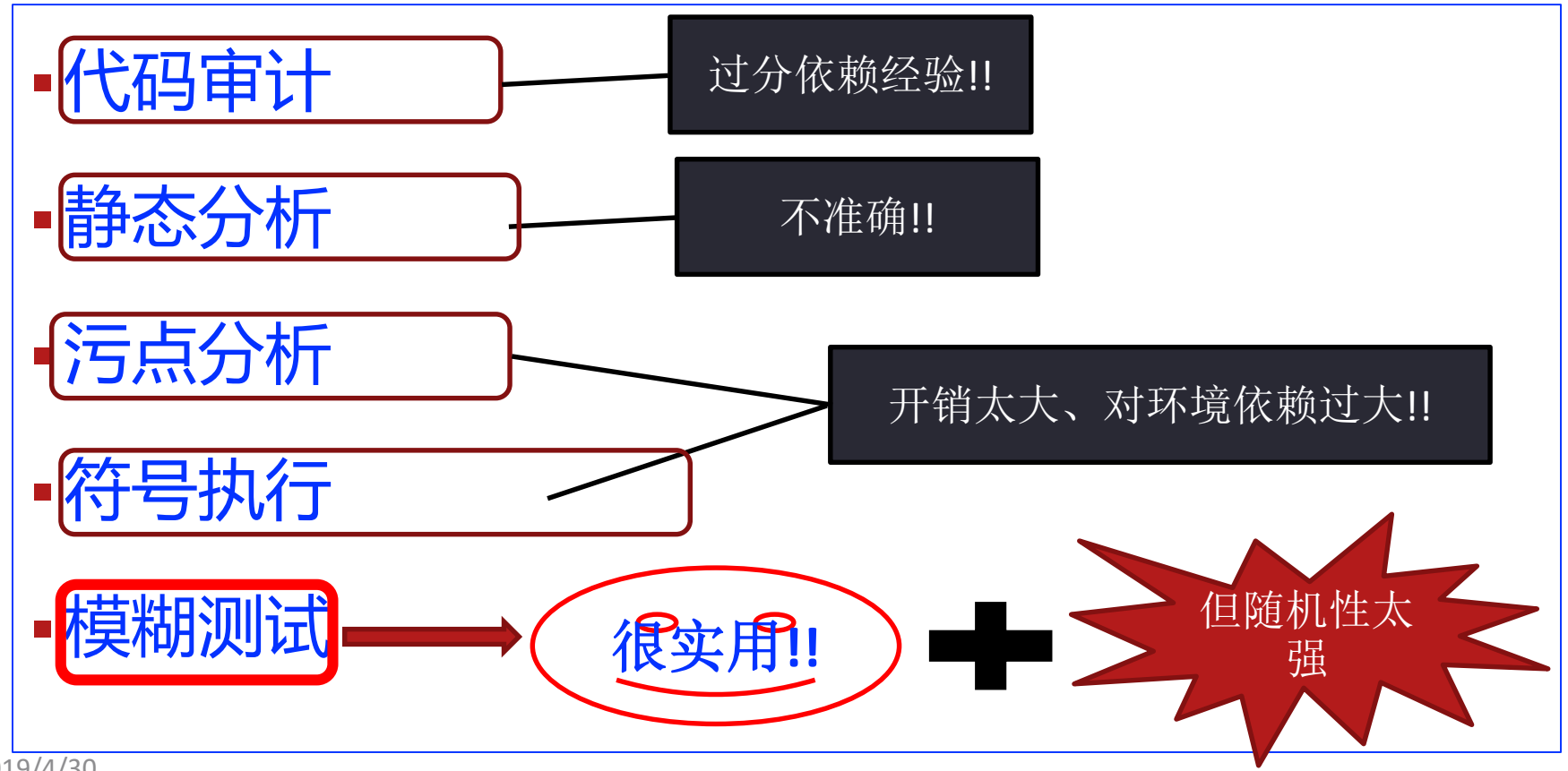
使用数据流敏感模糊测试发现漏洞

Shuitao Gan, **Chao Zhang**

ganshuitao@gmail.com chaoz@tsinghua.edu.cn

漏洞发掘



模糊测试

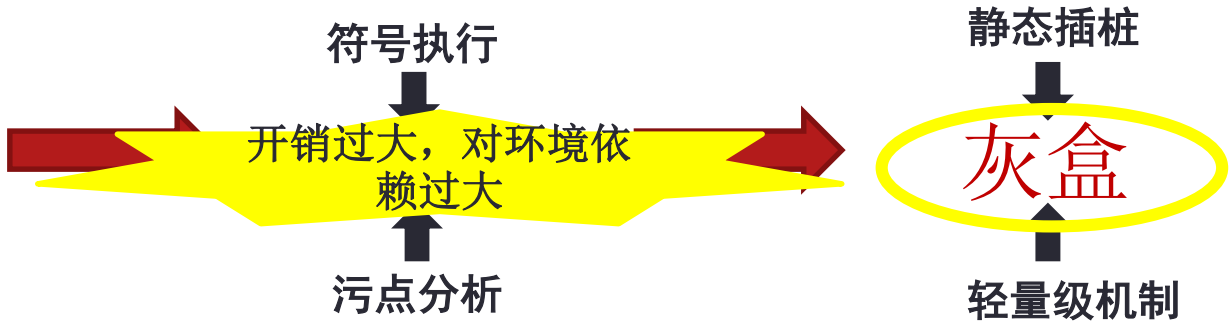
背景介绍

模糊测试

- 基于突变的模型 —— 过于随机
- 基于生成的模型 —— 使用人工经验减少随机性

开发流程：

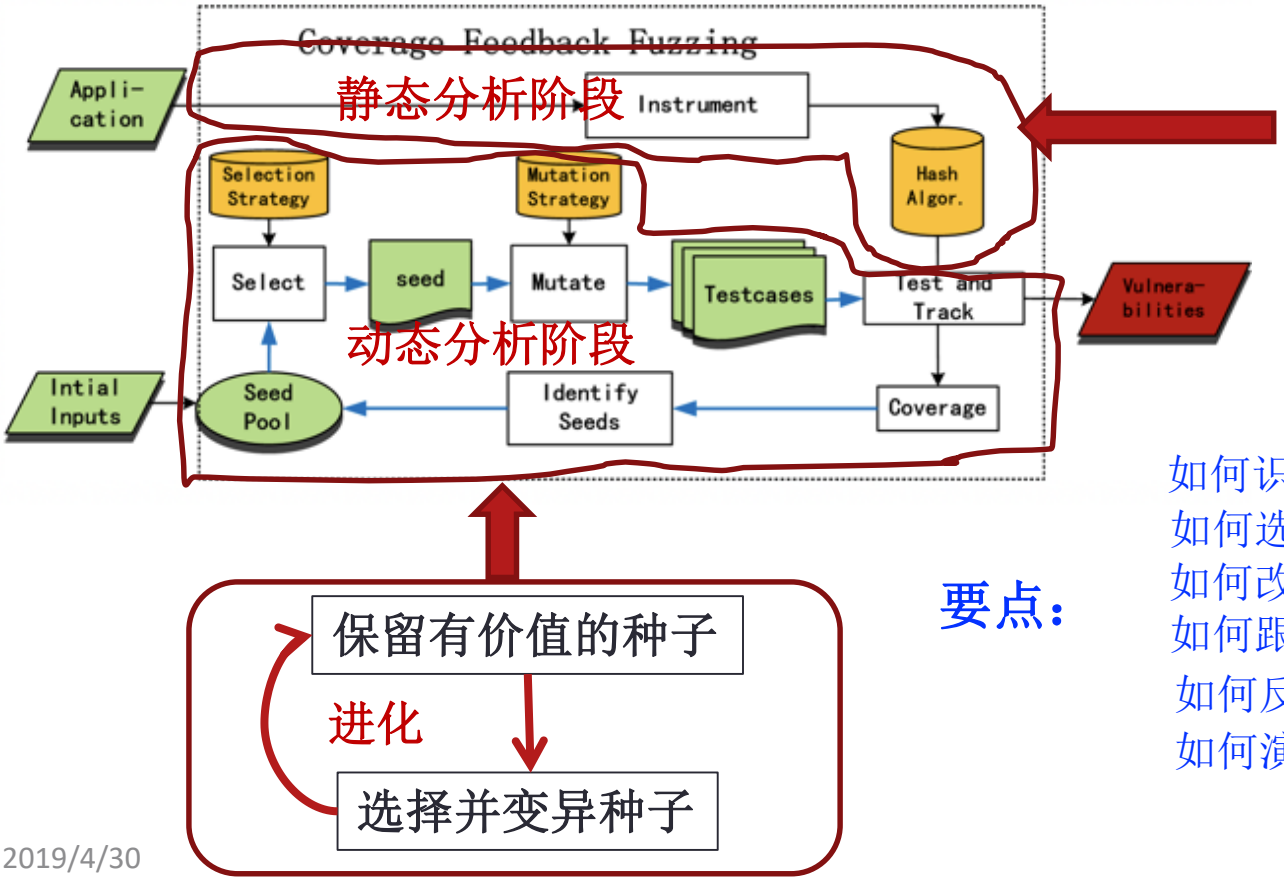
Dumb ➡ Smart

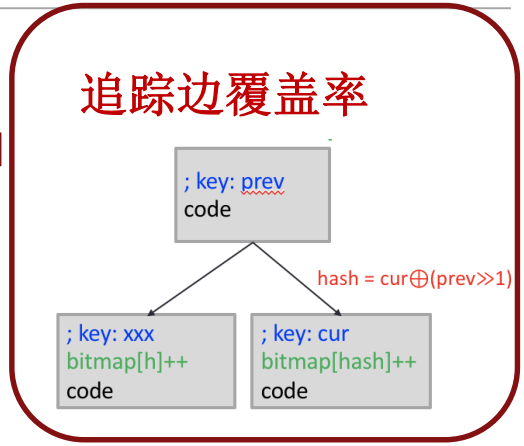


经典灰盒: 基于进化变异的模糊测试

背景介绍

代表性原型：AFL





要点：

如何识别新路径？
如何选择种子？
如何改变种子？
如何跟踪漏洞？
如何反馈？
如何演进？

AFL：如何构建测试方案?

如何反馈？
- ✓ Bitmap/共享内存
- ✓ 存储边覆盖信息
- ✓ 执行一次，反馈一次

如何跟踪漏洞？
- ✓ 利用默认的错误信号监控
- ✓ Sanitizer：捕捉更复杂的错误

哪些因素会影响演进？
- ✓ 种子选择策略
- ✓ 良好的种子存储（队列）

如何识别新路径？
- ✓ 新分支
- ✓ 新循环（摘要）

如何选择种子？
- ✓ 优先考虑跑的更快的路径
- ✓ 优先考虑具有更多分支的路径

如何改变种子？
- ✓ 确定性：位/字节/字典
- ✓ Havoc：拼接+随机

AFL：优点

可扩展性

✓非常少的指令插桩
✓动态阶段的轻量化分析

快速

✓forkserver，一致性模式，并行模式

可演化性

✓代码覆盖率制导/保留具有新路径分支
✓跑的快的种子可能会产生更多新路径
✓具有更多分支条件的种子可以生成更多新路径

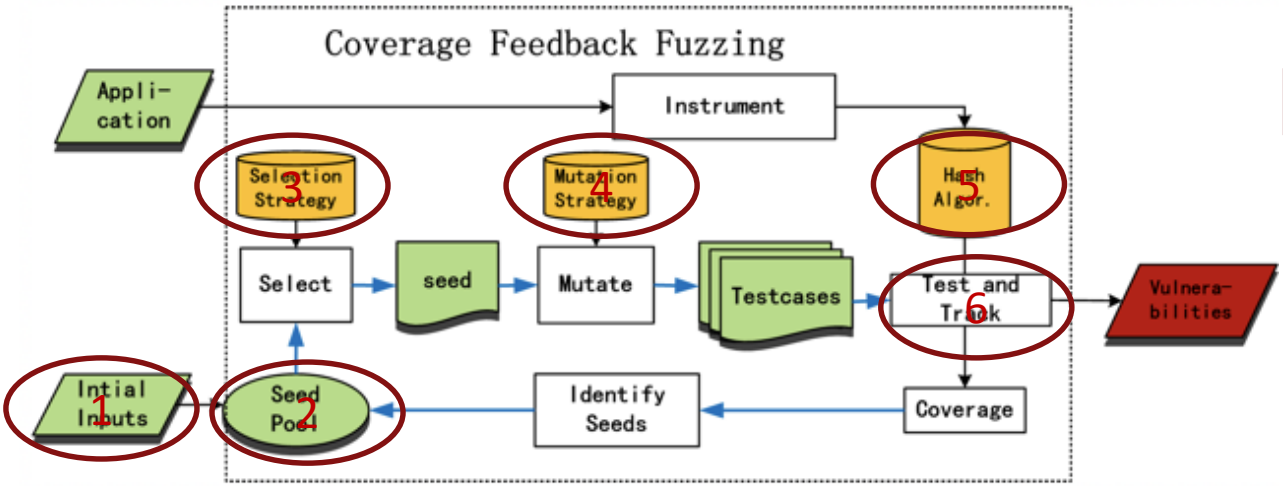
敏感性

✓漏洞类型：Asan / Ubsan / ThreadSan .....

应用对象的可扩展性

✓二进制：AFLdyinst/WinAFL
✓内核：KAFL/TrinityforceAFL

AFL：缺点



1. 如何获得初始种子？
2. 种子池设计太简单
3. 种子选择策略设计太简单
4. 变异策略设计太简单
5. 覆盖率反馈信息不足
6. 速度性能并不完美

处理数据流特征能力很差！！！

基于进化变异的模糊测试优化

相关工作

如何获得初始种子？

非常重要!

- ✓触发复杂的代码
- ✓合适的性能

相关解决方案

- ✓ 手动构建，从互联网搜索
- ✓ 从Crawled Input中学习概率上下文相关文法 **(Skyfile, s&p 17')**
- ✓ 从有效输入中学习循环神经网络（**Microsoft，2017**）
- ✓ 将文法与代码覆盖相结合（**NAUTILUS，NDSS19'**）

如何获得精确的覆盖？

AFL的问题

✓ 由哈希碰撞冲突引起的分支覆盖反馈信息非常不精确

SanitizerCoverage

✓ 通过处理支配节点来跟踪基本块+减少冲突

✓现有路径冲突并且可能表达很少的信息

我们之前的解决方案

✓通过设计独特的静态插桩解决不精确问题**(CollAFL, s&p 17')**

它是进化的原则

✓覆盖更多代码

✓发现更多漏洞

如何选择和更新种子？

进化方向控制

- ✓覆盖更多代码
- ✓发现更多漏洞
- ✓触发相关行为

相关工作

- ✓ AFLFast（CCS‘16）：优先选择产生路径更多或被跑频率较低的路径
- ✓ Vuzzer（NDSS‘17）：优先选择更深的路径
- ✓ QTEP（FSE‘17）：优先选择覆盖更多错误代码的种子
- ✓ AFLgo（CCS‘17）：优先选择更接近脆弱目标的路径
- ✓ SlowFuzz（CCS‘17）：优先选择消耗更多资源的种子

我们之前的工作

- ✓优先选择邻近未触及分支多的种子 **(CollAFL-br, s&p 17’)**
- ✓相比AFL，路径覆盖率增加20%

如何变异？(1)

使模糊测试测试最有效的方法

- ✓ 在哪里变异
- ✓ 变异什么

基于静态分析的优化

- ✓ 递归地分解长常量比较约束
  - ◆ 太多无用的分支
  - ◆ 对非常数比较无能为力
- ✓ 利用静态符号分析来检测输入位之间的依赖关系，并使用它来计算最佳变异率
  - ◆ 太缓慢
  - ◆ 字节之间的依赖关系得到的变异率得到变异策略不够智能

基于学习的模型

- ✓ 基于循环神经网络的模型，预测变异的最佳位置（Rajpal et.al）
  - ◆ 训练速度慢
  - ◆ 获取的位置太多
- ✓ 深层强化学习，变异动作优先排序
  - ◆ 变异策略粒度过于粗糙
- ✓ 程序平滑化和增量学习以指导变异
  - ◆ 缺乏准确的输入分支依赖性

如何变异？(2)

基于符号的解决方案

- ✓ 解决模糊测试测试中的硬约束（Driller，QSYM，DigFuzz）
  - ◆ 受限于符号约束求解的多个难题

基于污点的突变

- ✓ 定位缓冲区溢出漏洞（Dowser，BORG）
- ✓ 跟踪影响敏感库或系统调用的外部种子输入区域（BuzzFuzz）
- ✓ 识别校验和分支（TaintScope）
- ✓ 跟踪Magic字节相关变量（VUzzer）
- ✓ 形状推断和梯度下降计算(Angora)
  - ◆ 传统的动态污点分析，很多未解决的问题

如何优化速度性能？

执行环境

- ✓ Fork
- ✓ Forkserver
- ✓ Persistent
- ✓ IPT

Boosting

- ✓ 多线程处理(Wen Xu,ccs17)
- ✓ 插桩 (Instrim NDSS 18,Untracer s&p19)
  - ◆ 删除不必要的插桩

遗留很多问题……

传统污点分析的瓶颈

- ✓ 消耗大量内存，执行缓慢
- ✓ 外部调用引起的污染不足
- ✓ 隐式控制流引起的污染不足
- ✓ 指定说明引起的过度污染

```
1:  …
2:  int yy0= 0, yy1=0, yy2, yy3;
3:  String xx=ReadSource();
4:  int point = xx.size()/2;
5:  for(int i = 0; i<point;i++){
6:     yy0 +=NormalFun0(xx[i]); /*Normal taint*/
7:     yy2 +=NormalFun1(xx[i]); /*Normal taint*/
8:  }
9:  for(int i = point; i<xx.size();i++){
10:    yy0 +=ExternalFun(xx[i]);/*Truncate taint*/
11:    for(int j=0; j<(int) xx[i]; j++){
12:       yy1 += 1;
13:    }
14: }
15: //br0: yy0's tainting range: [0 , xx.size()/2)
16: if(yy0 == MagicNumber){
17:      …  //Important code
18:}
```

```
19:  //br1: Implicit contral flow make
     //the following branch lose all taints data
20:  if(yy1>Min){
21:      …  //Important code
22:  }
23: //br2: lose harf of taints data
24:  if(yy1 + yy2> Max){
25:      …  //Important code
26:  }
27:  yy3 = yy2&0xff;
28:  if(yy3 > 20){
29:    …  //Important code
30: }
```

遗留很多问题

RQ1: 如何进行轻量级和准确的污点分析以实现高效的模糊测试测试？

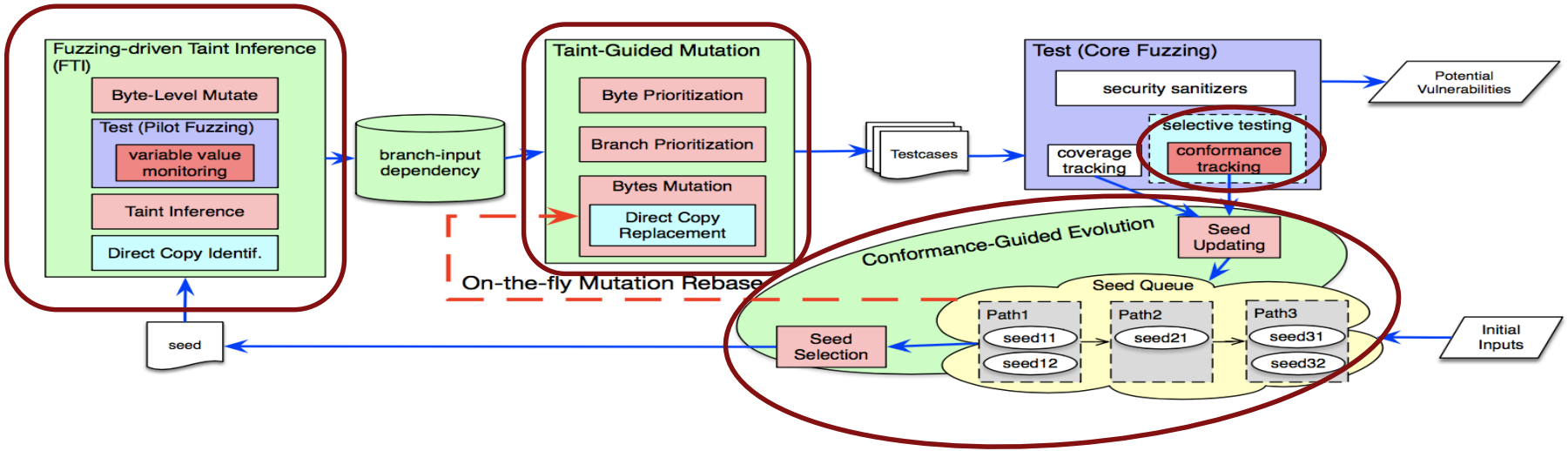
RQ2: 如何用污点数据有效地指导变异？

RQ3: 如何利用数据流分析方法调整模糊测试的进化方向？

GREYONE：数据流敏感模糊测试

我们的解决方案

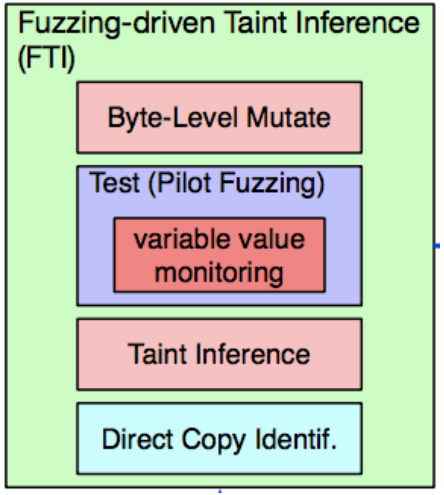
GREYONE 的架构



- FTI：模糊测试驱动的污点推理 → 解决 RQ1
- 污点制导的变异 → 解决 RQ2
- 一致性制导的进化 → 解决 RQ3
- 选择性测试 → 性能优化

Part 1: 模糊测试驱动的污点推理

模糊测试驱动的污点推理



字节级突变

✓ 一组预定义的变异规则

- 单位翻转
- 多位翻转
- 算术运算

变量值监控

✓ 静态插桩

- 具有多位翻转的约束中的变量

污点推理

✓ 污点规则

- 一旦 IF 变量var的值发生变化，我们可以推断var依赖于输入种子S的第pos个字节。

与传统污点分析的比较

速度

- ✓ 传统污点分析
  - 缓慢
  - 动态二进制检测
- ✓ FTI
  - 快速
  - 基于静态代码检测

精准度

- ✓ 传统污点分析
  - 过度污染
  - 污染不足
- ✓ FTI
  - 不会过度污染
  - 污染不足情况减少

手动需求

- ✓ 传统污点分析
  - 需要大量的人力
  - 每条指令的自定义特定污点传播规则
- ✓ FTI
  - 架构独立
  - 移植到新平台无需额外费力

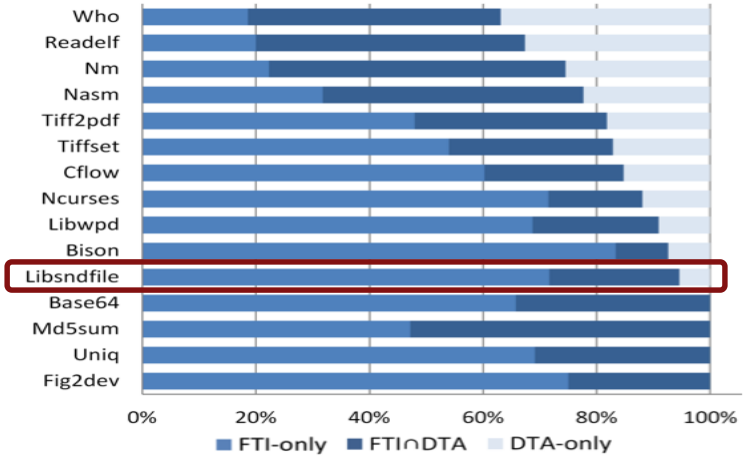
应用程序：分支输入的依赖性

```
1  // magic number: direct copy of input[0:8] vs. constant
2  if(u64(input) == u64("MAGICHDR")){
3      bug1();
4  }
5  // checksum: direct copy input[8:16] vs. computed val
6  if(u64(input+8) == sum(input+16, len-16)){
7      bug2();
8  }
9  // length: direct copy of input[16:18] vs. constant
10 if( u16(input+16) > len ) { bug3(); }
11 // indirect copy of input[18:20]
12 if(foo(u16(input+18))==...){ bug4(); }
13 // implicit dependency: var1 depends on input[20:24]
14 if(u32(input+20) == ...){
15     var1 = ...;
16 }
17 // var1 may change if input[20:24] changes
18 // FTI infers: var1 depends on input[20:24]
19 if(var1 == ...){
20     bug5();
21 }
```

分支输入的依赖性

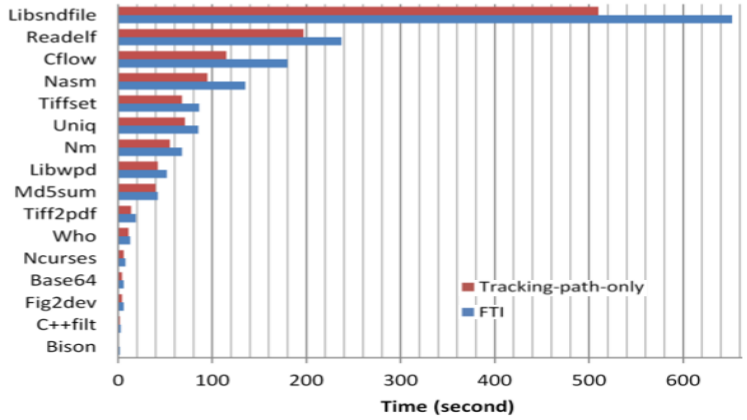
- ✓ 识别直接复制输入数据分支
- ✓ 识别间接复制输入数据分支

FTI的性能



被污染的还未被利用的分支的比例

- ✓ FTI 比常用的污点分析方法DFSan的性能要好
- ✓ FTI 能找出 1.3倍被污染的未触及的分支



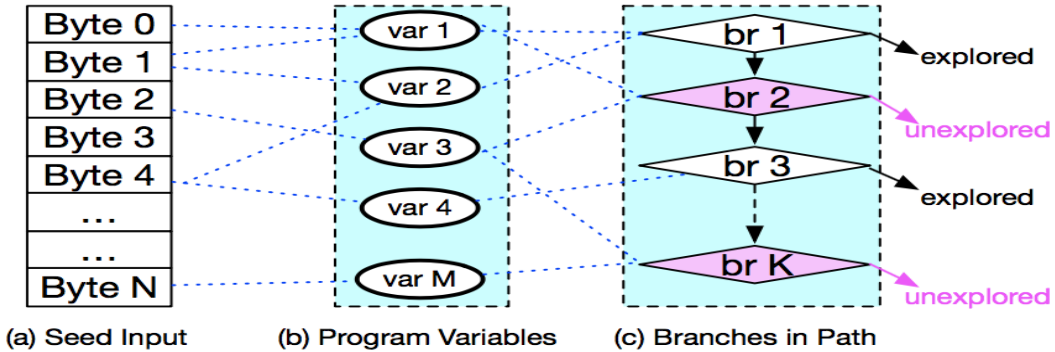
使用FTI分析每个种子的平均速度

- ✓ FTI 平均高出 25%

Part 2: 污点制导的变异

污点制导的变异

- 优先选择字节变异
- 优先选择分支进行变异
- 确定变异的位置和方式



优先选择字节变异

$$W_{byte}(S, pos) = \sum_{br \in Path(S)} IsUntouched(br) * DepOn(br, pos)$$

- 如果到目前为止任何测试用例都没有覆盖br分支，则IsUntouched返回1，否则返回0。
- 如果分支 br 依赖于第 pos个输入字节，则根据FTI，DepOn 返回1，否则返回0。

优先选择分支进行变异

$$W_{br}(S,br) = \sum_{pos \in S} DepOn(br,pos) * W_{byte}(S,pos)$$

相应路径中未覆盖分支br的权重，作为其所有相关输入字节权重的总和

确定变异的位置和方式

在哪里变异

- ✓ 根据路径逐个分析相邻的未覆盖分支
  - ◆ 分支权值降序排列
- ✓ 对于特定的未覆盖的相邻分支
  - ◆ 逐个变换其相关输入字节
  - ◆ 字节权重的降序

如何变异间接复制输入分支

- ✓ 每个相关字节的随机位翻转和算术运算
- ✓ 多个相关字节可以一起变异

缓解污染不足的问题

- ✓ 使用小概率随机变异他们的相邻字节

如何变异直接复制输入分支

- ✓ 执行两次
  - ◆ 第一次用来获得值
  - ◆ 第二次用于覆盖相关分支

Part 3: 一致性制导的进化

数据流特征：约束的一致性

约束的一致性

- ✓ 表示受污染变量与未覆盖分支被覆盖预期值的距离
- ✓ 更高的一致性意味着更低的变异复杂性

Q1：如何评估单一约束？
Q2：如何评估一组约束？

Conformance of one branch

$$C_{br}(br,S) = NumEqualBits(var1, var2)$$

Conformance of a basic block

$$C_{BB}(bb,S) = \underset{br \in Edges(bb)}{MAX} IsUntouched(br) * C_{br}(br,S)$$

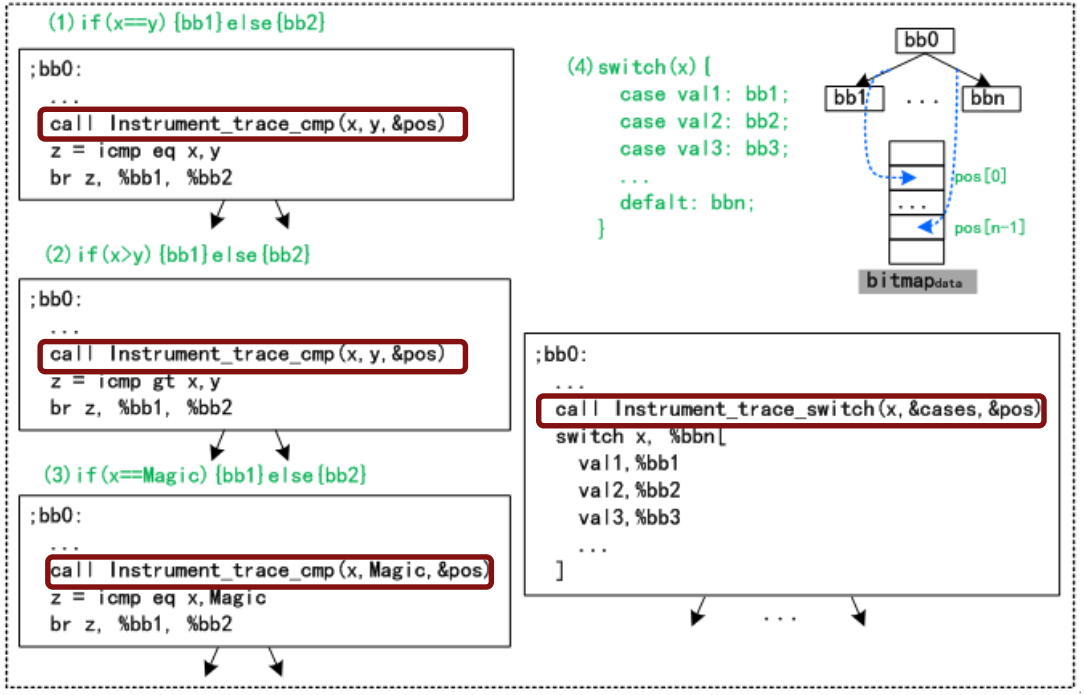
A set of constraints : Conformance of one path

$$C_{seed}(S) = \sum_{bb \in Path(S)} C_{BB}(bb,S)$$

优点

- ✓ 几乎不需要额外的插桩
- ✓ 保留程序的原始结构
- ✓ 可以计算非常数变量比较分支

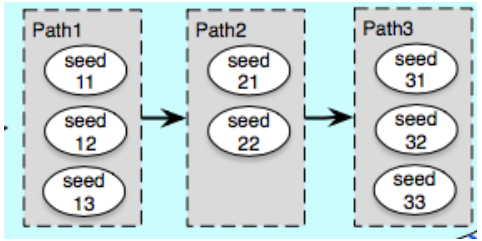
一致性计算的细节


(1) if(x==y) {bb1} else {bb2}
;bb0:
...
call Instrument_trace_cmp(x,y,&pos)
z = icmp eq x,y
br z, %bb1, %bb2
(2) if(x>y) {bb1} else {bb2}
;bb0:
...
call Instrument_trace_cmp(x,y,&pos)
z = icmp gt x,y
br z, %bb1, %bb2
(3) if(x==Magic) {bb1} else {bb2}
;bb0:
...
call Instrument_trace_cmp(x,Magic,&pos)
z = icmp eq x,Magic
br z, %bb1, %bb2
(4) switch(x) {
case val1: bb1;
case val2: bb2;
case val3: bb3;
...
defalt: bbn;
}
bb0
bb1
...
bbn
pos[0]
...
pos[n-1]
bitmapdata
;bb0:
...
call Instrument_trace_switch(x,&cases,&pos)
switch x, %bbn[
val1,%bb1
val2,%bb2
val3,%bb3
...
]
...

一致性引导种子更新

- 二维种子队列

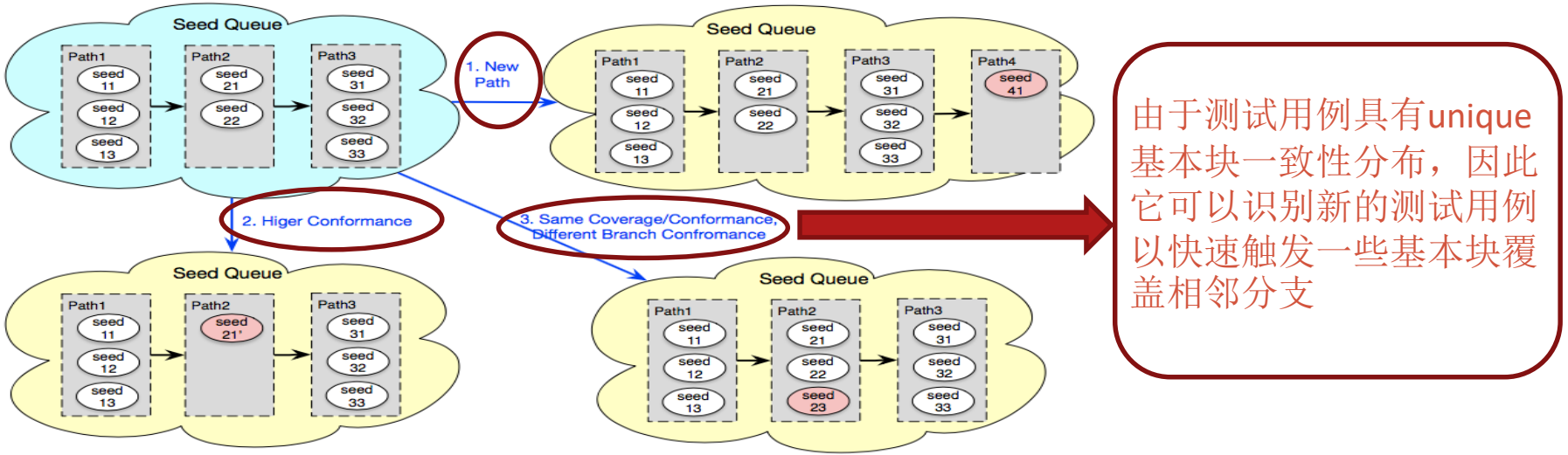
传统种子队列通常保存在链表中，其中每个节点代表跑unqiue路径的单一种子



GREYONE扩展每个节点 包含多个种子，这些跑相同路径的种子具有相同的一致性和不同的基本块一致性，构成二维种子队列

一致性制导的种子更新

- 种子队列更新



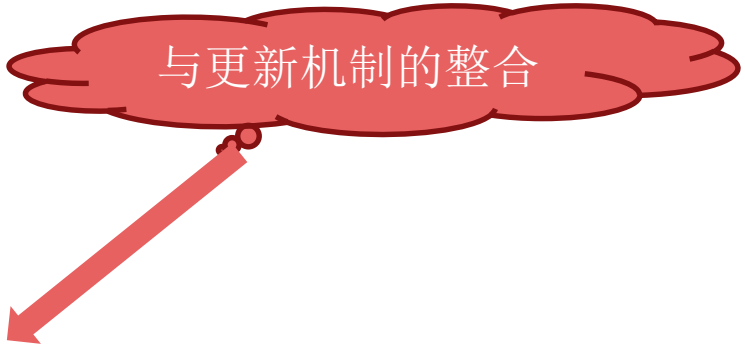
由于测试用例具有unique基本块一致性分布，因此它可以识别新的测试用例以快速触发一些基本块覆盖相邻分支

一致性制导的种子更新

优点

- ✓ 长期稳定的改善
- ✓ 避免陷入局部最小值，如梯度下降算法(s&p 2018)
- ✓ 一致性集中在未受影响的分支上，这比测量Honggfuzz和libfuzzer更好

一致性制导的种子选择



优先考虑高一致性的种子

Advantages: accelerate the evolution of fuzzing

- ✓ 稳步提升覆盖率
- ✓ 避免陷入局部最小值，如梯度下降算法（s&p 2018）
- ✓ 只评估未覆盖分支的一致性，这比测量Honggfuzz和libfuzzer更好

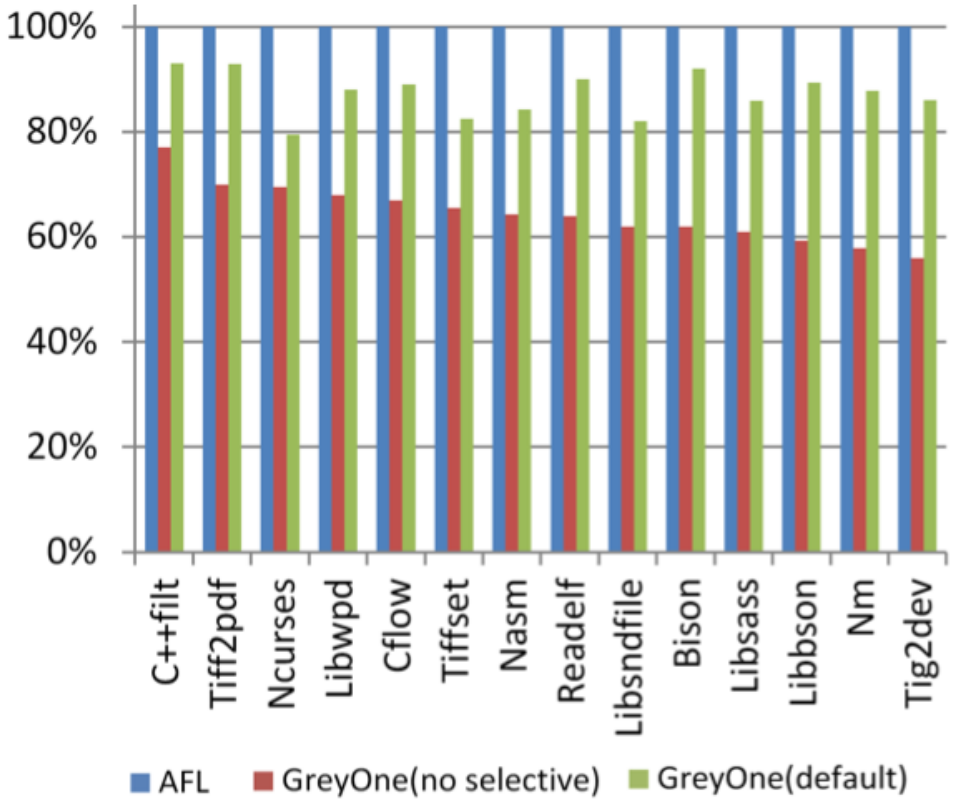
Part 4: 性能优化

性能优化

选择性执行机制

- ✓ GREYONE在测试期间还有两种额外的模式
  - ◆ 用于FTI的变量值监测模式
  - ◆ 用于进化调整的一致性跟踪模式
- ✓ 扩展AFL使用的fork服务器，以便按需切换它们
  - ◆ 当一致性跟踪模式带来过少的一致性提升时，切换到正常跟踪模式

性能优化



选择性执行机制

✓ 通过将这两种模式与AFL进行比较

- 没有选择机制的模式将减慢到低于65%
- GREYONE可以将执行速度保持在80%以上

评估

漏洞发掘

Applications	Version	AFL	CollAFL- br	Honggfuzz	VUzzer	Angora	GREYONE	Vulnerabilities		
								Unknown	Known	CVE
readelf	2.31	1	1	0	0	3	4	2	2	-
nm	2.31	0	0	0	0	0	2	1	1	*
c++filt	2.31	1	1	1	0	0	4	2	2	*
tiff2pdf	v4.0.9	0	0	0	0	0	2	1	1	0
tiffset	v4.0.9	1	2	0	0	0	2	1	1	1
fig2dev	3.2.7a	1	3	2	0	0	10	8	2	0
libwpd	0.1	0	1	0	0	0	2	2	0	2
ncurses	6.1	1	1	0	0	0	4	2	2	2
nasm	2.14rc15	1	2	2	1	2	12	11	1	8
bison	3.05	0	0	1	0	2	4	2	2	0
cflow	1.5	2	3	1	0	0	8	4	4	0
libsass	3.5-stable	0	0	0	0	0	3	2	1	2
libbson	1.8.0	1	1	1	0	0	2	1	1	1
libsndfile	1.0.28	1	2	2	1	0	2	2	0	1
libconfuse	3.2.2	1	2	0	0	0	3	2	1	1
libwebm	1.0.0.27	1	1	0	0	0	1	1	0	1
libsolv	2.4	0	0	3	2	2	3	3	0	3
libcaca	0.99beta19	2	4	1	0	0	10	8	2	6
liblas	2.4	1	2	0	0	0	6	6	0	4
libslax	20180901	3	5	0	0	0	10	9	1	*
libsixl	v1.8.2	2	2	2	2	3	6	6	0	6
libxsmm	release-1.10	1	1	2	0	0	5	4	1	3
Total	-	21	34	18	6	12	105 (+209%)	80	25	41

测试19种流行应用

GREYONE能检测到105个漏洞（41 CVE），是其它工具的3倍

在对每个应用程序进行60小时测试后，包括AFL，CollAFL-br，VUzzer，Honggfuzz，Angora和GREYONE在内的6个模糊测试工具检测到的漏洞数量（累计5次运行）

CVEs

libwpd	CVE-2017-14226, CVE-2018-19208
libtiff	CVE-2018-19210
libbson	CVE-2017-14227,
libncurses	CVE-2018-19217, CVE-2018-19211
libsass	CVE-2018-19218, CVE-2018-19218
libsndfile	CVE-2018-19758
nasm	CVE-2018-19213, CVE-2018-19215, CVE-2018-19216, CVE-2018-20535, CVE-2018-20538, CVE-2018-19755
libwebm	CVE-2018-19212
libconfuse	CVE-2018-19760
libsixel	CVE-2018-19757, CVE-2018-19756, CVE-2018-19762, CVE-2018-19761, CVE-2018-19763, CVE-2018-19763
libsolv	CVE-2018-20533, CVE-2018-20534, CVE-2018-20532
libLAS	CVE-2018-20539, CVE-2018-20536, CVE-2018-20537, CVE-2018-20540
libxsmm	CVE-2018-20541, CVE-2018-20542, CVE-2018-20543
libcaca	CVE-2018-20545, CVE-2018-20546, CVE-2018-20547, CVE-2018-20548, CVE-2018-20544, CVE-2018-20544

Libxsmm: CVE-2018-20541

```
There is a  heap-buffer-overflow in libxsmm_sparse_csc_reader  at src/generator_spgemm_csc_reader.c:174
src/generator_spgemm_csc_reader.c:122) in libxsmm.

Description:

The asan debug is as follows:

$./libxsmm_gemm_generator sparse b a 10 10 10 1 1 1 1 1 1 0 wsm nopf SP POC0

=================================================================
==51909==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000eff0 at pc 0x000000444875 b
WRITE of size 4 at 0x60200000eff0 thread T0
    #0 0x444874 in libxsmm_sparse_csc_reader src/generator_spgemm_csc_reader.c:174
    #1 0x405751 in libxsmm_generator_spgemm src/generator_spgemm.c:279
    #2 0x40225a in main src/libxsmm_generator_gemm_driver.c:318
    #3 0x7f73105a0a3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x20a3f)
    #4 0x402ea8 in _start (/home/company/real_sanitize/poc_check/libxsmm/libxsmm_gemm_generator_asan+0x

0x60200000eff1 is located 0 bytes to the right of 1-byte region [0x60200000eff0,0x60200000eff1)
allocated by thread T0 here:
    #0 0x7f7310c009aa in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x989aa)
    #1 0x443f78 in libxsmm_sparse_csc_reader src/generator_spgemm_csc_reader.c:122
    #2 0x7ffc367e92bf  (<unknown module>)
    #3 0x439  (<unknown module>)
```

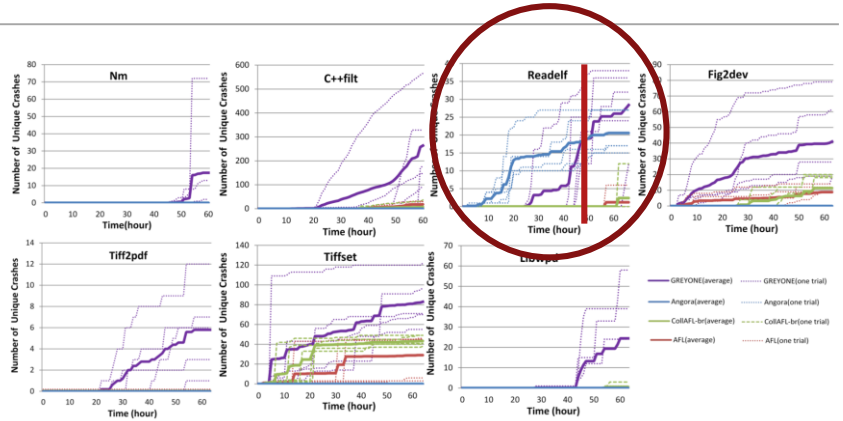
Libsixel:CVE-2018-19757

```
$./img2sixel   POC2
=================================================================
==624==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000a7b1 at pc 0x7fcd853aa04c bp 0x7ffd2dcd54d0 sp
0x7ffd2dcd4c78
WRITE of size 67108863 at 0x60200000a7b1 thread T0
    #0 0x7fcd853aa04b in __asan_memset (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x8804b)
    #1 0x7fcd8508bf10 in memset /usr/include/x86_64-linux-gnu/bits/string3.h:90
    #2 0x7fcd8508bf10 in image_buffer_resize /home/company/real_sanitize/libsixel-master/src/fromsixel.c:311
    #3 0x7fcd8508d5d4 in sixel_decode_raw_impl /home/company/real_sanitize/libsixel-master/src/fromsixel.c:565
    #4 0x7fcd8508e8b1 in sixel_decode_raw /home/company/real_sanitize/libsixel-master/src/fromsixel.c:881
    #5 0x7fcd850c042c in load_sixel /home/company/real_sanitize/libsixel-master/src/loader.c:613
    #6 0x7fcd850c042c in load_with_builtin /home/company/real_sanitize/libsixel-master/src/loader.c:782
    #7 0x7fcd850c43d9 in sixel_helper_load_image_file /home/company/real_sanitize/libsixel-master/src/loader.c:1352
    #8 0x7fcd850cf283 in sixel_encoder_encode /home/company/real_sanitize/libsixel-master/src/encoder.c:1737
    #9 0x4017f8 in main /home/company/real_sanitize/libsixel-master/converters/img2sixel.c:457
    #10 0x7fcd84a88a3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x20a3f)
    #11 0x401918 in _start (/home/company/real_sanitize/poc_check/libsixel/img2sixel+0x401918)

0x60200000a7b1 is located 0 bytes to the right of 1-byte region [0x60200000a7b0,0x60200000a7b1)
allocated by thread T0 here:
    #0 0x7fcd853b59aa in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x989aa)
    #1 0x7fcd8508be1f in image_buffer_resize /home/company/real_sanitize/libsixel-master/src/fromsixel.c:292
```

唯一崩溃评估

Applications	AFL		CollAFL-br		Angora		GREYONE	
	Average	Max	Average	Max	Average	Max	Average	Max
tiff2pdf	0	0	0	0	0	0	6	12
libwpd	0	0	1	3	0	0	21	58
fig2dev	8	12	11	20	0	0	40	79
readelf	0	0	0	0	21	27	28	38
nm	0	0	0	0	0	0	16	72
c++filt	18	30	7	32	0	0	268	575
ncurses	7	18	12	23	0	0	28	37
libsndfile	4	13	8	20	0	0	23	33
libbson	0	0	0	0	0	0	6	12
tiffset	22	46	43	49	0	0	83	122
libsass	0	0	0	0	0	0	8	12
cflow	9	47	17	35	0	0	32	185
nasm	5	15	20	42	6	12	157	212
Total	73	181	119	229	27	39	716 (+501%)	447 (+631%)



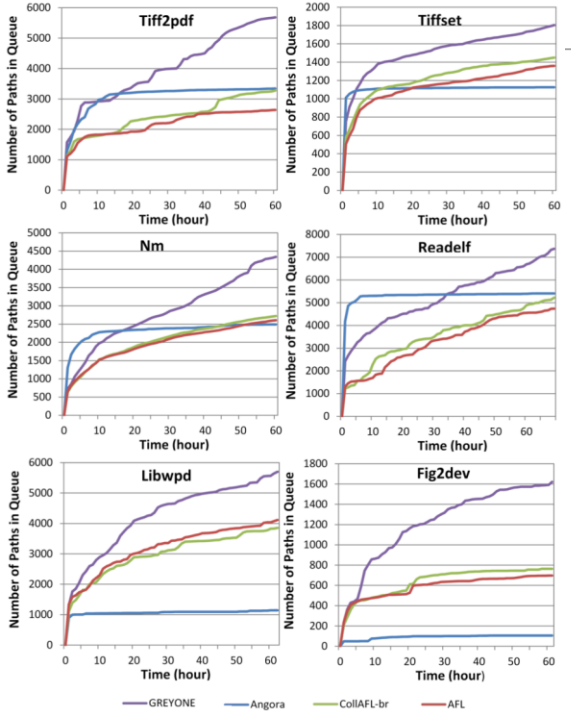
各种模糊测试在现实世界程序中发现的unique crashes数量（5次运行中的平均和最大次数）

AFL , CollAFL-br , Angora和GREYONE检测到的unique crashes 数量（平均值和每次运行5次）的增长趋势

代码覆盖率评估

Applications	Path Coverage				Edge Coverage			
	AFL	CollAFL-br	Angora	GREYONE (INC)	AFL	CollAFL-br	Angora	GREYONE (INC)
tiff2pdf	2638	3278	3344	5681(+69.9%)	6261	6776	6820	8250(+20.9%)
readelf	4519	4782	5212	6834(+32%)	6729	6955	7395	8618(+14.5%)
fig2dev	697	764	105	1622(+112%)	934	1754	489	2460(+40.2%)
ncurses	1985	2241	1024	2926(+30.6%)	2082	2151	1736	2787(+28.2%)
libwpd	4113	3856	1145	5644(+37.2%)	5906	5839	4034	7978(+35.1%)
c++filt	9791	9746	1157	10523(+8%)	6387	6578	3684	7101(+8%)
nasm	7506	7354	3364	9443(+25.8%)	6553	6616	4766	8108(+22.5%)
tiffset	1373	1390	1126	1757(+26%)	3856	3900	3760	4361(+11.8%)
nm	2605	2725	2493	4342(+59%)	5387	5526	5235	8482(+53.5%)
libsndfile	911	848	942	1185(+25.8%)	2486	2392	2525	2975(+17.8%)

各种模糊测试工具在现实世界程序中发现的新路径和新边的数量（5次运行中的平均和最大次数）



AFL , CollAFL-br , Angora和GREYONE检测到的路径数量（平均5次运行）的增长趋势

结论

结论

我们提出了一种新颖的数据流敏感模糊测试解决方案 GREYONE

- ✓ 模糊测试驱动的污点推断方法比传统的动态污点推断方法更有效
- ✓ 它在代码覆盖率和漏洞发现方面表现出比许多流行的模糊测试测试工具（包括AFL，CollAFL，Honggfuzz）更好的性能
- ✓ 它发现105个未知漏洞，其中已获得41个CVE

谢谢!

Q&A