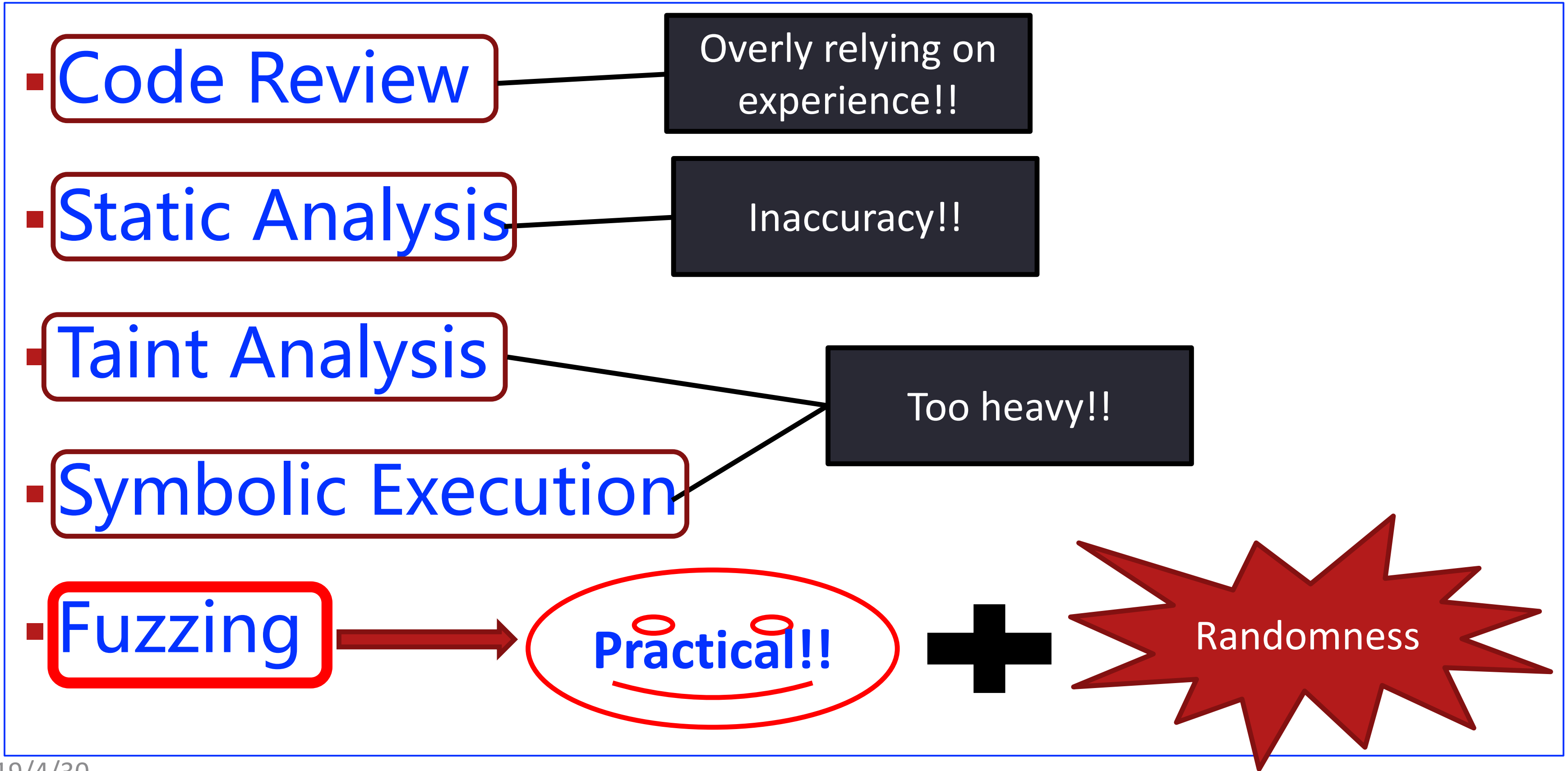
Discovering Vulnerabilities with Data Flow Sensitive Fuzzing

Shuitao Gan, **Chao Zhang**

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

BLUEHAT SHANGHAI 2019

Vulnerability Discovery



Fuzzing

background

Fuzzing

- Mutation-based Model — Too random
- Generation-based Model — Reduce randomness using manual experience

Developing process : Dumb → Smart

Symbolic execution

Blackbox → **Too Heavy!** → Greybox

Taint analysis

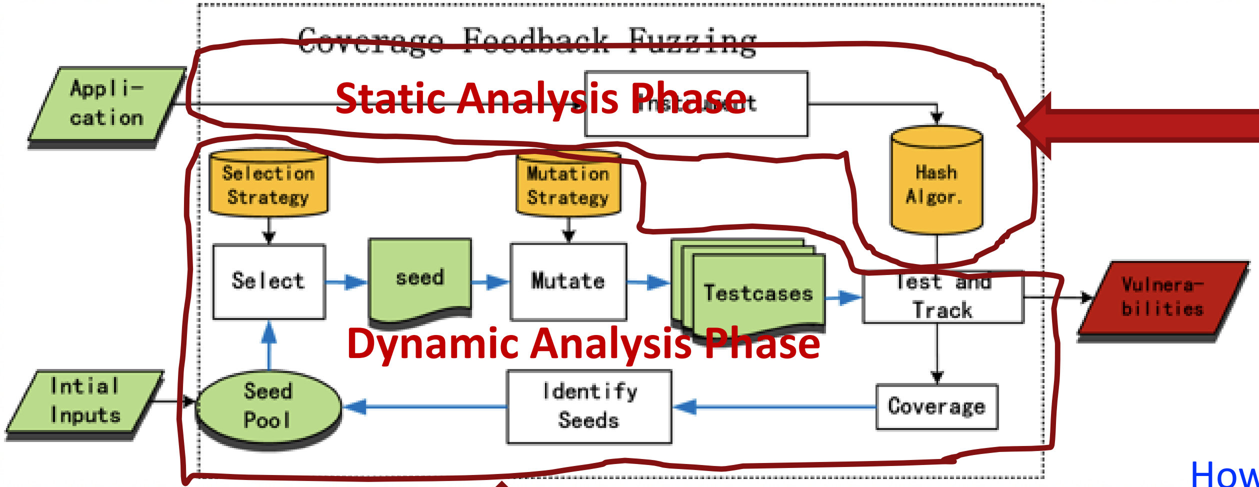
Static instrumentation

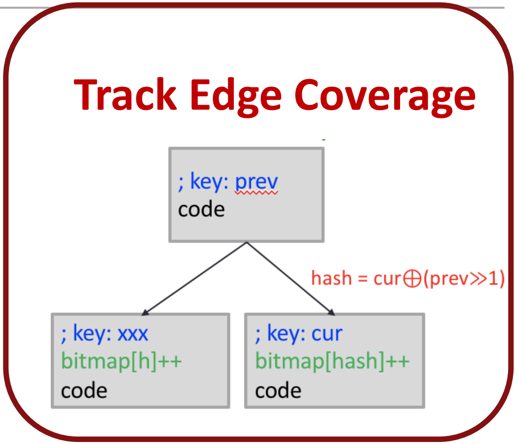
Light-weight mechanism

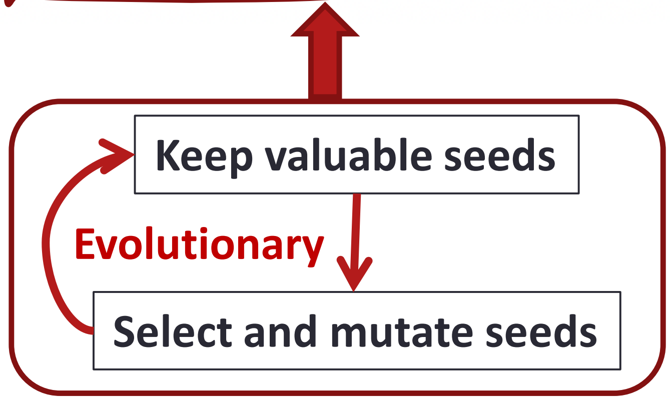
Classic Greybox: Evolutionary Mutation-based Fuzzing

background

Representive prototype : AFL







Key points:

How to identity new path?
How to select seeds?
How to mutate seeds?
How to track vulnerabilities?
How to feedback?
How to evolute?

AFL : How to do?

How to feedback?

- ✓Bitmap/shared memory
- ✓Store edges
- ✓Execute one time, feedback one time

How to track vulnerabilities?

- ✓Default error signal monitor
- ✓Sanitizer: catch more sophisticated errors

What factors affect evolution?

- ✓Seeds selection policies
- ✓Good seeds storage (queue)

How to identity new path?

- ✓New edge
- ✓New loop (abstract)

How to select seeds?

- ✓Prioritize to quick paths
- ✓Prioritize to paths with more edges

How to mutate seeds?

- ✓Deterministic : bit/byte/dictionary
- ✓Havoc : splice+random

AFL : Advantages

Scalability

✓Few instructions instrumentation
✓Light-weight analysis in the dynamic phase

Fast

✓Forkserver, persistent mode, parallel

Evoluting

✓Code coverage guided/keep paths with new edges
✓Fast seeds may generate more new paths
✓Seeds with more edges may generate more new paths

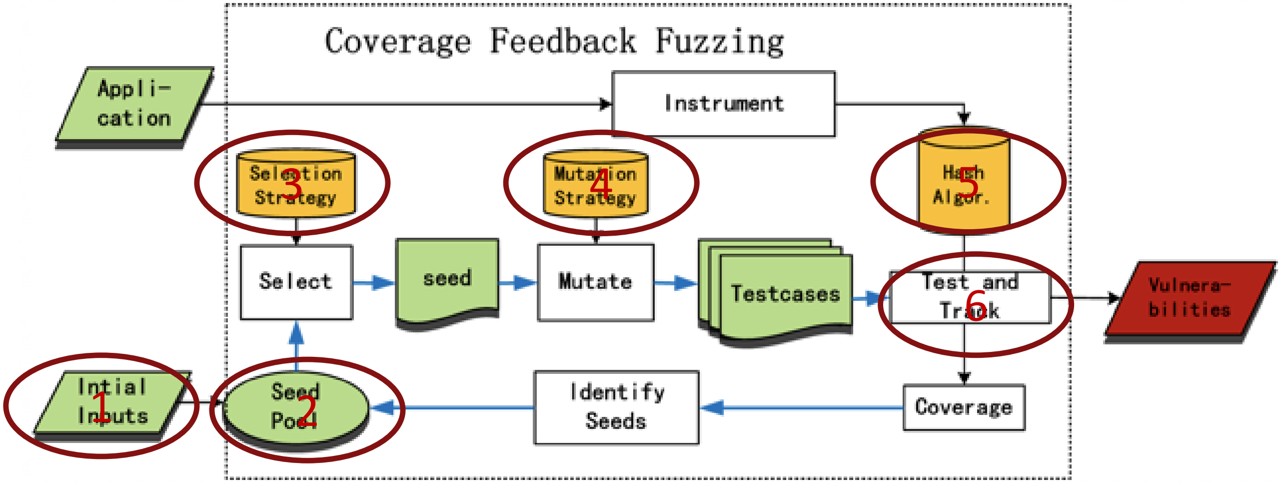
Sensitivity

✓Vulnerabilities types: Asan/Ubsan/ThreadSan…

Extension

✓Binary : AFLdyinst/WinAFL
✓Kernel : KAFL/TrinityforceAFL

AFL : Weaknesses



1. How to get initial seeds?

2. Weak seed pool

3. Weak selection policy

4. Weak mutation strategy

5. Weak coverage feedback

6. Speed performance is not perfect

Very pool in processing data flow features ！ ！ ！

Optimization on Evolutionary Mutation-based Fuzzing

related work

How to get initial seeds ?

It is important!

✓trigger complex code
✓appropriate performance

Related solutions

✓manual constructing, searching from internet
✓learn probabilistic Context-Sensitive Grammar from crawled inputs **(Skyfile, s&p 17')**
✓learn RNN from valid inputs**(Microsoft, 2017)**
✓ combine grammars with code coverage**(NAUTILUS, NDSS19')**

How to get precise coverage ?

Problem of AFL

✓ Extremely imprecise in edge coverage caused by hash collision

It is the guidance of evolution

✓Covering more code

✓Discovering more vulnerabilities

SanitizerCoverage

✓ Tracking basic blocks + reducing collision by processing dominate node

✓Existing path collision and may express few information

Our previous solution

✓ Solve the imprecise by static instrumentation **(CollAFL, s&p 17')**

How to select and update seeds ?

Evolutionary direction control

- ✓Covering more code
- ✓Discovering more vulnerabilities
- ✓Triggering relevant behavior

Related work

- ✓ AFLFast (CCS’16): seeds being picked fewer or exercising less-frequent paths
- ✓ Vuzzer (NDSS’17): seeds exercising deeper paths
- ✓ QTEP (FSE’17): seeds covering more faulty code
- ✓ AFLgo (CCS’17): seeds closer to target vulnerable paths
- ✓ SlowFuzz (CCS’17): seeds consuming more resources

Our previous solution

- ✓Prioritize seeds with more untouched branches**(CollAFL-br, s&p 17’)**
- ✓20% more paths over AFL

How to mutate ? (1)

The most efficient way to make fuzzing smart

- ✓Where to mutate
- ✓What to mutate

Static analysis-based optimization

- ✓ Decomposing long constant comparisons constraint recursively
  - Too many useless branches
  - Helpless on non-constant comparisons
- ✓ Leverages static symbolic analysis to detect dependencies among input bits, and uses it to compute an optimal mutation ratio
  - Slowly
  - The calculated dependency between bits do not show many improvements for mutation.

Learning-based model

- ✓ RNN-based model, predicting best locations to mutate (Rajpal et.al)
  - Slow training speed
  - Get too many locations
- ✓ Deep reinforcement learning, mutation actions prioritization
  - The granularity of mutation actions are too coarse
- ✓ Program smoothing and incremental learning to guide mutation
  - Lack of accurate input-branches dependence

How to mutate ? (2)

Symbolic-based solution

- ✓ Solve hard constraints in fuzzing (Driller, QSYM, DigFuzz)
  - open challenge of constraint solving

Taint-based mutation

- ✓ Locating buffer boundary violations and buffer over-read vulnerabilities (Dowser, BORG)
- ✓ Tracking the regions of external seed inputs that affect sensitive library or system calls (BuzzFuzz)
- ✓ Identifying checksum branch (TaintScope)
- ✓ Tracking magic bytes related variables (VUzzer)
- ✓ shape inference and gradient descent computation (Angora)
  - Traditional dynamic taint analysis, many open problems

How to optimize speed performance ?

Execution environment

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

Boosting

- ✓ Parallel execution(Wen Xu,ccs17)
- ✓ Instrumentation (Instrim NDSS 18,Untracer s&p19)
  - ◆ Removing unnecessary instrumentation

Leave many questions …

Bottleneck of traditional taint analysis

- ✓ Consume large memory, execute slowly
- ✓ Under-taint by external call
- ✓ Under-taint by implicit control flow
- ✓ Over-taint by specified instructions

```
…
int yy0= 0, yy1=0, yy2, yy3;
String xx=ReadSource();
int point = xx.size()/2;
for(int i = 0; i<point;i++){
    yy0 +=NormalFun0(xx[i]); /*Normal taint*/
    yy2 +=NormalFun1(xx[i]); /*Normal taint*/
}
for(int i = point; i<xx.size();i++){
    yy0 +=ExternalFun(xx[i]);/*Truncate taint*/
    for(int j=0; j<(int) xx[i]; j++){
        yy1 += 1;
    }
}
//br0: yy0's tainting range: [0 , xx.size()/2)
if(yy0 == MagicNumber){
    … //Important code
}
 //br1: Implicit contral flow make
//the following branch lose all taints data
 if(yy1>Min){
    … //Important code
 }
//br2: lose harf of taints data
 if(yy1 + yy2> Max){
    … //Important code
 }
yy3 = yy2&0xff;
if(yy3 > 20){
  … //Important code
}
```

Leave many questions …

RQ1: How to perform lightweight and accurate taint analysis for efficient fuzzing ?

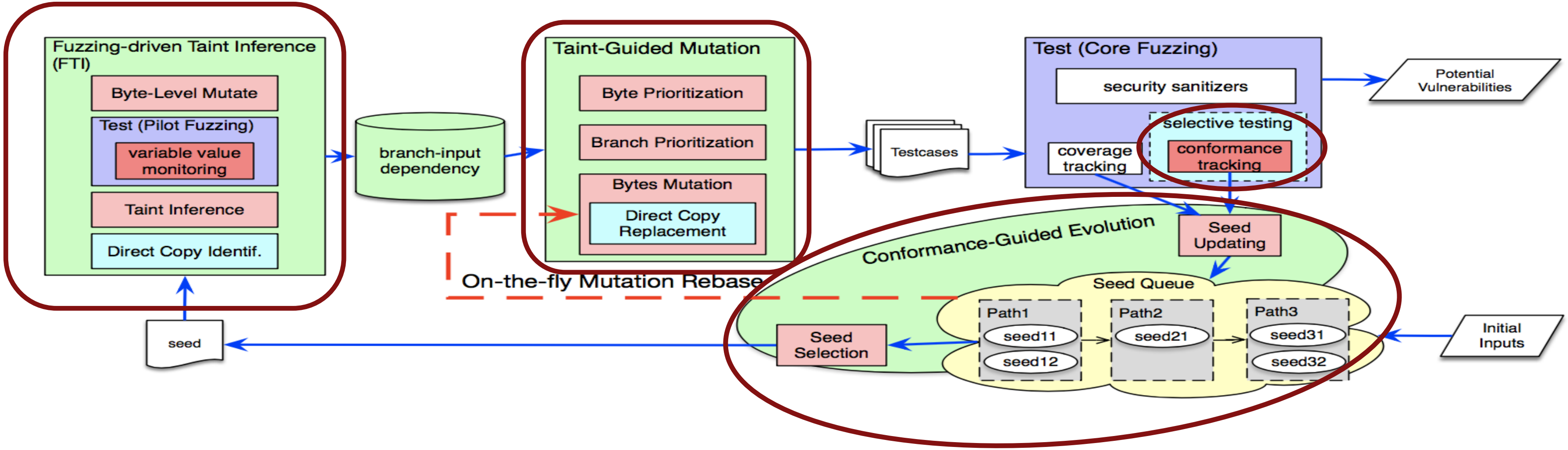
RQ2: How to efficiently guide mutation with taint?

RQ3: How to tune fuzzers' evolution direction with data flow features?

GREYONE: Data Flow Sensitive Fuzzing

Our Solution

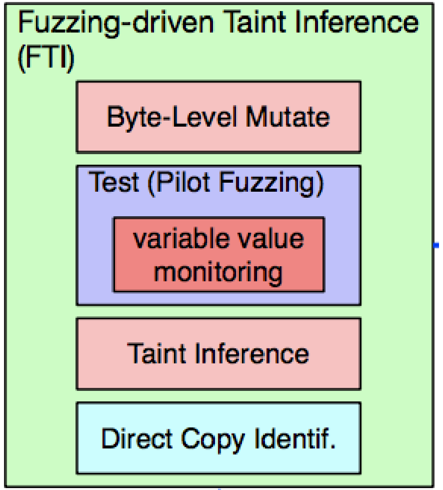
Architecture of GREYONE



- FTI :Fuzzing-driven Taint Inference → Solve RQ1
- Taint-Guided Mutation → Solve RQ2
- Conformance-Guided Evolution → Solve RQ3
- Selective testing → Performance optimization

Part 1: Fuzzing-driven Taint Inference

Fuzzing-driven Taint Inference



Byte-level Mutation

- ✓ A set of predefined mutation rules
  - ◆ Single bit flipping
  - ◆ Multiple-bits flipping
  - ◆ Arithmetic operations

Variable Value Monitoring

- ✓ Static instrumentation
  - ◆ Variables in constraints with multiple-bits flipping

Taint Inference

- ✓ Taint rule
  - ◆ Multiple If the value of a variable var changes, we could infer that var is tainted and depends on the pos-th byte of the input seed S.

Comparison with Traditional Taint Analysis

Speed

- ✓ Traditional taint analysis
  - ◆ Slow
  - ◆ Dynamic binary instrumentation
- ✓ FTI
  - ◆ Fast
  - ◆ Based on static code instrumentation

Accuracy

- ✓ Traditional taint analysis
  - ◆ Over-taint
  - ◆ Under-taint
- ✓ FTI
  - ◆ No over-taint
  - ◆ Less under-taint

Manual Efforts

- ✓ Traditional taint analysis
  - ◆ Labor-intensive efforts
  - ◆ Custom specific taint propagation rules for each instruction
- ✓ FTI
  - ◆ Architecture independent
  - ◆ No extra efforts to port to new platforms

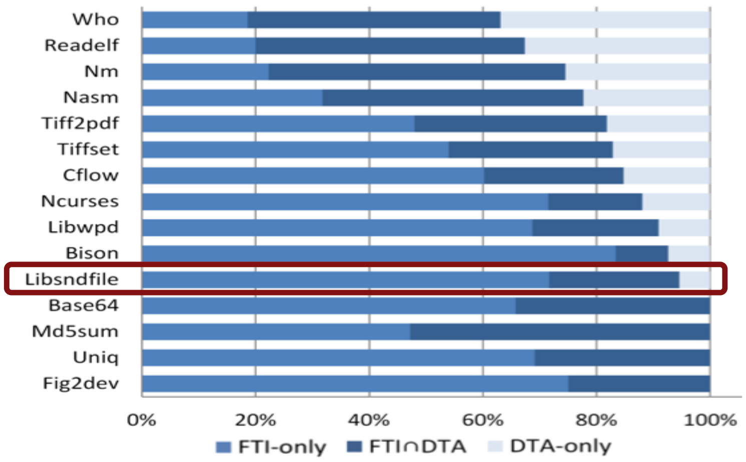
Application : Branch-Input Dependency

```
// magic number: direct copy of input[0:8] vs constant
if(u64(input) == u64("MAGICHDR")){
    bug1();
}
// checksum: direct copy input[8:16] vs. computed val
if(u64(input+8) == sum(input+16, len-16)){
    bug2();
}
// length: direct copy of input[16:18] vs. constant
if( u16(input+16) > len )) { bug3(); }
// indirect copy of input[18:20]
if(foo(u16(input+18))==...){ bug4(); }
// implicit dependency: var1 depends on input[20:24]
if(u32(input+20) == ...){
    var1 = ...;
}
// var1 may change if input[20:24] changes
// FTI infers: var1 depends on input[20:24]
if(var1 == ...){
    bug5();
}
```

Branch-Input Dependency

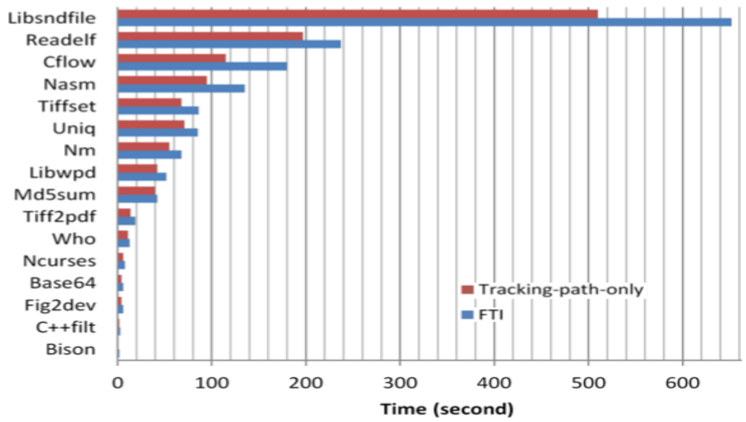
- ✓ Identify Direct Copies of Inputs
- ✓ Identify InDirect Copies of Inputs

Performance of FTI



Proportion of tainted untouched branches reported

- ✓ FTI outperforms the classic taint analysis solution DFSan
- ✓ FTI finds 1.3X more untouched branches that are tainted



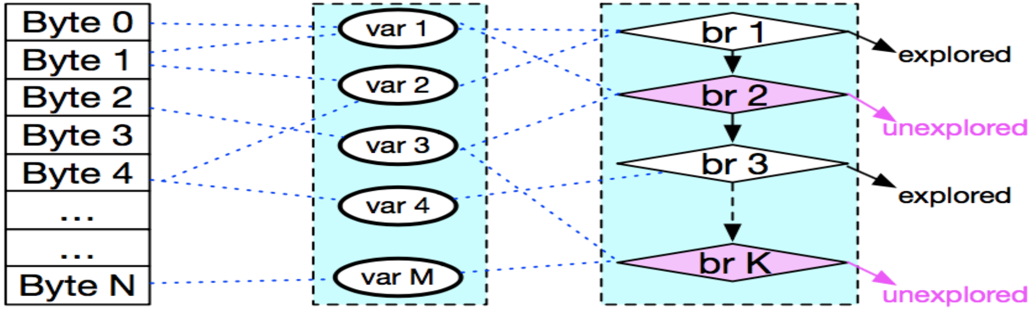
Average speed of analyzing one seed by FTI

- ✓ FTI brings 25% overhead on average

Part 2: Taint-guided Mutation

Taint-guided Mutation

- **Prioritize Bytes to Mutate**
- **Prioritize Branches to Explore**
- **Determine Where and How to Mutate**



(a) Seed Input (b) Program Variables (c) Branches in Path

Prioritize Bytes to Mutate

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

- **IsUntouched** returns 1 if the branch br is not explored by any test case so far, otherwise 0.
- **DepOn** returns 1 if the branch br depends on the pos-th input byte, according to FTI, otherwise 0.

Prioritize Branches to Explore

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

The weight of an untouched branch br in the according path as the sum of all its dependent input bytes' weight

Determine Where and How to Mutate

Where to mutate

- ✓ Exploring the untouched neighbor branches along this path one by one
  - ◆ Descending order of branch weight
- ✓ For specific untouched neighbor branch
  - ◆ Mutating its dependent input bytes one by one
  - ◆ Descending order of byte weight

Mitigate the under-taint issue

- ✓ Randomly mutating their adjacent bytes with a small probability

How to mutate indirect copies of input

- ✓ Random bit flipping and arithmetic operations on each dependent byte
- ✓ Multiple dependent bytes could be mutated together

How to mutate direct copies of input

- ✓ Executing twice
  - ◆ The first time used to get value
  - ◆ The second time used to cover relevant branch

Part 3: Conformance-Guided Evolution

Data flow features: conformance of constraints

Conformance of constraints

- ✓ Expressing the distance of tainted variables to the values expected in untouched branches
- ✓ Higher conformance means lower complexity of mutation

Q1: How to evaluate single constraint?

Q2 How to evaluate a set of constraints?

Conformance of one branch

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

Conformance of a basic block

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

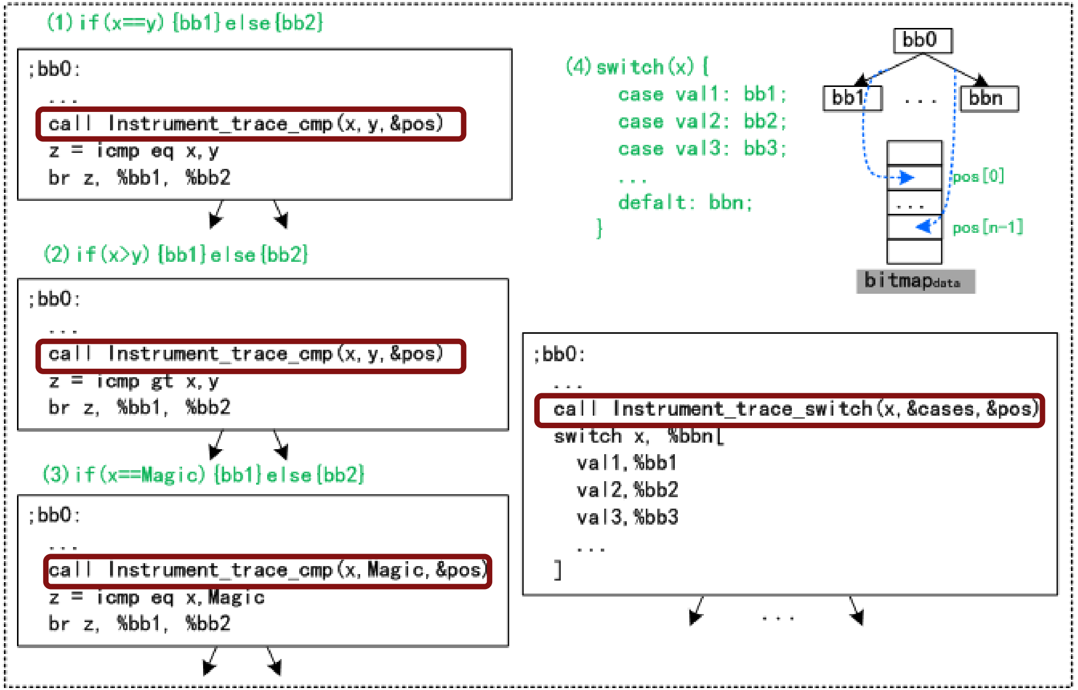
A set of constraints : Conformance of one path

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

Advantages

- ✓ Few extra instrumented overhead
- ✓ Keep the original construct of program
- ✓ Non-constant variables comparison branch could be calculated

Details of Conformance Calculation

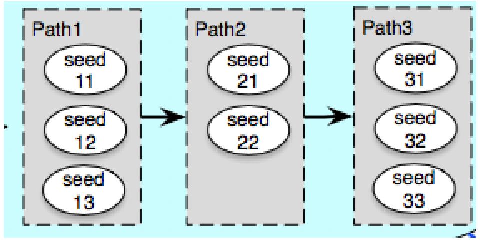


Conformance-Guided Seed Updating

▪ Two-Dimensional Seed Queue

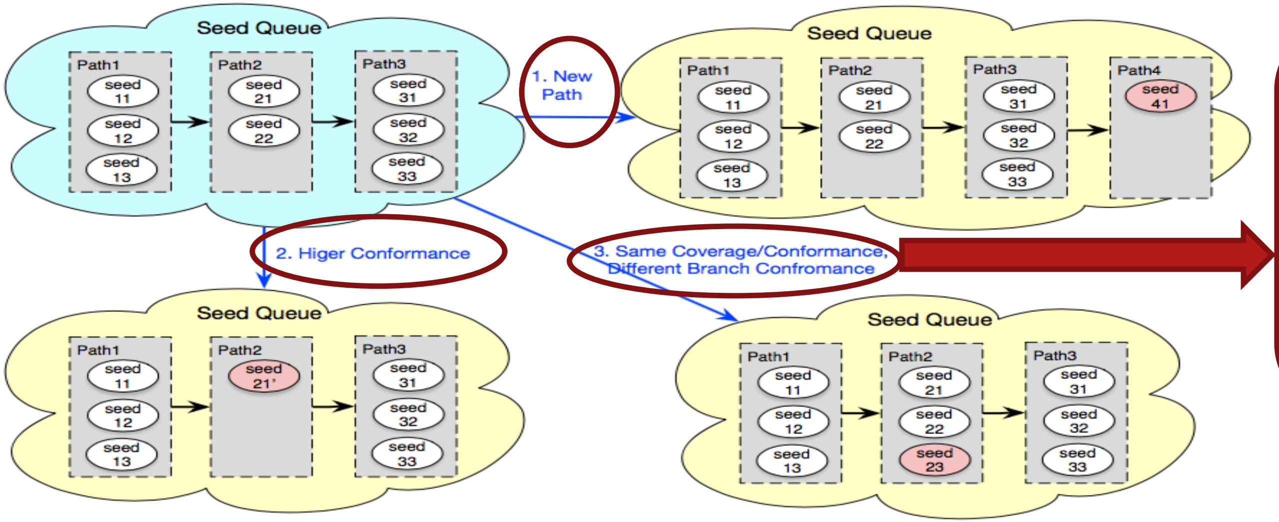
Traditional seed queues are usually kept in a linked list, where each node represents a seed that explores a unique path

GREYONE extend each node to include multiple seeds that explore a same path and have a same conformance but different block conformance, to form a two-dimensional seed queue



Conformance-Guided Seed Updating

- Seed queue Updates



since the test case has a unique distribution of basic block conformance, it could derive new test cases to quickly trigger untouched neighbor branches of some basic blocks

Conformance-Guided Seed Updating

Advantages

- ✓ Long-term stable improvements
- ✓ Avoid getting stuck in local minimum like gradient descent algorithm(s&p 2018)
- ✓ The conformance focuses on untouched branches, which is better than the measurement of Honggfuzz and libfuzzer

Conformance-Guided Seed Selection

Combining with updating mechanism

Giving priority to seeds with high conformance

Advantages: accelerate the evolution of fuzzing

- ✓ Long-term stable improvements
- ✓ Avoid getting stuck in local minimum like gradient descent algorithm(s&p 2018)
- ✓ The conformance focuses on untouched branches, which is better than the measurement of Honggfuzz and libfuzzer

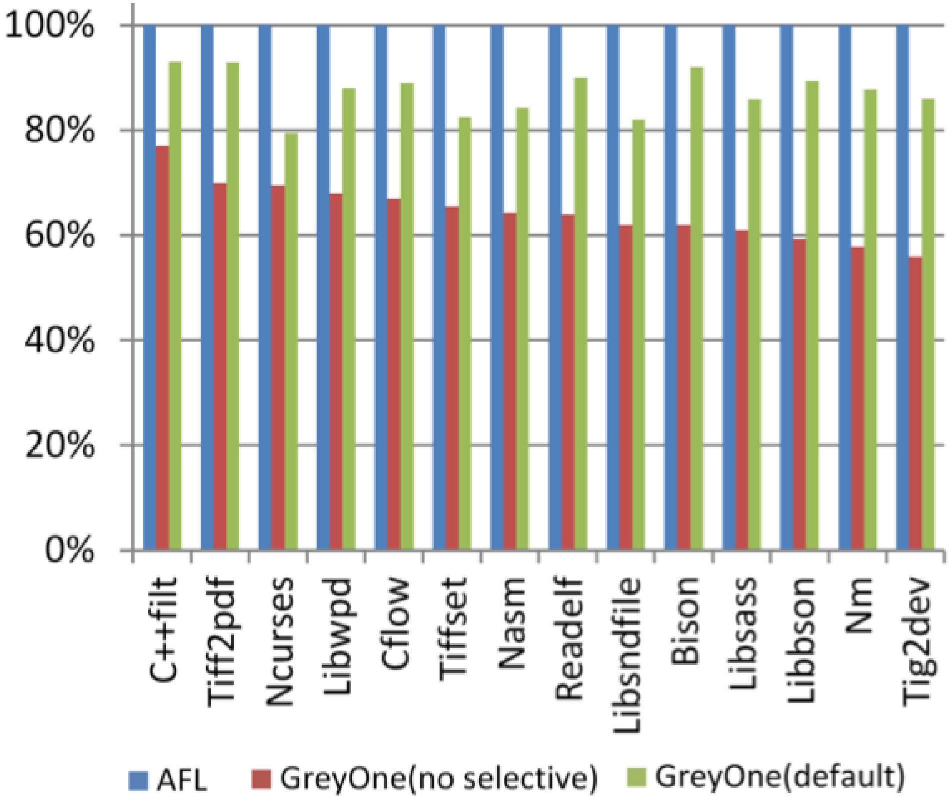
Part 4: Performance Optimization

Performance Optimization

Selective execution mechanism

- ✓ GREYONE has two more modes during testing
  - ◆ Variable value monitoring mode used for FTI
  - ◆ Conformance-guided tracking mode for evolution tuning
- ✓ Extending the fork server used by AFL to switch between them on demand
  - ◆ When conformance tracking mode brought few conformance promotion, switching to normal tracking mode

Performance Optimization



Selective execution mechanism

✓ **By comparing these two mode with AFL**

- The mode without selective mechanism will slow down to less than 65%
- GREYONE's could keep execution speed more than 80%

Evaluation

Vulnerabilities Discovery

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

Testing 19 popular applications

GREYONE detected 209% more vulnerabilities (41 CVEs)

Number of vulnerabilities (accumulated in **5 runs**) detected by 6 fuzzers, including AFL, CollAFL-br, VUzzer, Honggfuzz,Angora, and GREYONE, after testing each application for **60 hours**

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

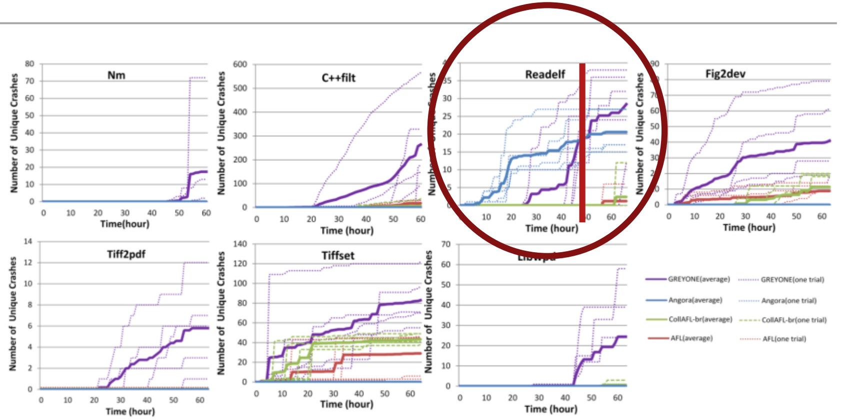
Libxsmm: CVE-2018-20541

```
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

=================================================================
==51000==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>)


$./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+0x8c04b)
    #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
```

Libsixel:CVE-2018-19757

Unique Crashes Evaluation

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%)



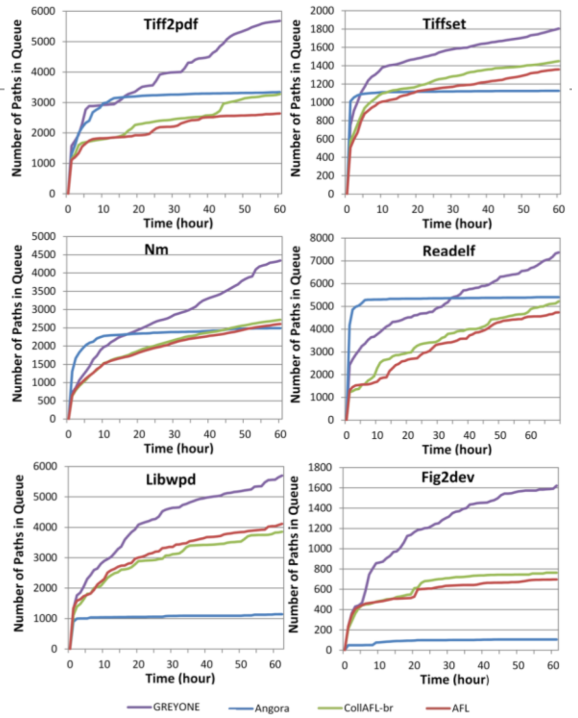
Number of unique crashes (average and maximum count in 5 runs) found in real world programs by various fuzzers

The growth trend of number of unique crashes (average and each of 5 runs) detected by AFL, CollAFL-br, Angora and GREYONE

Code Coverage Evaluation

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%)

Number of unique crashes (average and maximum count in 5 runs) found in real world programs by various fuzzers



The growth trend of number of unique paths (average in 5 runs) detected by AFL, CollAFL-br, Angora and GREYONE

Conclusion

Conclusions

We propose a novel data flow sensitive fuzzing solution GREYONE

- ✓ Where Fuzzing-driven taint inference is further more efficient than traditional dynamic taint inference
- ✓ It performs better performance than many popular fuzzing tools including AFL, CollAFL, Honggfuzz in terms of code coverage and vulnerabilities discovery
- ✓ It detected 105 unknown vulnerabilities with 41 CVEs

Thanks!

Q&A