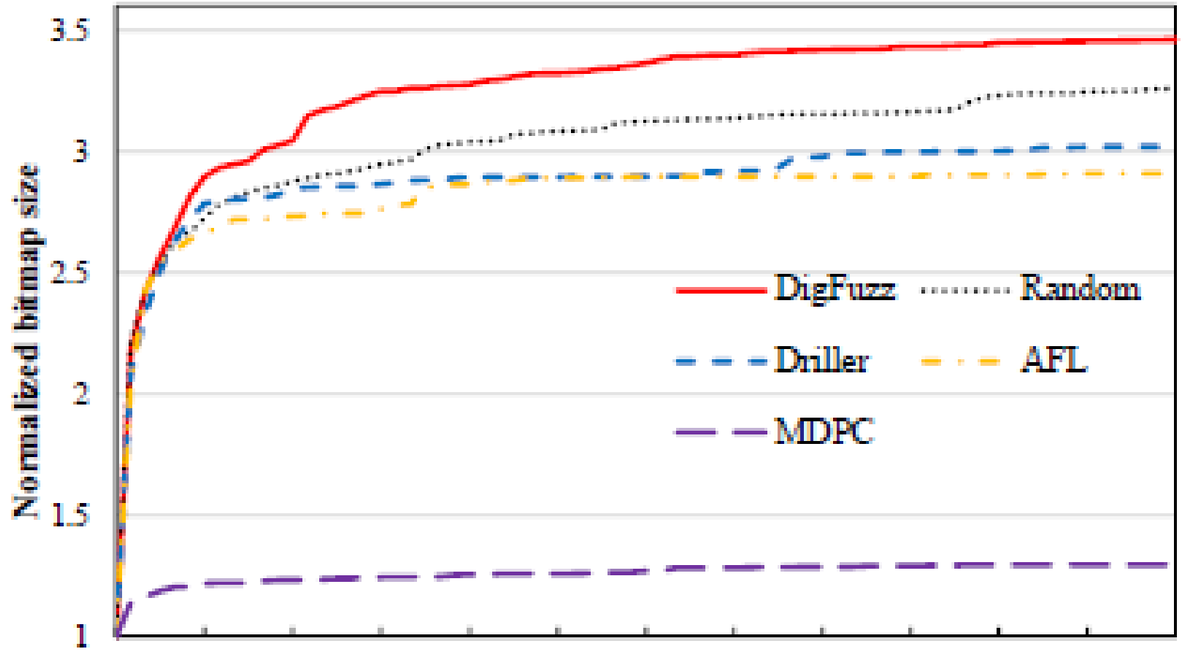
TABLE II: Number of discovered vulnerabilities

	$=3$	≥ 2	≥ 1
DigFuzz	73	77	81
Random	68	73	77
Driller	67	71	75
AFL	68	70	73
MDPC	29	29	31

the motivation is very well done (in some aspects better than the own evaluation)

Evaluation

- CQE样本集上的覆盖率
  - 样本特色
- Crash数量



有事没事多和Boss沟通；过程有时候更重要

TABLE II: Number of discovered vulnerabilities

	= 3	≥ 2	≥ 1
DigFuzz	73	77	81
Random	68	73	77
Driller	67	71	75
AFL	68	70	73
MDPC	29	29	31

the motivation is very well done (in some aspects better than the own evaluation)

Evaluation

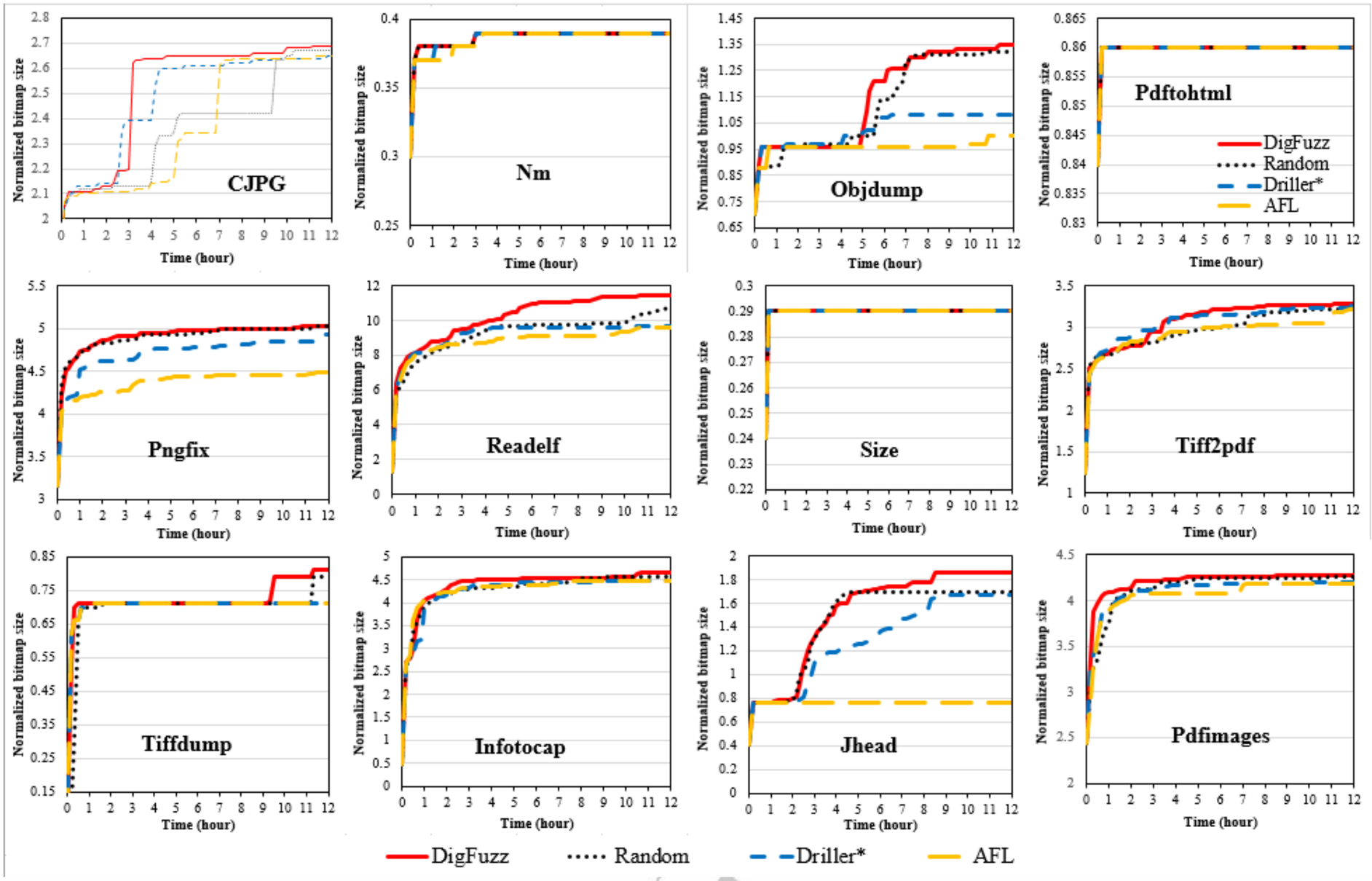
- Concolic Execution 的贡献
- 代码覆盖率的增长
  - 纯fuzz: 2.91倍
  - Driller : 3.02
  - Random: 3.25
  - DigFuzz: 3.46

TABLE III: Performance of concolic execution

	Ink.	CE	Aid.	Imp.	Der.	Vul.
DigFuzz	64	1251	37	719	9,228	12
	64	668	39	551	7,549	11
	63	1110	41	646	6,941	9
Random	68	1519	32	417	5,463	8
	65	1235	23	538	5,297	6
	64	1759	21	504	6,806	4
Driller	48	1551	13	51	1,679	5
	49	1709	12	153	2,375	4
	51	877	13	95	1,905	4

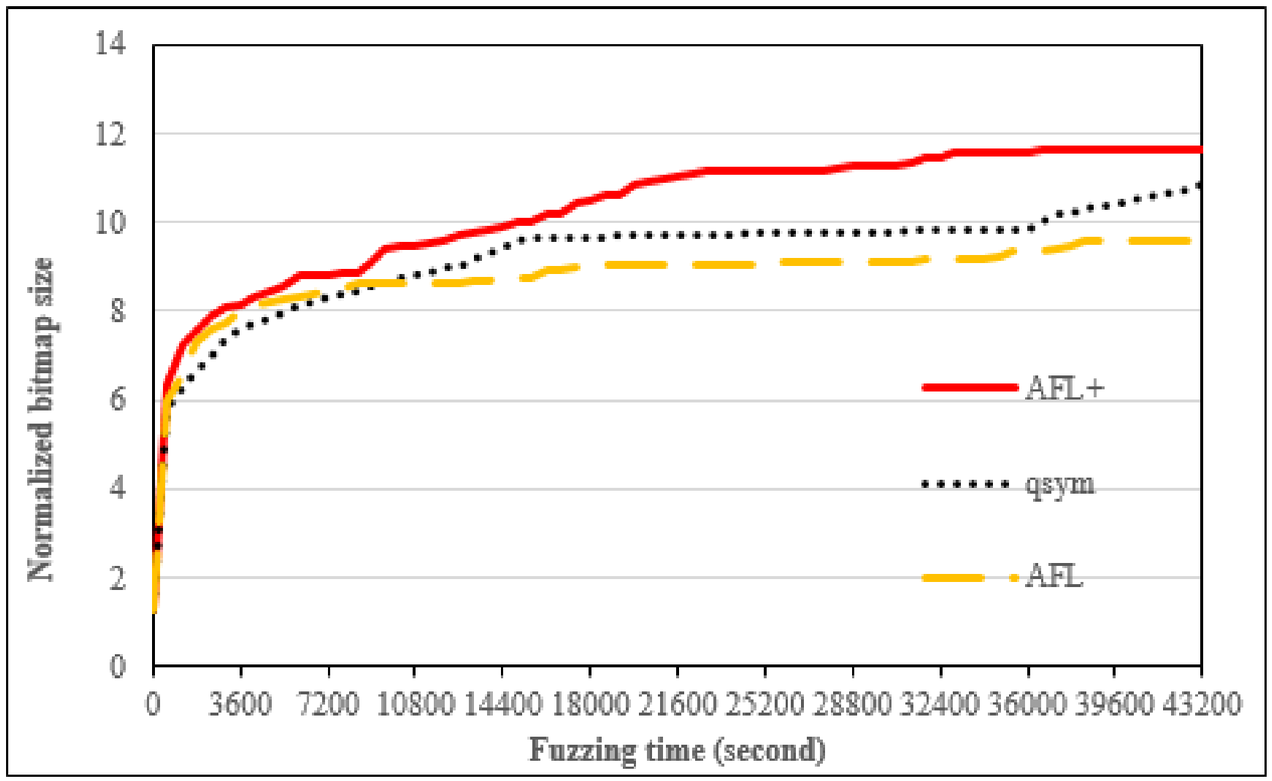
Evaluation

- 基于QSYM在Linux 程序上的结果



后续工作的发现

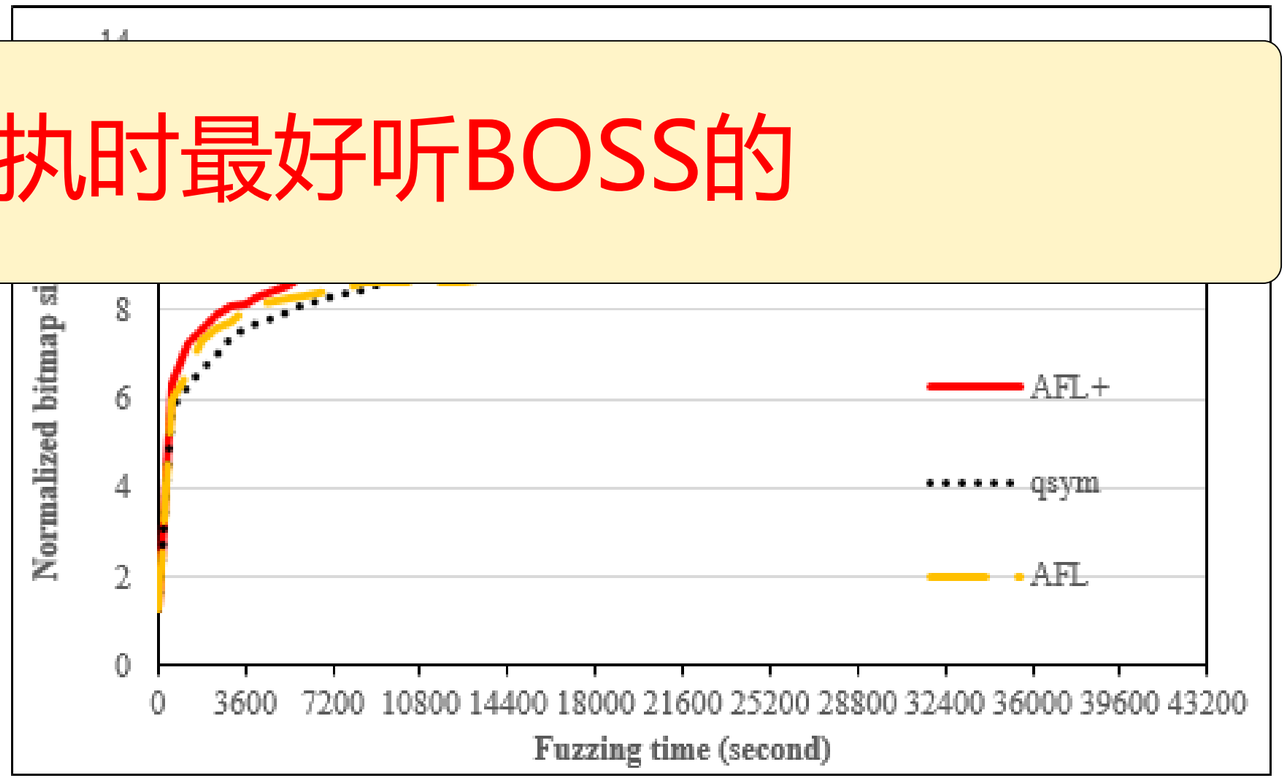
- 符号执行的贡献依然很低，性能亟待提升
- 同等资源限定下的收益不如Fuzzing
  - AFL+：将符号执行器所占的CPU资源全部分配给AFL
  - Qsym: hybrid fuzzing
    - CPU资源等价
    - 内存资源更多



后续工作的发现

- 符号执行的贡献依然很低，性能亟待提升
- 同等资源限定下的收益不如Fuzzing
  - AFL+：将符号执行器所占的CPU资源全部分配给AFL
  - Qsym: hybrid fuzzing
  - 内存资源更多

犹豫不决、起争执时最好听BOSS的




Q&A