2019-2020学年秋季学期

漏洞利用与攻防实践

Exploiting Software Vulnerability-Techniques and Practice

报告：宾浩宇 黄振洋

实验：丁子恒 刘宸睿

漏洞利用与攻防实践

Exploiting Software Vulnerability-Techniques and Practice

[第六次课] 模糊测试实践

概 要

- AFL回顾

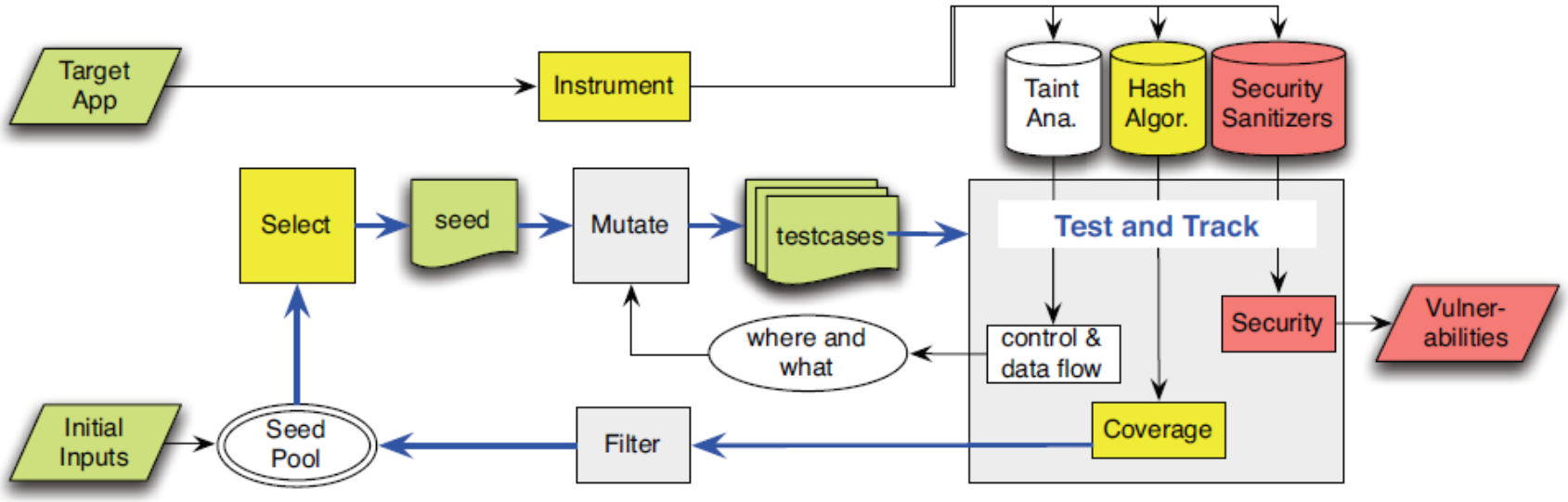
- AFL改进：

1. AFLFast：《Coverage-based Greybox Fuzzing as Markov Chain》-CCS'16

2. Hawkeye：《Hawkeye: Towards a Desired Directed Grey-box Fuzzer》-CCS'18

- 实验

AFL工作流程



①从源码编译程序时进行插桩，以记录代码覆盖率（Code Coverage）；
②选择一些输入文件，作为初始测试集加入输入队列（queue）；
③将队列中的文件按一定的策略进行“突变”；
④如果经过变异文件更新了覆盖范围，则将其保留添加到队列中;
⑤上述过程会一直循环进行，期间触发了crash的文件会被记录下来。

AFL回顾

AFL策略：

AFL维护了一个队列(queue)，每次从这个队列中取出一个文件seed，对其进行大量变异，并检查运行后是否会引起目标崩溃、发现新路径等结果。

AFL变异的主要类型：

1. bitflip，按位翻转，1变为0，0变为1
2. arithmetic，整数加/减算术运算
3. interest，把一些特殊内容替换到原文件中
4. dictionary，把自动生成或用户提供的token替换/插入到原文件中
5. havoc，中文意思是“大破坏”，此阶段会对原文件进行大量变异
6. splice，中文意思是“绞接”，此阶段会将两个文件拼接起来得到一个新的文件

AFL回顾：如何生成新的测试用例

- 如何从seed pool选择一个seed：
- AFL ：

Algorithm:Coverage-based Greybox Fuzzing

Input: Seed Inputs S
$T_{\lightning}$=∅
$T = S$
if $T = \emptyset$ **then**
    add empty file to T
end if
repeat
    t = choose_next (T)
    p = assign_energy(t)
    for i from 1 to p **do**
        $t' = mutate\_input(t)$
        if t' crashes **then**
            add t' to $T_{\lightning}$
        else if is_interesting(t') **then**
            add t' to T
        end if
    end for
until timeout reached or abort-signal
Output: Crashing Input $T_{\lightning}$

AFL改进

- 优化AFL选择seed策略：

1. AFLFast：《Coverage-based Greybox Fuzzing as Markov Chain》-CCS'16

2. Hawkeye：《Hawkeye: Towards a Desired Directed Grey-box Fuzzer》-CCS'18

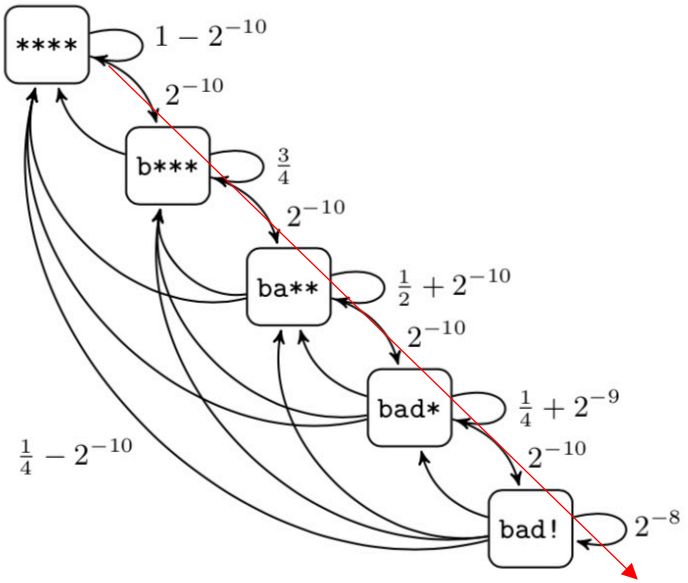
AFL改进：AFLfast

- 改进原因：

✓挑战：大多数模糊操作路径相同，无效操作居多

✓改进方向：AFLfast偏向低频路径，探索更多路径

```
void crashme (char * s){
        if (s[0] == 'b')
            if (s[1] == 'a')
                if (s[2] == 'd')
                    if (s[3] == '!')
                        abort();
}
```



AFL改进：AFLfast

- AFLfast改进方法：

✓AFLfast修改seed输入的选择策略(update_bitmap_score)

✓AFLfast修改每个种子fuzz次数的计算方法(calculate_score)

AFLfast：选择策略

1. 种子t_i之前从队列中选择的次数小的优先级高

2. 执行路径i生存输入能量小的优先级高

AFLfast：能量分配策略

AFLfast将GCF视为马尔科夫链：

- 假设当前种子输入执行路径为i，fuzz之后的路径为j的概率为P_{ij},则称路径i到路径j所要生成的测试用例个数为该种子输入能量$E[X_{ij}] = \frac{1}{P_{ij}}$
- 由于P_{ij}通常是未知的，因此对于能量的分配是经验性的。AFL采用的是一个常数，AFLfast采用了多种新的分配策略。

$$p(i) = E\big(s(i), f(i)\big)$$

- $p(i)$是分配的能量，即seed将被变换的次数
- $s(i)$是种子t_i之前从队列中选择的次数
- $f(i)$是路径I的生成输入能量，$f(i) = E[X_{ij}]$

AFLfast：能量分配策略

- EXPLOIT： $p(i) = \alpha(i)$
- EXPLORE: $p(i) = \frac{\alpha(i)}{\beta}$
- COE: $p(i) = \begin{cases} 0 & , f(i) > u \\ min(\frac{\alpha(i)}{\beta} 2^{s(i)}, M) & , others \end{cases}$
- FAST: $p(i) = min(\frac{\alpha(i)}{\beta} \frac{2^{s(i)}}{f(i)}, M)$
- LINEAR: $p(i) = min(\frac{\alpha(i)}{\beta} \frac{s(i)}{f(i)}, M)$
- QUAD: $p(i) = min(\frac{\alpha(i)}{\beta} \frac{s(i)^2}{f(i)}, M)$

AFL改进： Hawkeye

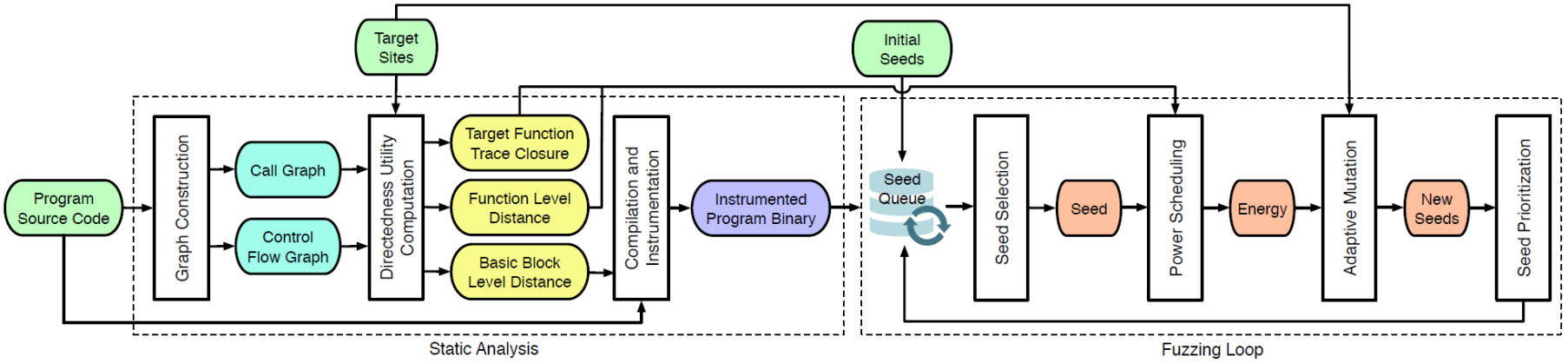
- 改进原因：

- ✓挑战：
  - 大部分的fuzz工具是基于覆盖率的，没有指向性
  - 现有的指向性Fuzz方法(AFLGo)存在偏向于短路径，有些长路径上会存在crash

- ✓改进方向：分配更多能量给长路径

AFL改进：Hawkeye

Hawkeye改进方法：静态分析 + fuzzing loop

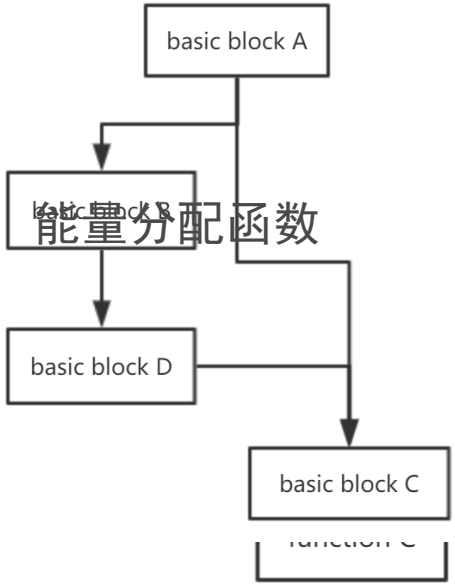


- **静态分析：构建CFG(control flow graph)，CG(call graph)**
  - 基本块级别距离
  - 函数级别距离
  - 目标函数trace的闭包

→ 基本块trace的距离

覆盖函数的相似度

→ 能量分配函数



- **fuzzing loop**

Hawkeye：能量分配策略

- **基本块级别的距离：** $d_b(m_1, m_2)$为基本块m_1到m_2的最短路径

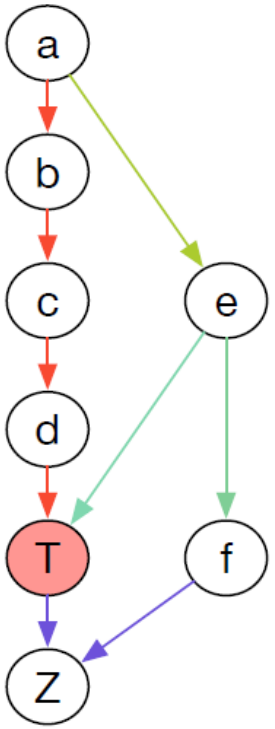
- **基本块路径的距离：** $d_s(s, T_b) = \frac{\Sigma_{m \in \xi_b(s)} d_b(m, T_b)}{|\xi_b(s)|}$

S:种子，
T_b：目标基本块

$$d_s(abcdTZ) = \frac{d_s(a,T) + d_s(b,T) + d_s(c,T) + d_s(d,T) + d_s(T,T)}{5}$$
$$= \frac{2+3+2+1+0}{5}$$
$$= 1.6$$

$$d_s(aeTZ) = 1 \qquad d_s(aefZ) = 1.5$$

AFLGo： $d_s(aeTZ) > d_s(aefZ) > d_s(abcdTZ)$



图：函数的控制流图

Hawkeye：能量分配策略

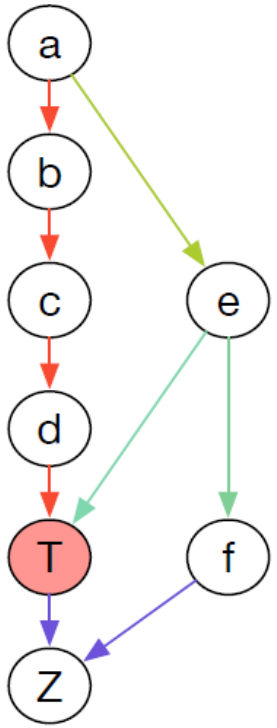
- **函数级别的距离 ：** $d_f(n, T_f) = \begin{cases} \textbf{undefined.} & \text{if } R(n, T_f) = \emptyset \\ [\Sigma_{t_f \in R(n,T_f)} d_f(n, t_f)^{-1}]^{-1} & otherwise \end{cases}$

- **函数路径闭包 ：** 例如 $\xi_f(T_f) = \{a, b, c, d, e, T\}$

- **覆盖函数的相似度：** $c_s(s, T_f) = \frac{\Sigma_{f \in \xi_f(s) \cap \xi_f(T_f)} d_f(f, T_f)^{-1}}{|\xi_f(s) \cup \xi_f(T_f)|}$

- **能量分配函数：** $p(s, T_b) = \tilde{c}_s(s, T_f) \cdot (1 - \tilde{d}_s(s, T_b))$

T_f：目标函数



图：程序的调用图

实验

- **实验目的：**
  - 修改AFL种子队列调度策略，在单位时间内，加快fuzz出crash的速度，并且验证调度优化效果，并分析优化优劣的原因

- **实验策略：**
  - 偏向低频路径，给予低频路径更多机会
  - 偏向能产生更多新路径的种子，给予其更多机会

实验设计

对照一

	AFL	AFLFAST	AFLSLOW
Low Frequency Path		✓	✓
Generate New Path Ability			✓

对照二

Influence of generate new path ability	AFLFAST	AFLSLOW 1.2	AFLSLOW 1.3	AFLSLOW 1.4
Search Schedule			✓	✓
Power Schedule		✓		✓

AFL分析

- 预处理：解析用户输入命令，检查环境变量的设置、输入输出路径、目标文件
- 函数perform_dry_run() 会使用初始的测试用例进行测试，确保目标程序能够正常执行，生成初始化的queue和bitmap。
- 函数 cull_queue() 会对初始队列进行筛选（更新favored entry）。遍历top_rated[]中的queue，然后提取出发现新edge的entry，并标记为favored，使得在下次遍历queue时，这些entry能获得更多执行fuzz的机会。
- 进入while(1)开始fuzz循环
- 进入循环后第一步还是 cull_queue() 对队列进行精简
- 判断queue_cur是否为空，如果是，则表示已经完成对队列的遍历，初始化相关参数，重新开始遍历队列
- fuzz_one() 函数会对queue_cur所对应文件进行fuzz，包括(跳过-calibrate_case-修剪测试用例-能量分配-变异)
- 判断是否结束，更新queue_cur和current_entry
- 当队列中的所有文件都经过变异测试了，则完成一次” cycle done” 。整个队列又会从第一个文件开始，再次继续进行变异

struct queue_entry

```
struct queue_entry {

  u8* fname;                          /* File name for the test case      */
  u32 len;                            /* Input length                     */

  u8  cal_failed,                     /* Calibration failed?              */
      trim_done,                      /* Trimmed?                         */
      passed_det,                     /* Deterministic stages passed?     */
      has_new_cov,                    /* Triggers new coverage?           */
      var_behavior,                   /* Variable behavior?               */
      favored,                        /* Currently favored?               */
      fs_redundant;                   /* Marked as redundant in the fs?   */

  u32 bitmap_size,                    /* Number of bits set in bitmap     */
      fuzz_level,                     /* Number of fuzzing iterations     */
      exec_cksum;                     /* Checksum of the execution trace  */

  u64 exec_us,                        /* Execution time (us)              */
      handicap,                       /* Number of queue cycles behind    */
      depth,                          /* Path depth                       */
      n_fuzz,                         /* Number of fuzz, does not overflow */
      num_new_cov;                    /* New coverage count                */

  u8* trace_mini;                     /* Trace bytes, if kept             */
  u32 tc_ref;                         /* Trace bytes ref count            */

  struct queue_entry *next,           /* Next element, if any             */
                     *next_100;       /* 100 elements ahead               */
};
```

calculate_score()　　能量分配

实验

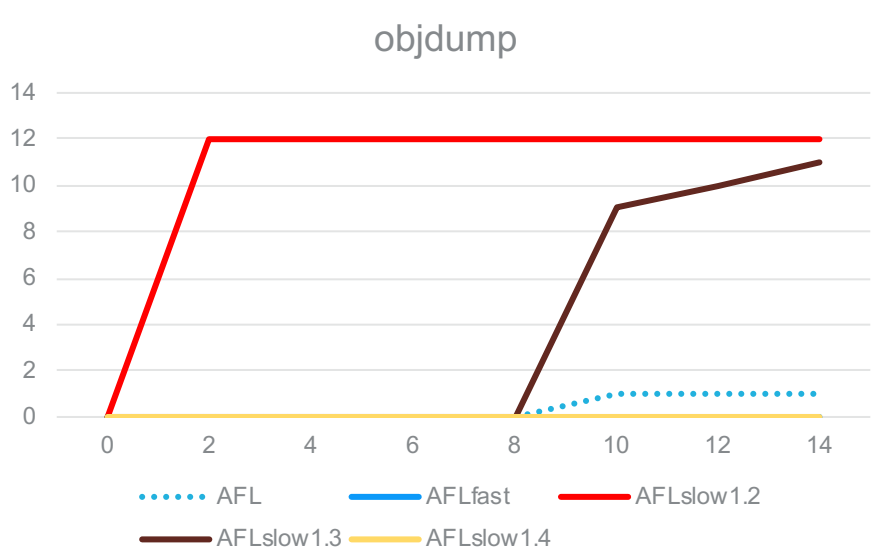
- **实验配置：**
  - 服务器1: 64位机 1核(Xeon®E5-26XX V4) 2G Debian 9.0
  - 服务器2: 64位机 1核(Xeon®E5-26XX V4) 2G Debian 9.0
  - 服务器3: 64位机 1核(Virtual CPU) 1G Debian 9.0
  - 服务器4: 64位机 1核(Xeon®Platinum 8163) 2G ubuntu 16.04.3
  - 服务器5: 64位机 1核(Xeon®E5-2682 V4) 2G ubuntu 16.04.6
  - 服务器6: 64位机 1核(intel®Xeon® CPU) 2.6G Debian 9.0
- **实验对象：**初始种子集为空
  - objdump –x –D –R -S
  - nm-new -C
  - readelf
  - size
  - strings

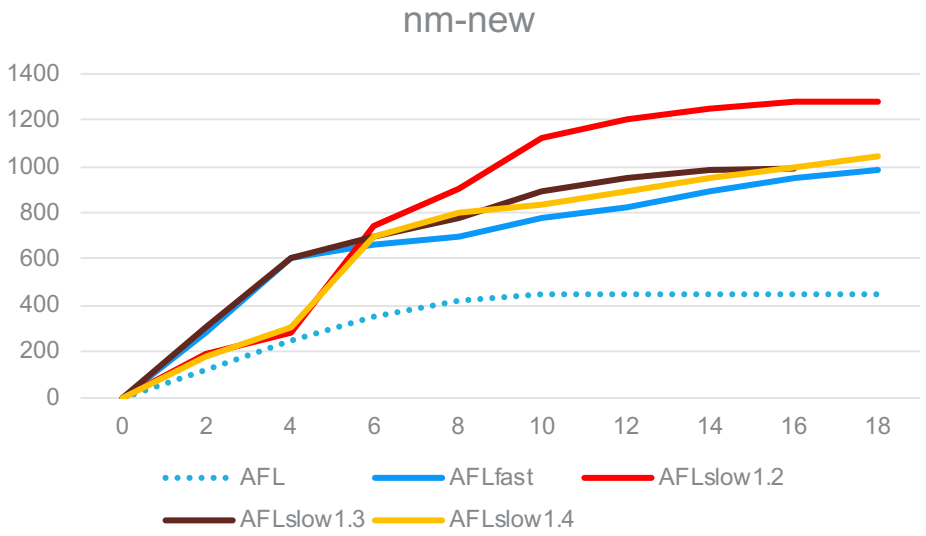
实验结果

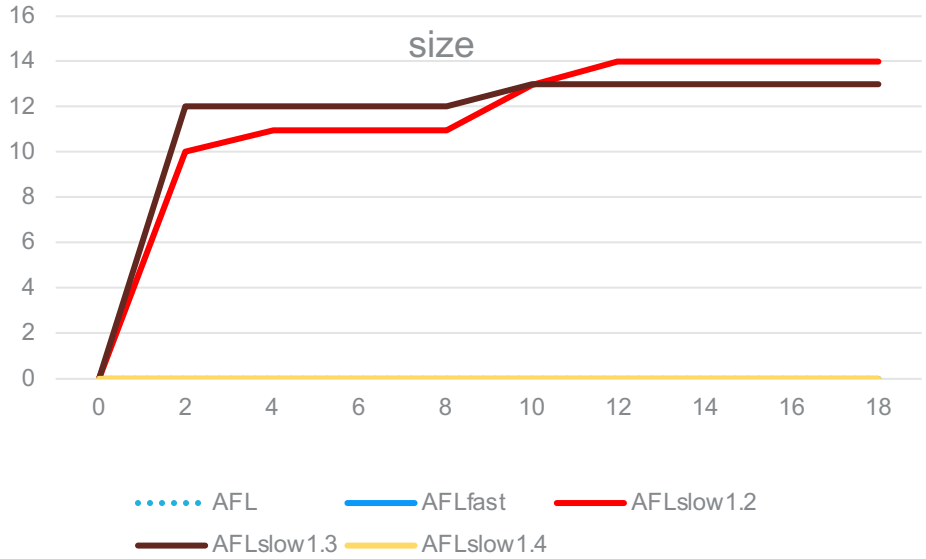
实验运行时间：13-20小时

Crash/hangs	objdump	nm-new	readelf	size	strings
AFL	1/0	448/14	0/0	0/0	0/0
AFLfast	0/0	985/216	0/0	0/0	0/0
AFLslow1.2	12/0	1218/39	0/0	14/0	0/0
AFLslow1.3	11/0	991/32	0/0	13/0	0/0
AFLslow1.4	0/0	1044/31	0/0	0/0	0/0

实验：


objdump
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
AFL
AFLfast
AFLslow1.2
AFLslow1.3
AFLslow1.4


nm-new
1400
1200
1000
800
600
400
200
0
0
2
4
6
8
10
12
14
16
18
AFL
AFLfast
AFLslow1.2
AFLslow1.3
AFLslow1.4


16
14
12
10
8
6
4
2
0
size
0
2
4
6
8
10
12
14
16
18
AFL
AFLfast
AFLslow1.2
AFLslow1.3
AFLslow1.4

实验结果分析

- **低频路径：**

- 选择低频路径能够有效地提高fuzz效率：

表1 ： paths与unique crashes在AFL与AFLfast比较表

Paths/crashes	objdump	nm-new	readelf	size	strings
AFL	1760/1	6706/448	18/0	904/0	79/0
AFLfast	1161/0	10.4k/985	15/0	1795/0	55/0

实验结果分析

更多新路径：

- 已经产生过新路径的种子对fuzz的结果有着积极的影响：

图1 objdump中更多新路径的影响

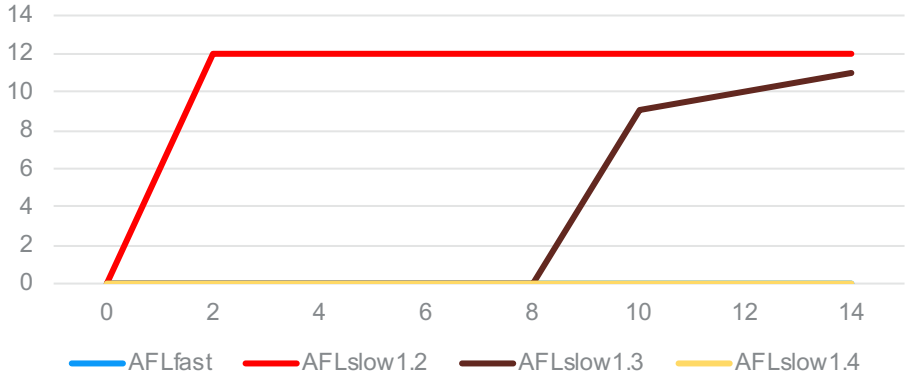
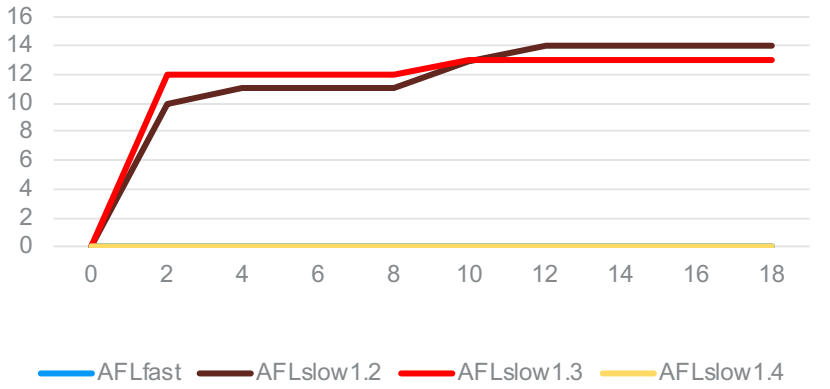


图2 size中更多新路径的影响



- 如图1，图2所示，AFLslow1.2(能量分配),1.3(搜索策略)在objdump、size和nm-new中展现了较好的性能，说明种子产生新路径的数量对挖掘crash的有效性。
- 但是AFLslow1.4却并没有挖掘出任何漏洞。
- 猜测：由于参数设置不够合理，导致已经产生了较多路径但确很难继续产生新路径的种子没有被及时淘汰，因此浪费过多时间。

实验结果分析

重复crash：

- 搜索策略对偏向能产生更多新路径的种子可能会导致更多重复的crash

表2： nm结果uni crashes 与crashes比较

	AFLfast	AFLslow1.2	AFLslow1.3	AFLslow1.4
total crashes	157k	79.4k	649k	242k
unique crashes	985	1218	991	1044

- 由表2可得AFLslow1.3,1.4版本使用了偏向能产生更多新路径的种子搜索策略，这两个版本的总crash数量非常可观，但并没有产生更多的unique crashes。

实验结果分析

误差：

- 由于时间关系，实验的次数只有一次，可能由于运气原因或者服务器性能导致误差产生：

- 在《Coverage-based Greybox Fuzzing as Markov Chain》作者使用AFLfast对objdump进行模糊测试(运行6个小时， 64位机 40 cores (2.6 GHz IntelR XeonR E5- 2600), 64GB内存)获得30 unique crashes，而在我们的实验中AFLfast没有产生任何结果。


Q&A