

# AFL-SYS 介绍与使用文档

---

AFL-SYS（暂定名，maintained by Jifan Xiao & Zixi Zhao）是一个不使用插桩方法的无源码模糊测试框架，使用系统调用日志信息作为新的指导信息，适用性上相比传统方法有更多优势。

## AFL-SYS 介绍与使用文档

- 架构与设计

- 模块与流程

  - AFL-SYS的主要模块

  - AFL-SYS的一次执行流程

- 目录结构

  - afl-sys-demo

  - NoDrop

- 安装配置

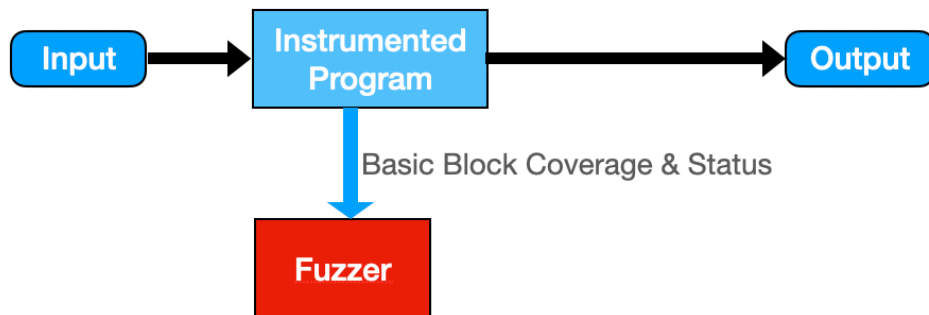
  - 安装NoDrop

  - 安装afl-sys(demo)

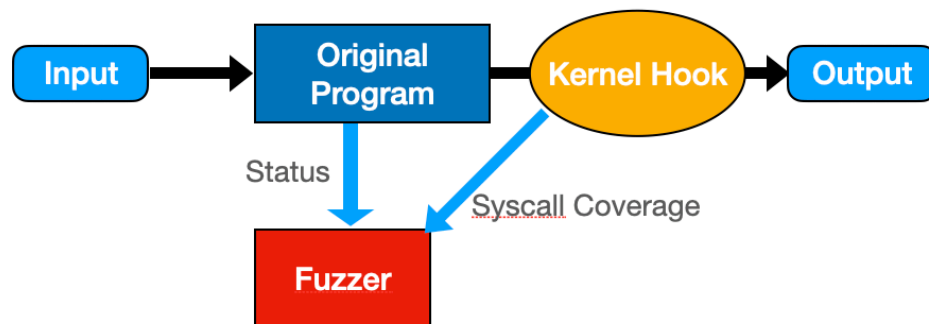
- 使用示例

- 实验效果

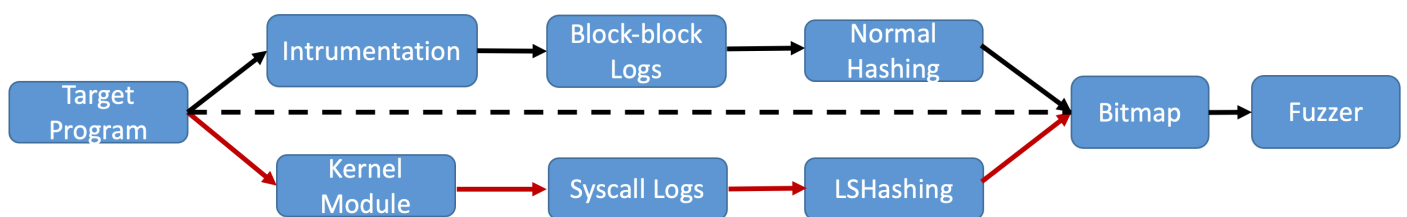
## Previous methods



## New method



- 传统的Fuzzer获取变异指导信息的方法是程序插桩，而在无源码时为了达成插桩，一般会使用静态或动态的二进制重写方法。这种方法一则大多数效率较为低下，二则大多对于被测试二进制文件要求苛刻。
- AFL-SYS使用一个名为**NoDrop**的内核模块获取每次程序执行的系统调用日志信息，将对应的日志发送给主体Fuzzer。主体Fuzzer在AFL的基础上进行了修改，可以分析获取到的日志信息，用以指导后续的文件变异和调度，从而绕开插桩这一获取覆盖率信息的方法。
- 具体来说，在Coverage Pipeline这一部分上的区别如下图所示：（上方为传统方法，下方为AFL-SYS）



- AFL-SYS并没有改动较多AFL原有的变异和计算优先级的算法，仍然保留了原有的**bitmap**，使bitmap反映当前的路径覆盖情况。但是**AFL-SYS**得到的“路径”信息并非传统意义上的路径，而是以一条独特的syscall日志作为一条新路径的反映。因此，实际上在运行过程中面板上所显示的“path”指的是独特的syscall日志的数量，而非传统AFL插桩获得的独特的block edge数量。理论上讲，这两者之间存在正相关关系，不同的执行路径基本上都会产生不同的syscall日志，但实际实验中也发现有特例，无法变异出新的syscall日志“路径”，对于这些特例则暂时没有较好的办法。
- 以伪代码形式表现两种路径的区别：

---

**Algorithm 1** Encoding of AFL

---

**Input:**  $BB_{src} \rightarrow BB_{dst}, prevLoc$ **Output:**  $prevLoc$  - hash value for the next branch

- 1:  $curLoc = random(BB_{dst})$
  - 2:  $bitmap[curLoc \oplus prevLoc] += 1$
  - 3:  $prevLoc = curLoc \gg 1$
- 

---

**Algorithm 2** Encoding of AFL-SYS

---

**Input:**  $S$  - syscall logs of one execution

- 1:  $s_1, s_2, \dots, s_k \leftarrow getNgrams(S)$
  - 2: **for all**  $s_i$  **do**
  - 3:    $hash \leftarrow LSHash(s_i)$
  - 4:    $bitmap[hash] += 1$
  - 5: **end for**
- 

## 模块与流程

本部分首先介绍AFL-SYS中重要的模块，并通过一次的执行流程的分析讲解各个模块之间的协作。

### AFL-SYS的主要模块

- **NoDrop**: 一个内核模块，基于sysdig的内核进行了修改，修复了高压下丢包的问题，并完成了针对Fuzzing频繁重新启动的优化、与Fuzzing对接的API设计等。
- **Forkserver**: 调度各个模块的核心，也兼具每次fork新进程并监控的作用。
- **Fuzzer**: Fuzzer主体程序，负责接收参数、初始化、检查各项内容，然后与Forkserver沟通，发送开辟新进程的命令，变异输入文件，接收覆盖率信息并计算输入文件池中的优先级，以及打印信息、处理各项信号等任务。
- **LSHashing**: 一个Python脚本，会在初始化时与Forkserver建立联系。如有必要，会提供LocalSensitiveHashing的计算功能。
- **TestPaths**: 一组Python脚本套件，读取保留下的输入文件并转化为传统AFL中的bitmap和path信息，用于实验对比。
- **FunctionHook**: Hook一些重要的库函数并获取其参数等作为补充的日志信息，可以通过编译选项启用。目前版本中效果不明显，因此默认不启用。

### AFL-SYS的一次执行流程

- 首先，装载**NoDrop**到内核中。
- 按照与AFL类似的格式接收Fuzzing指令，**Fuzzer主体程序**初始化，建立**Forkserver**及其通信管道，Forkserver再启动建立**LSHashing**，以及与NoDrop建立联系。
- Fuzzer主体程序执行多次calibrate\_case()函数，对初始种子池做遍历，获取初始路径和数据。

- Fuzzer主体程序按照AFL的算法开始变异种子文件，并传送给run\_target()函数，该函数会给Forkserver发送信息，让其fork被测程序后执行对应输入并捕获syscall log信息，更新到bitmap。
- 每次，执行完毕后Fuzzer主体程序会检查bitmap，并更新“路径”等信息，然后按照bitmap指导后续变异出的输入文件优先级。如果有开启input file dump，则会在run\_target()函数中每次执行前先将输入文件拷贝到对应的文件夹。
- 等待用户键盘输入停止指令，然后信号处理函数设置快速结束标志，各个模块迅速终止工作，导出相关信息。
- 如果需要获得传统Fuzzer格式下的度量信息，且之前开启了input file dump，则可以在导出了输入文件的文件夹下执行**TestPaths**的脚本，获得对应的bitmap或路径信息。

## 目录结构

### afl-sys-demo

```

1  | .
2  | |─ Makefile                // make配置文件
3  | |─ afl-analyze.c          // AFL 原有文件
4  | |─ afl-as.c               // AFL assembler
5  | |─ afl-as.h               // AFL assembler 头文件
6  | |─ afl-cmin                // AFL 输入文件剪裁器
7  | |─ afl-fuzz.c             // 新Fuzzer主体程序
8  | |─ afl-gcc.c              // AFL compiler
9  | |─ afl-gotcpu.c           // 获取cpu状态
10 | |─ afl-plot                // 为bitmap绘图
11 | |─ afl-showmap.c          // 读取输入文件并获取对应（传统）bitmap
12 | |─ afl-sys-showmap.c      // 读取输入文件并获取对应（syscall）
    | bitmap
13 | |─ afl-tmin.c             // AFL 输入文件剪裁器
14 | |─ afl-whatsup            // AFL原有文件
15 | |─ alloc-inl.h            // 内存分配头文件
16 | |─ begin.sh               // 记录实验中的各种执行命令
17 | |─ calculateMD5.py        // 为文件生成MD5
18 | |─ config.h               // 配置文件
19 | |─ debug.h                // debug所需头文件
20 | |─ dictionaries           // AFL原有文件夹，测试样例的词典
21 | |─ docs                   // AFL原有文档
22 | |─ exit.sh                // 退出执行的命令脚本（目前版本已不需要）
23 | |─ forkserver.h           // Forkserver部分
24 | |─ funchook               // FunctionHook部分
25 | |─ allfile-hooking.c

```

```

26 |   └─ hooking.so
27 | └─ hash.h                                // 传统hash函数实现
28 | └─ include                              // 与NoDrop模块同步一些信息的头文件
29 |   └─ common.h
30 |   └─ events.h
31 |   └─ export.h
32 |   └─ ioctl.h
33 | └─ libdislocator                        // 省略，原版AFL所用的库
34 | └─ libtokencap                          // 同上
35 | └─ llvm_mode                           // 同上
36 | └─ logs                                // debug模式下存储各个log信息的文件夹
37 |   └─ cur_log.txt
38 |   └─ cur_tuple
39 |   └─ cur_tuple1
40 |   └─ cur_tuple2
41 |   └─ logging.txt
42 |   └─ tupleComp.txt
43 | └─ lsh.py                              // LSHashing模块
44 | └─ qemu_mode                           // 原版AFL-QEMU所需的QEMU部分
45 |   └─ build_qemu_support.sh
46 |   └─ patches
47 |     └─ afl-qemu-cpu-inl.h
48 |     └─ configure.diff
49 |     └─ cpu-exec.diff
50 |     └─ elfload.diff
51 |     └─ memfd.diff
52 |     └─ syscall.diff
53 | └─ test-instr.c                         // AFL原版文件，用于测试是否有插桩
54 | └─ test_paths.py                       // TestPaths套件之一
55 | └─ test_readtuple
56 | └─ test_readtuple.c
57 | └─ test_readtuple.py                   // TestPaths套件之一
58 | └─ testcases                           // 原版AFL中自带的测试样例，省略内部结构
59 | └─ types.h                             // 一些类型的规定头文件
60 |
61 | 55 directories, 194 files

```

## NoDrop

```

1 | .
2 | └─ CMakeLists.txt
3 | └─ README.md
4 | └─ benchmark                          // 用于检测NoDrop本身性能的benchmark
5 |   └─ apache2

```

```

6 | | | |— apache2_install.sh
7 | | | |— apr-1.7.0.tar.bz2
8 | | | |— apr-util-1.6.1.tar.bz2
9 | | | |— http-test-files-1.tar.xz
10 | | | |— httpd-2.4.48.tar.bz2
11 | | |— nginx
12 | | | |— http-test-files-1.tar.xz
13 | | | |— nginx-1.21.1.tar.gz
14 | | | |— nginx_install.sh
15 | | |— redis
16 | | | |— redis-6.0.9.tar.gz
17 | | | |— redis_install.sh
18 | | |— test_7z.py
19 | | |— test_nginx.py
20 | | |— test_openssl.py
21 | | |— test_postmark.py
22 | | |— test_redis.py
23 |— include // 导出的头文件，用于给外部程序引用
24 | |— common.h
25 | |— events.h
26 | |— export.h // 与events_table.c等文件同步
27 | |— ioctl.h
28 |— kmodule // 核心代码部分
29 | |— CMakeLists.txt
30 | |— Makefile.in
31 | |— elf.c
32 | |— events.c // 处理一个syscall event的hook
33 | |— fillers.c
34 | |— fillers.h
35 | |— fillers_table.c
36 | |— flags.h
37 | |— loader.c
38 | |— nod_main.c
39 | |— nodrop.h // 控制各项参数的头文件
40 | |— privil.c
41 | |— proc.c // 控制与用户态进程交互的API
42 | |— procinfo.c
43 | |— procinfo.h
44 | |— syscall.h // syscall 头文件
45 | |— syscall_table.c
46 | |— tables
47 | | |— dynamic_params_table.c
48 | | |— events_table.c // 记录syscall信息的表格，决定log格式
49 | | |— flags_table.c
50 | |— trace.c
51 |— monitor
52 | |— CMakeLists.txt

```

```

53 | | | mmheap
54 | | | | mmheap.c
55 | | | | mmheap.h
56 | | | musl.specs
57 | | | script_x86-64.ld
58 | | | src
59 | | | | dynlink.h
60 | | | | main.c
61 | | | | pkeys.h
62 | | | | startup.c
63 | | musl // 外部库依赖，省略内部结构
64 | | scripts
65 | | | CMakeLists.txt
66 | | | StressTesting
67 | | | | 1.txt
68 | | | | CMakeLists.txt
69 | | | | attack.c
70 | | | | attack.sh
71 | | | | stress.c
72 | | | | stress.sh
73 | | | ctrl // 一个用于测试的接口程序
74 | | | | CMakeLists.txt
75 | | | | nodrop-ctl.c // 该程序可以获取API中的信息
76 | | | getmusl.sh
77 | | | mkfig
78 | | | | CMakeLists.txt
79 | | | | draw.cpp
80 | | | | matplotliblibcpp.h
81 | | | | pyconfig.cmake
82 | | | musl-1.2.3.tar.gz
83 | | | tests
84 | | | | CMakeLists.txt
85 | | | | multithread.c
86
87 70 directories, 514 files

```

## 安装配置

目前已知NoDrop模块在Linux 4.15和5.4版本的内核工作良好，其他版本尚未测试。建议使用4.15版本内核的系统。

## 安装NoDrop

- 找到仓库中NoDrop的部分，进入其目录
- 运行目录下的 scripts/getmusl.sh 脚本，以获得依赖musl
- 创建一个build文件夹，进入其中运行：

```
1 cmake ..
2 make load
```

- 如果遇到提示需要gcc版本号大于8，安装gcc-8并在执行命令前添加

```
1 export CC=gcc-8
```

- （可选）如果需要测试NoDrop的工作情况，可以在build文件夹中运行

```
1 make scripts
2 ./scripts/ctrl/ctrl clean
3 ./scripts/ctrl/ctrl fetch
```

clean与fetch分别是ctrl脚本的两个功能，应该能够分别看到"Success"的反馈和被监控进程的syscall log（如果没有运行被监控进程则应该反馈为空）

- （高级）进入kmodule文件夹中，可以修改修改nodrop.h中对应的NoDrop参数。比如比较重要的被监控进程名，对应于NOD\_TASK宏的定义，默认的被监控进程名为toTest。另外，MAX\_EVENTS宏定义了单次执行最多接收的syscall event数量，可以通过修改该宏控制此参数。注意，内核模块无法完成热修改，修改此文件后，注意清除build文件夹下缓存并重新进行编译和装载。

## 安装afl-sys(demo)

- 找到仓库中afl-sys-demo的部分，进入其目录
- 在该目录下执行make即可
- （可选）如果需要观察debug信息，则通过编译选项开启debug模式：

```
1 make debug=1
```

此时，对应的debug信息文件应存在于afl-sys-demo/logs/logging.txt

- （可选）如果需要开启LocalSensitiveHashing替代原有的普通Hash函数（注意，当前版本此选项会带来比较多的时间开销），也有对应的编译选项：

```
1 make lsh=1
```

- （可选）如果需要开启Library Hooking补充syscall log信息（注意，经测试，当前版本此选项效果提升并不多），也有对应的编译选项：



```
1 | make lib_hooking=1
```

- 以上几个可选的编译选项间可以组合

## 使用示例

- AFL-SYS的大多数参数用法与AFL相同，但注意要将被测程序的binary名称改为toTest（或根据自己在nodrop.h文件中的修改情况自行调整）
- 比如，如果要执行一个针对于xpdf suite的测试，提前将pdftotext的binary改为toTest，然后命令为：

```
1 | ./afl-fuzz -i ../xpdfTest/inputs -o ../xpdfTest/output ../xpdfTest/toTest @@  
../xpdfTest/out/null
```

- 由于syscall监控部分可能会占用额外的内存，因此 -m 选项所提供的内存空间基础上，AFL-SYS默认会再加100M。不过如果使用了ASAN或要测试非常大的binary，建议还是开启-m none选项以取消内存限制
- 更多的使用指令可以参见begin.sh

## 实验效果

- 实验中，相比SOTA的二进制重写为基础的binary-only fuzzer，AFL-SYS在速度和效率上的表现有好有坏。一个较为理想的benchmark：xpdf suite上，AFL-SYS（右）与源码编译插桩的AFL（左）的速度与效率对比如图：

american fuzzy lop 2.52b (pdftotext)			
process timing		overall results	
run time : 2 days, 4 hrs, 2 min, 43 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 4 min, 37 sec		total paths : 1887	
last uniq crash : 0 days, 0 hrs, 28 min, 56 sec		uniq crashes : 85	
last uniq hang : none seen yet		uniq hangs : 0	
cycle progress		map coverage	
now processing : 1705 (90.36%)		map density : 2.77% / 4.76%	
paths timed out : 0 (0.00%)		count coverage : 3.98 bits/tuple	
stage progress		findings in depth	
now trying : arith 8/8		favored paths : 236 (12.51%)	
stage execs : 26.0k/189k (13.73%)		new edges on : 432 (22.89%)	
total execs : 53.0M		total crashes : 1738 (85 unique)	
exec speed : 191.5/sec		total tmouts : 1003 (97 unique)	
fuzzing strategy yields		path geometry	
bit flips : 461/3.69M, 81/3.69M, 80/3.69M		levels : 5	
byte flips : 0/461k, 3/212k, 5/219k		pending : 1617	
arithmetics : 129/11.6M, 2/1.54M, 0/90.5k		pend fav : 57	
known ints : 13/1.04M, 5/5.77M, 18/9.51M		own finds : 1884	
dictionary : 0/0, 0/0, 98/10.3M		imported : n/a	
havoc : 1074/937k, 0/0		stability : 100.00%	
trim : 0.80%/197k, 54.44%			

[cpu001: 78%]

american fuzzy lop 2.52b (toTest)			
process timing		overall results	
run time : 2 days, 4 hrs, 4 min, 0 sec		cycles done : 0	
last new path : 0 days, 4 hrs, 22 min, 26 sec		total paths : 1172	
last uniq crash : 0 days, 0 hrs, 2 min, 31 sec		uniq crashes : 93	
last uniq hang : none seen yet		uniq hangs : 0	
cycle progress		map coverage	
now processing : 907 (77.39%)		map density : 0.16% / 2.17%	
paths timed out : 0 (0.00%)		count coverage : 1.67 bits/tuple	
stage progress		findings in depth	
now trying : auto extras (over)		favored paths : 482 (41.13%)	
stage execs : 6156/62.1k (9.92%)		new edges on : 701 (59.81%)	
total execs : 41.3M		total crashes : 1827 (93 unique)	
exec speed : 245.5/sec		total tmouts : 74 (10 unique)	
fuzzing strategy yields		path geometry	
bit flips : 205/5.30M, 38/5.30M, 39/5.30M		levels : 4	
byte flips : 0/663k, 0/119k, 2/124k		pending : 785	
arithmetics : 44/6.50M, 0/982k, 0/21.5k		pend fav : 150	
known ints : 1/600k, 4/3.29M, 3/5.48M		own finds : 1169	
dictionary : 0/0, 0/0, 115/6.34M		imported : n/a	
havoc : 811/970k, 0/0		stability : 100.00%	
trim : 12.70%/309k, 82.19%			

[cpu000: 78%]

当然，也存在诸如gzip这种benchmark，在我们的实验中虽然速度较高，但难以发现新的“路径”，即不同的syscall log。

- 但在**non-stripped binary**，**obfuscated binary**等binary形式的适用性上，AFL-SYS是最为广泛的。诸如RetroWrite、StochFuzz等都要求stripped binary才能工作，而ZAFL虽然不需要此前提，但也无法突破obfuscation等技术的限制：

```
(base) xjf@Fuzz:~/zafl$ zafl.sh ../tcpdumpTest/obuild/bin/tcpdump ../tcpdumpTest/obuild/tcpdump.zafl
Zafl: Detected main at: 0x0000000000403d00
Zafl: Transforming input binary /home/xjf/tcpdumpTest/obuild/bin/tcpdump into ../tcpdumpTest/obuild/tcpdump.zafl
Zafl: Issuing command: /home/xjf/zipr/installed/tools/ps_zipr.sh /home/xjf/tcpdumpTest/obuild/bin/tcpdump ../tcpdumpTest/obuild/tcpdump.zafl -c rida -c move_globals -c zax -o move_globals:--elftables-only -o move_globals:--no-use-stars -o zax:--stars -o zax:--enable-floating-instrumentation -o zax:'-e 0x0000000000403d00'
Using Zipr backend.
Detected ELF non-PIE executable.
Performing step rida [dependencies=mandatory] ...Done. Successful.
Performing step pdb_register [dependencies=mandatory] ...Done. Command failed! *****
*****
The pdb_register step is necessary, but failed. Exiting ps_analyze early.
Zafl: error transforming input program
```

但AFL-SYS并不受影响：

The screenshot displays the American Fuzzy Lop (AFL) interface, titled "american fuzzy lop 2.52b (toTest)". The interface is divided into several sections:

- process timing**:
  - run time : 0 days, 0 hrs, 0 min, 33 sec
  - last new path : 0 days, 0 hrs, 0 min, 1 sec
  - last uniq crash : none seen yet
  - last uniq hang : none seen yet
- cycle progress**:
  - now processing : 2 (0.16%)
  - paths timed out : 0 (0.00%)
- stage progress**:
  - now trying : bitflip 2/1
  - stage execs : 1921/14.4k (13.34%)
  - total execs : 27.5k
  - exec speed : 894.5/sec
- fuzzing strategy yields**:
  - bit flips : 25/14.4k, 0/0, 0/0
  - byte flips : 0/0, 0/0, 0/0
  - arithmetics : 0/0, 0/0, 0/0
  - known ints : 0/0, 0/0, 0/0
  - dictionary : 0/0, 0/0, 0/0
  - havoc : 0/0, 0/0
  - trim : 0.00%/883, n/a
- overall results**:
  - cycles done : 0
  - total paths : 1286
  - uniq crashes : 0
  - uniq hangs : 0
- map coverage**:
  - map density : 0.00% / 3.30%
  - count coverage : 1.00 bits/tuple
- findings in depth**:
  - favorable paths : 168 (13.06%)
  - new edges on : 193 (15.01%)
  - total crashes : 0 (0 unique)
  - total tmoouts : 0 (0 unique)
- path geometry**:
  - levels : 2
  - pending : 1286
  - pend fav : 168
  - own finds : 26
  - imported : n/a
  - stability : 100.00%

At the bottom, the CPU usage is shown as "[cpu000: 61%]". The terminal prompt at the bottom left is "[0] 0:./afl-fuzz\*", and the timestamp at the bottom right is "\"Fuzz\" 10:43 04-Dec-22".

- 目前实验和优化仍在继续，后续仍会有代码和实验结果上的更新。