论文分享

CollAFL: Path Sensitive Fuzzing

- 2018 S&P 清华大学网络科学与网络空间研究院 张超副教授

前言

AFL -- Fuzz工具中的利器/始祖级别，由Google开发，现在很多工具以它为标杆，进行改进、二次开发。

而我们最近学习的**Angr**，是为了增强Fuzz（模糊测试）能力而出现的，具体来说，*ShellPhish*团队为增强AFL的能力，以应对CGC自动化比赛，搞了Angr，然后和AFL配合使用，AFL + Angr = **Driller**。

流程图

下面这张图是CollAFL的工作流程图，其实，也可以看做是，一类以覆盖率为导向的Fuzz工具的流程图，AFL的流程图也可以粗略用这张图。

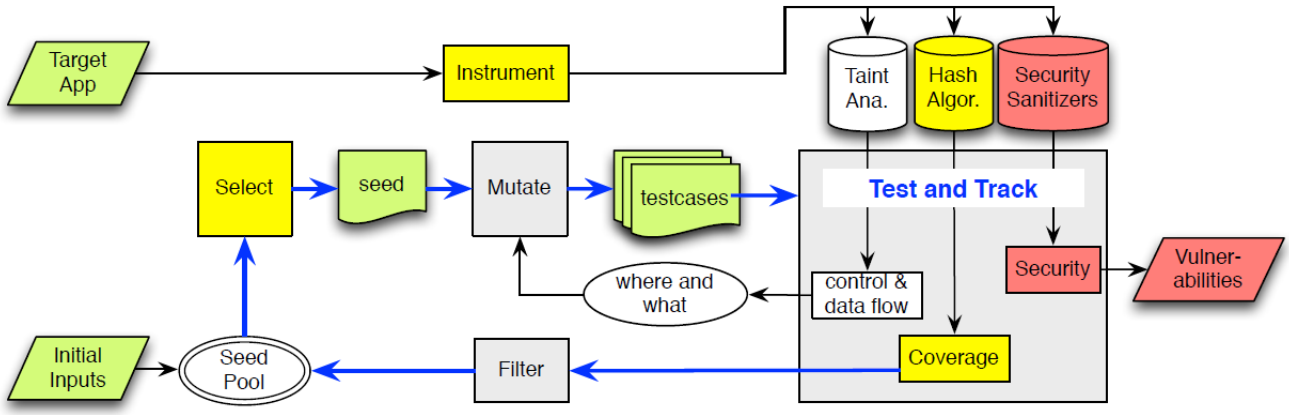


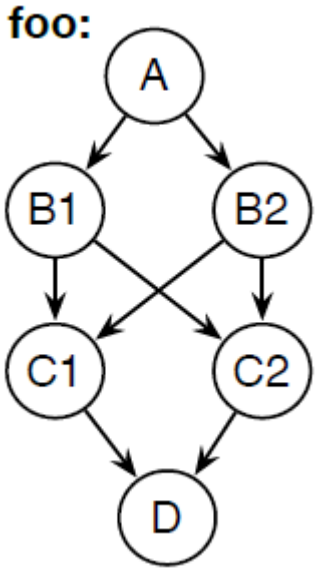
Fig. 1: The general workflow of coverage-guided fuzzing solutions.

存在的问题

- **path collision 路径碰撞导致覆盖率低**

  为什么会碰撞?

AFL要用到一个64KB bitmap来保存**Coverage**边覆盖的信息。怎么保存的呢？它关心的是边的Coverage，这个程序的边到底怎么走过，边是有两个块连接构成，它怎么保存呢？它对每个基本块**block**赋了一个key，上面是**prev**，下面是**cur**，然后它算这个边，它给这个边算了哈希出来，这个哈希就代表这条边。这样的做法，原理很简单。大家看这个哈希算法很简单，会进行碰撞。



碰撞带来的问题

两个不同的边算出来的哈希是一样的，也就是说，路径不同，但在bitmap中标记点相同。碰撞有什么问题呢？最大的问题，它影响这个Fuzzer的判断，因为这个Fuzzer会根据bitmap来判断当前这个种子是不是好种子，判断这个种子是不是走了新的边，如果碰撞了，它是看不出来这是新的边。如果新的边出现了，它的哈希值与前哈希值是一样的，那么Fuzzer认为这个边是测过的边，认为这个种子没用，其实扔掉了，但它实际是好的种子。

More specifically, AFL instruments the target application and assigns random keys to its basic blocks. Given an edge A->B, AFL then computes its hash as follows:

$$cur \oplus (prev \gg 1) \quad (1)$$

where *prev* and *cur* are keys of the basic block A and B respectively. Due to the randomness of keys, two different edges could have a same hash. Moreover, the number of edges is high (i.e., comparable to the bitmap size 64K), the collision ratio would be very high, considering the birthday attack [16].

第二，它甚至可能帮你找到漏洞了。找到漏洞，找到了一个crash，AFL会做一个检测，判断是不是重复的崩溃，如果走的路径和原来一模一样，就认为重复了，要扔掉。碰撞以后导致这样的情况，就可以导致你漏掉漏洞。还有比较严重的是，bitmap碰撞以后导致它提供Coverage信息不准确，其他依赖Coverage信息做决策的，就会做出错误的决策，这就包括前面讲的怎么选种的问题，有策略，选种子的时候是基于Coverage做的判断，Coverage不准的话会导致它做出错误的判断。

解决办法

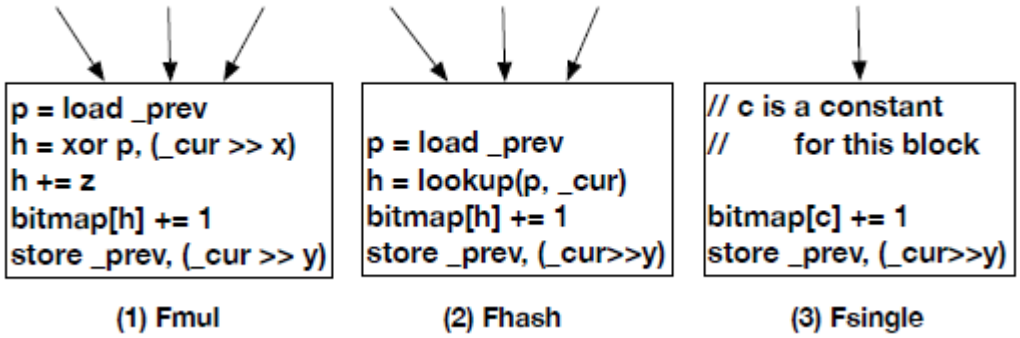
1. 消除碰撞

> AFL用64KB bitmap跟踪边的覆盖率。为了消除碰撞，一个很简单的方式，就是把64kb bitmap增大，把哈希表变大，这样碰撞概率就会降低，但是，把bitmap变大之后，AFL的速度会往下降，比如，64KB变到4MB，执行速度会降60%。所以，这不是最优方案。

改变计算哈希的方法：原先是右移1位，现在做了泛化，设置三个参数x，y，z，`cur >> y`，`prev >> x`，做异或，然后加上一个z。这样变成每条边赋值一个x、y、z来消除碰撞，这样，就需要额外维护几个表存储key值。

$$Fmul(cur, prev) = (cur \gg x) \oplus (prev \gg y) + z$$

但不是所有的边都赋值三个参数，分三种情况，多个前驱且已分配，多条前驱但未分配，单个前驱。第一种，分配三个参数；第二种，找不到合适的三个参数了，就从哈希表中找未使用的hash；第三种，单个前驱，分配固定的常量。



Algorithm 1 The collision mitigation algorithm.

Input: Original program
Output: Instrumented program

1: $(BBS, SingleBBS, MultiBBS, Preds) =$ **GetCFG**()
2: $Keys =$ **AssignUniqRandomKeysToBBs**(BBS)
3: // Fixate algorithms. Preds and Keys are common arguments
4: $(Hashes, Params, Solv, Unsolv) =$ **CalcFmul**$(MultiBBS)$
5: $(HashMap, FreeHashes) =$ **CalcFhash**$(Hashes, Unsolv)$
6: // Instrument program with coverage tracking.
7: **InstrumentFmul**(Solv, Params)
8: **InstrumentFhash**(Unsolv, HashMap)
9: **InstrumentFsingle**(SingleBBS, FreeHashes)

同时，把bitmap适当增大。算法调整后，最麻烦的是查哈希表的内容，我们严格控制哈希表的大小，如果哈希表大的话，runtime查询哈希表会非常慢，测试的时候会受到严重的影响。这里哈希表最多126个，查起来还是非常快，不受什么影响。它的速度比AFL快一些，相当于优化了，比原来的算法快一点，并且可以消除碰撞。

2. 改进种子选择策略

优先选择对Coverage有贡献的种子。几个策略：

- 考虑每个种子走过不同的分支，有些分支是被其他种子测过。CollAFL会统计这个种子多少分支被测过，有多少分支没有被测过。举个列子，现在可能有两个种子，第一个种子有一个分支没测过，第二个种子有100个分支没测过，那我们选择哪一个？选择第二个，因为在第二个进行变异的时候有非常大的概率，去测试到没有走过的分支，第一个种子只有一个分支没有被测过，你的变异想触发这个测试种子的概率要低一些。第二个种子有100个分支，你都要对它变异，更大的概率是触发没有走过的分支，会更快提升覆盖率，这是它背后的想法。
- 考虑到内存访问，我们会统计这个种子所走的路径，基本块访问的数量，会优先排序那些访问数量多的。为什么会这么排序呢？因为我们关注的是内存破坏漏洞，如果你的内存访问操作多，会更大概率地触发内存破坏漏洞的想法，所以，会优先排序。

小结

- CollAFL针对覆盖率，进行了提高
- 评判fuzzer的因素：覆盖率、速度、crash数量、漏洞数量等

关于以后的论文分享，**主要**从CCF推荐列表中 A类期刊、会议中选择二进制相关论文。

推荐会议：四大顶会（NDSS、S&P、CCS、USENIX）

E.g. 整理摘要时，吸引你的文章可以讲；别的同学讲过的，但是有些地方仍模糊的，需要补充的，也可以讲。