

VeriFuzz: Program Aware Fuzzing

M. Raveendra Kumar (Raveendra.kumar@tcs.com)

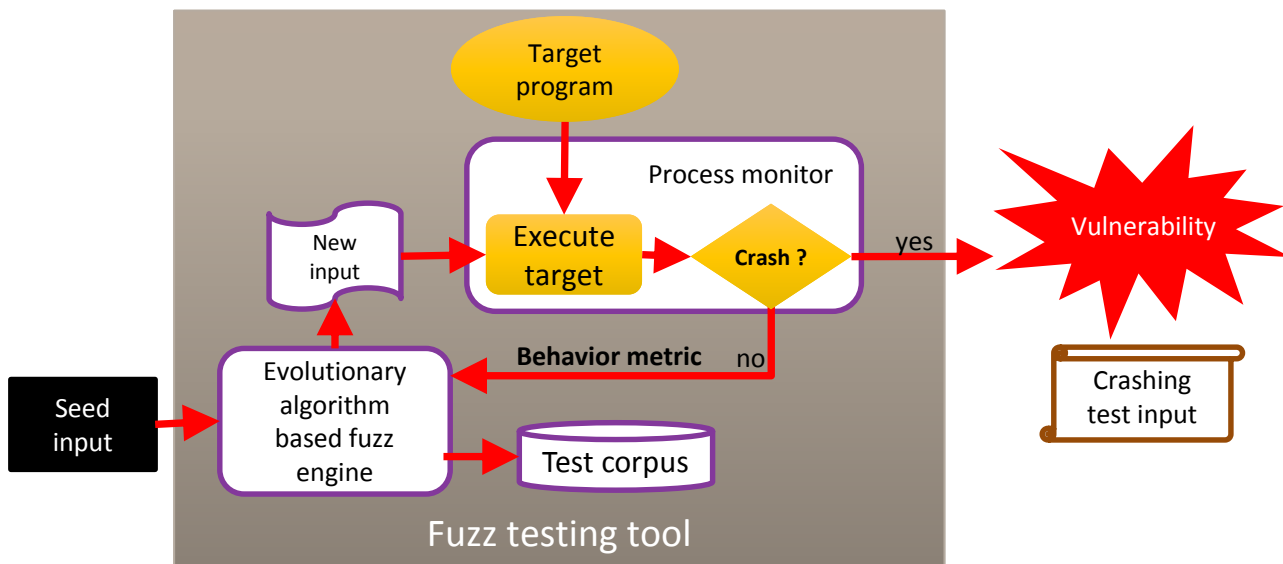
Test-Comp 2019 - TOOLympics at TACAS 2019



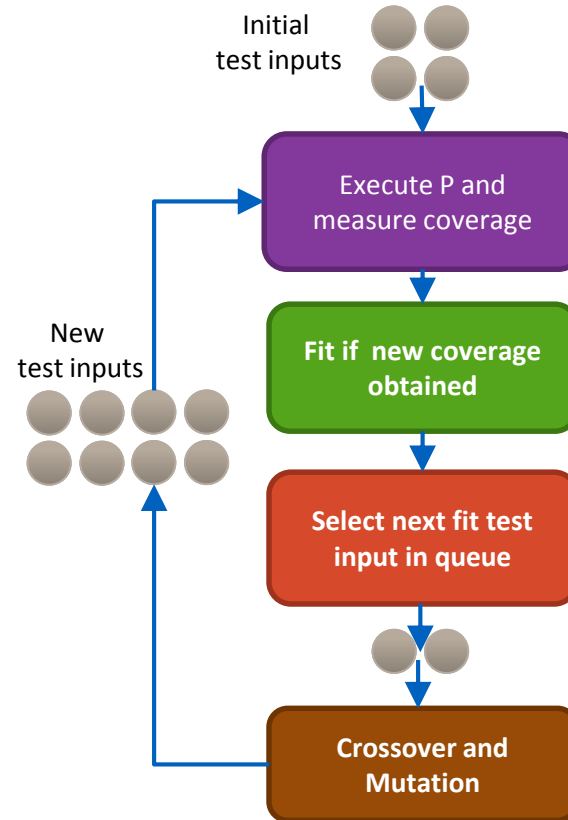
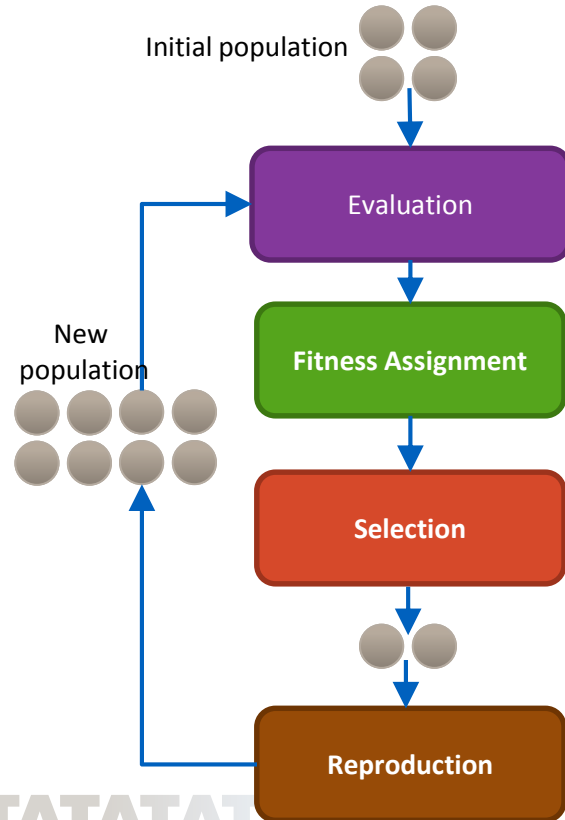
6-Apr-2019

Grey-box fuzzing

- Observe the *behaviors* exhibited on a set of test runs using a light weight instrumentation.
- Use this information to discover new test inputs that might exhibit new behaviors.
- Example : AFL is industrial strength grey box fuzz testing tool.



Automated testing powered by Evolutionary algorithms

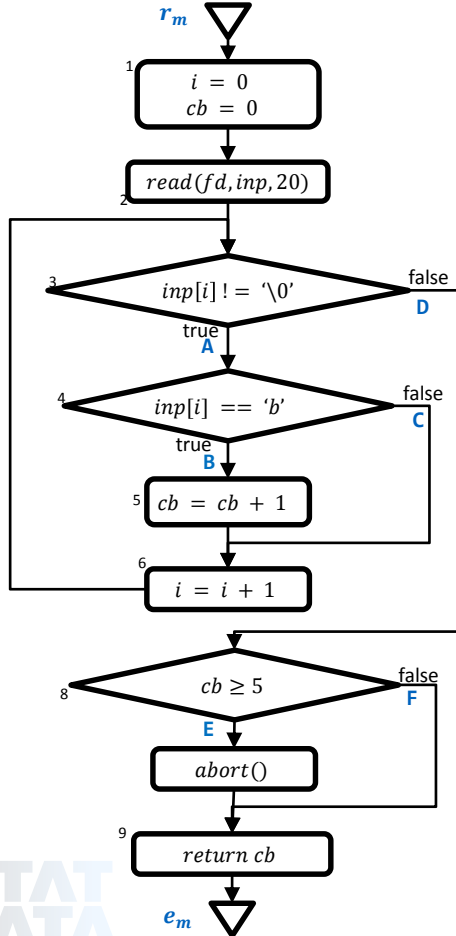


J.Wegner,A.Barsel et. al.,
Evolutionary test environment,
Information and software technology,
2001.

R.P. Pargs, M.J. Harrold, et. al.,
Test-data generation Using genetic
Algorithms, Journal of software testing
1999.

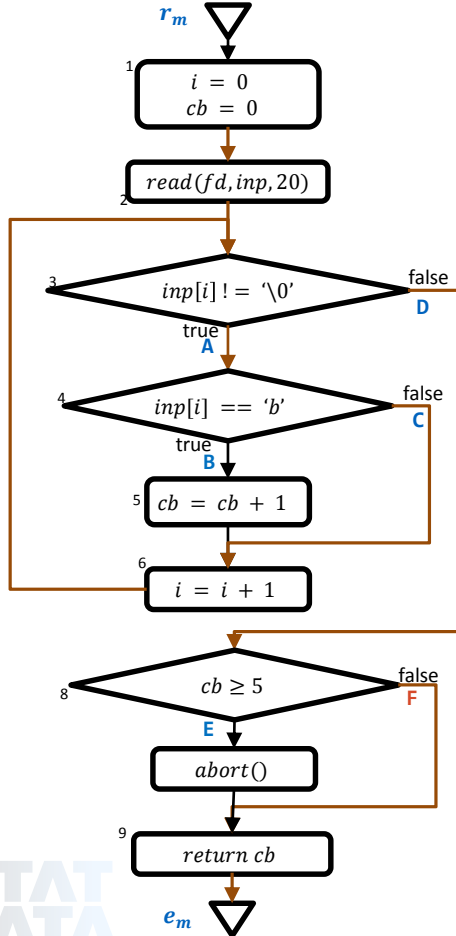
P. McMinn, Search based software testing,
Software testing, verification, and Reliability,
2004.

Grey-box fuzzing – Working example



Grey-box fuzzing – Working example

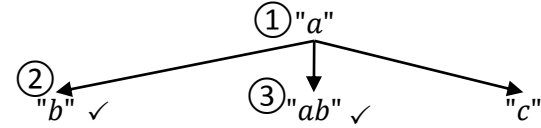
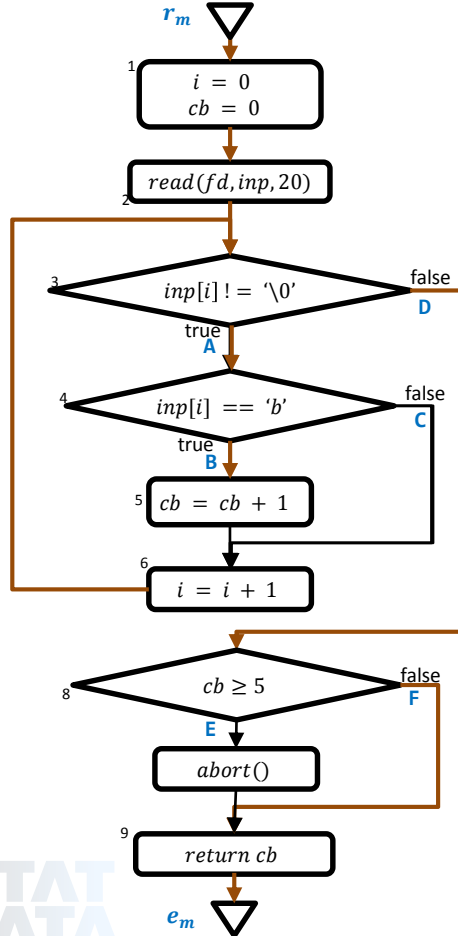
Initial input ① "a"



Evaluation

Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1

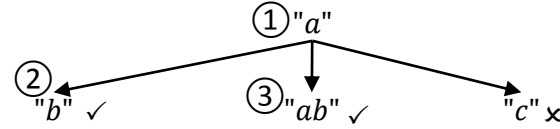
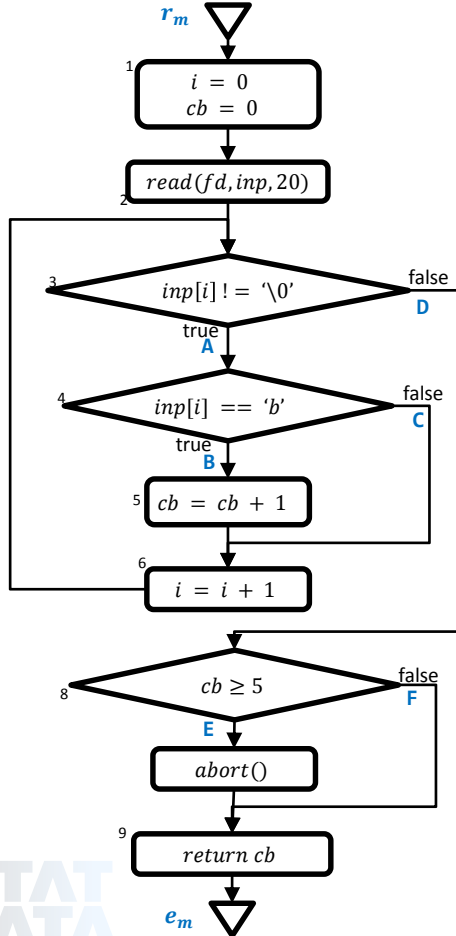
Grey-box fuzzing – Working example



Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
	"c"		1				1		1

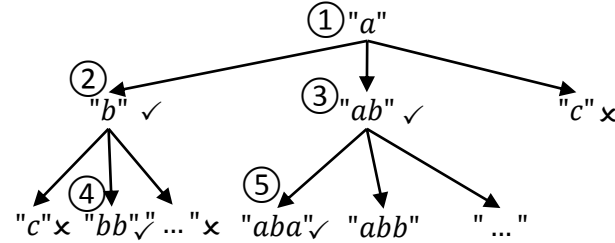
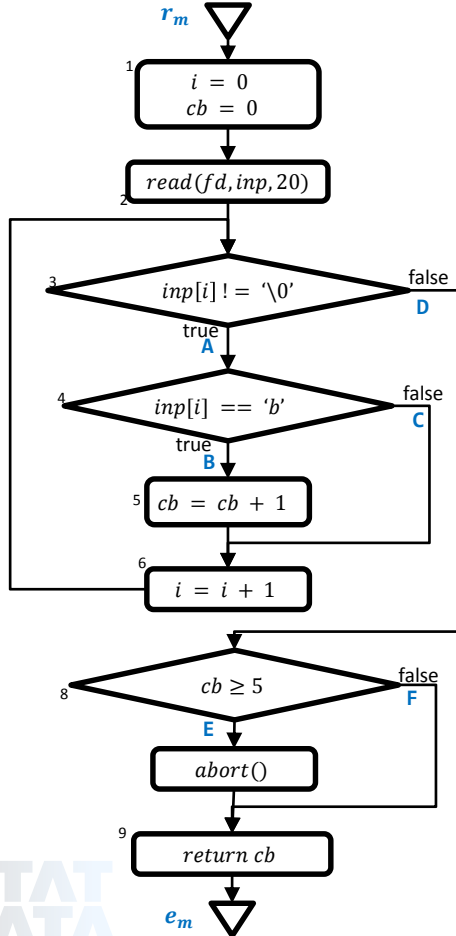
Fitness
check

Grey-box fuzzing – Working example



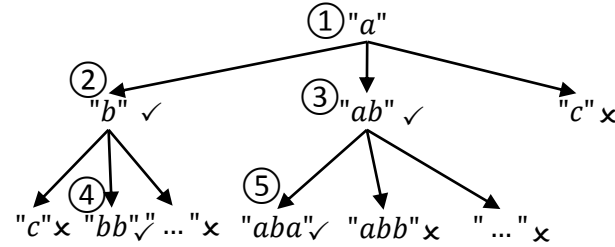
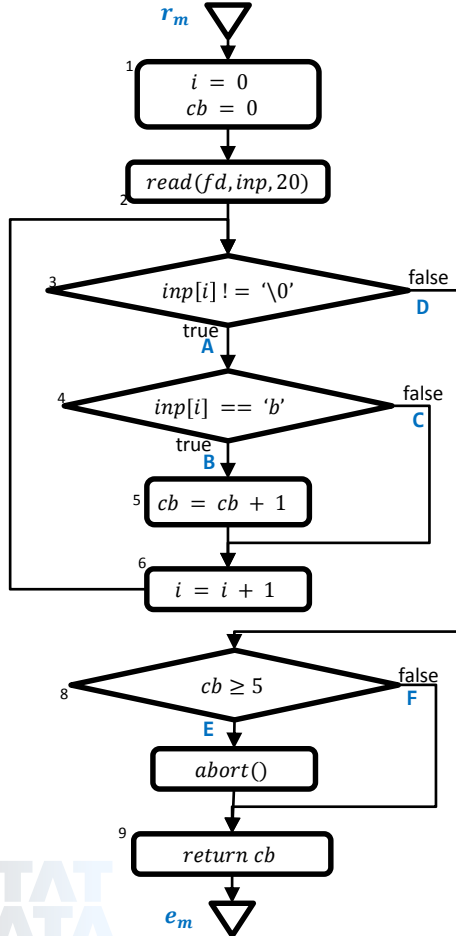
Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
	"c"		1				1		1

Grey-box fuzzing – Working example



Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
4	"bb"	2		1		1			1
5	"aba"	1	2	1	1		1		1
	"abb"	2	1	1	1	1			1

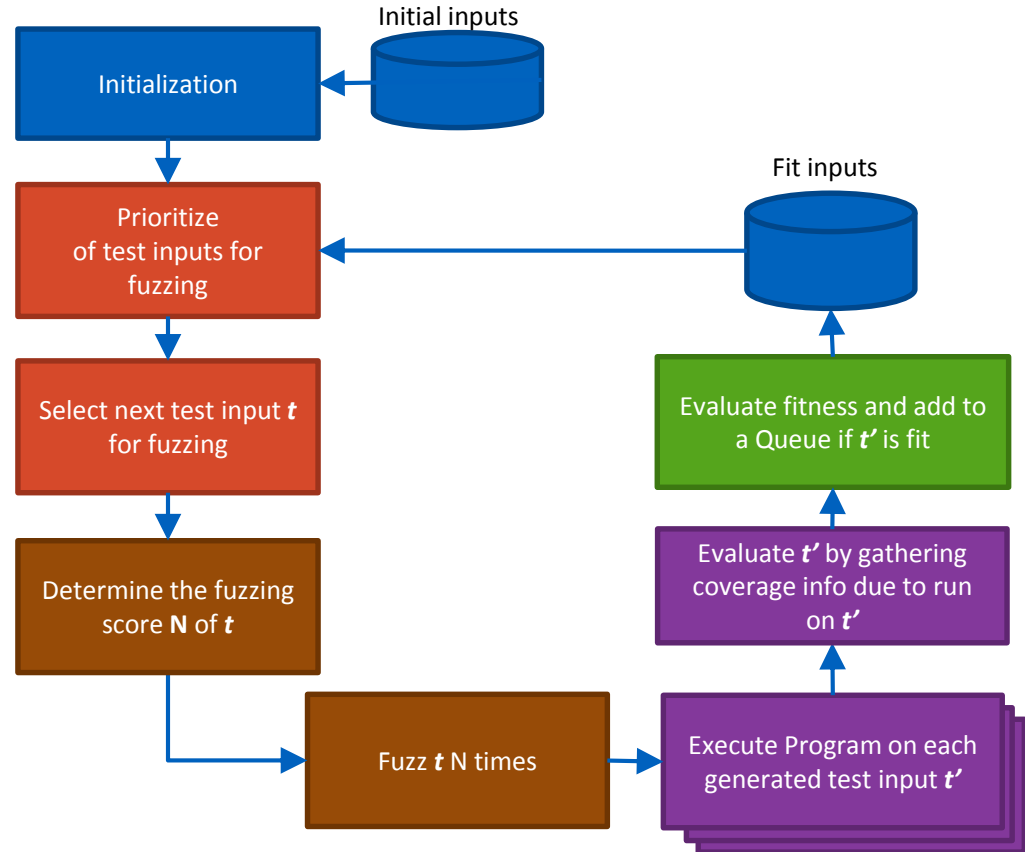
Grey-box fuzzing – Working example



Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
4	"bb"	2	1	1		1			1
5	"aba"	1	2	1	1		1		1
	"abb"	2	1	1	1	1			1

AFL – Gray Box fuzzer

- Simple to install and use.
- Several in-built heuristics.
- Found security vulnerabilities in several software libraries and critical applications.
- No dependency on program structure and structure of input.



AFL - Issues

- Insensitive to Program structure. [Vuzzer]
 - The metric used for evaluation is the branch pair visit counts.
- Insensitive to input structure.[SGF]
- Sensitive to input seeds. [skyfire]
- Sensitive to amount instrumentation. [instrim]
- Not directed towards error. [AFLGo]
- Inability to explore deeper paths. [AFL-fast][Fairfuzz][Driller][safI]
- Fuzzing is random and not adoptive. [Learn & Fuzz]

[AFLGo,AFL-Fast,SGF] : Marcel Bohme, Van-Thuan, et. al., CC17,CC16,arxiv 2019,

[Driller]N.Stephens et.al. Driller: Augmenting fuzzing with Symbolic execution. NDSS 2016.

[Fairfuzz] : C.Lemieux, K.Sen, FairFuzz: Targetting rare braches, ASE 2018.

[instrim]: Chin-Chia Hsu et.al., Instrim: Light weight instrumentation for CGF, BAR 2018.

[Learn&Fuzz]: P. Godefroid, R.Singh Learn & Fuzz, Machine Learning for Input fuzzing, ASE 2017

[Skyfire]: J.Wang. et.al., Data driven seed generation for fuzzing, S&P, 2017

[Vuzzer]: Sanjay Rawat et. Al., Application aware evolutionary fuzzing, NDSS 2017

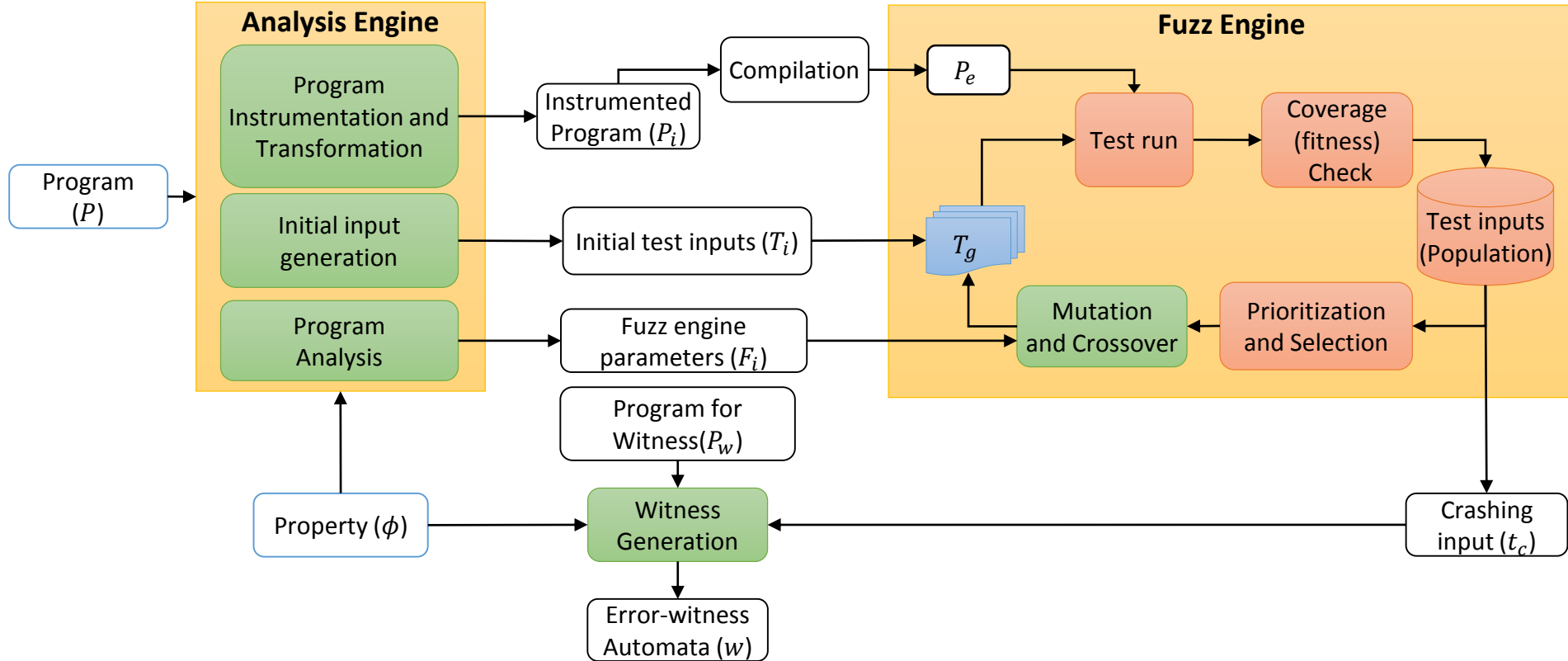
....and we encountered few more issues

- Unbounded programs :
 - Programs terminates only when the execution reaches the error location.
 - Reads input continuously as a stream.
- Restrictive range of inputs
 - Programs input is drawn from a small bounded range.
 - Programs that process event chains (aka ECA).
- Competition specific issues :
 - Limited time budget : Fuzzing approach is requires a longer exploration time for deeper bugs; often days and weeks. Budgeted time for each tasks in the competition is 900s.
 - Benchmark diversity : Large diversity in program's structure their constraints on input space, and input size.
 - Creating binaries : Benchmarks are targeted towards the static verification. Not for runtime verification. There are benchmarks with missing function definitions.
 - Execution issues : Large size memory allocations, kernel mode execution etc.

VeriFuzz - approach

- VeriFuzz is designed to address few of these issues.
- Emphasis is on speed.
- Proposed approaches in literature are either heavy or require large changes to AFL.
- Focus is on light weight instrumentation, analysis and transformation.
- Addressed the following issues.
 1. Reduced instrumentation overhead.
 2. Novel seed generation for programs with complex input validations.
 3. Bounding the unbounded programs.
 4. Restricted range of inputs.
- Algorithmic selection of the techniques (specific to competition).

VeriFuzz Architecture



AFL is the backbone of VeriFuzz

Complex validations on input

```
/1/ int main(){  
/2/  short b , c , d = 0,k ;  
/3/  b = getShort(); //Read input 1  
/4/  c = getShort(); //Read input 2  
/5/  k = getShort(); //Read input 3  
/6/  if (b > 25000 && c == 30000){  
/7/      while(d++ < k){  
/8/          if(d % 3 != 0){  
/9/              b++ ;c-- ;  
/10/         }else{  
/11/             b-- ;c++ ;  
/12/         }  
/13/         assert(b != c);  
/14/     }  
/15/ }  
/16/ return 0 ;  
/17/ }
```

Probability of generating input values such that this condition evaluates to *true* is
 $7768/(2^{32}) = 0.000002$

AFL needs to generate on average 500,000 inputs for the execution to enter this branch.
Observation : Large fuzzing time is wasted to generate a test input that can satisfy complex input constraints.

(A solution to fail the assertion at /13/ is b = 29,998,c=30,000, k=1)

TATATATAT
TATATATAT

One **idea** is that a generated input using an initial input that satisfies the condition at line no /6/ is *likely* to reach the error location *quickly*.

How to obtain an initial input that can satisfy complex input constraints ?

How to obtain an initial input that can satisfy complex input constraints ?

Complex validations on input

```

/1/  int main(){
/2/    short b , c , d = 0,k ;
/3/    b = getShort(); //Read input 1
/4/    c = getShort(); //Read input 2
/5/    k = getShort(); //Read input 3
/6/    if (b > 25000 && c == 30000){
/7/        assume(d++ < k);
/8/        if(d % 3 != 0){
/9/            b++ ;c-- ;
/10/        }else{
/11/            b-- ;c++ ;
/12/        }
/13/        assert(b != c);
/14/    }
/15/    assert(false);
/16/    return 0 ;
/17/ }

```

Construct a *loop free* program that preserves as many input constraints as possible.

Use symbolic execution along a path that contains complex condition in the transformed program. Generate a test input that by solving path constraints.

Note that the intent is **not** to preserve original program behavior, but to make the program loop free.

Path constraint along /6-/7-/8-/9-/13-/16/
 $b1 > 25000 \wedge c1 == 30000 \wedge 0 < k1 \wedge 1\%3 \neq 0 \wedge b1+1 \neq c1-1$

Unbounded programs

```

/1/  short g1, g2, t;
/2/  int main(){
/3/      g1 = getShort() ;
/4/      g2 = getShort() ;
/5/      while(1){
/6/          g1++;
/7/          while(1){
/8/              g2++;
/9/              t = getShort(),
/10/             if(t == 0) break;
/11/             assert(g1 != g2);
/12/         }
/13/     }
/14/     return 0 ;
/15/ }

```

No loop bounds. Program terminates on the assertion failure.

Keep reading input in a loop

This program is likely cause fuzzer hang.

(A solution to fail the assertion at /11/ is $g1=0, g1=0, t=1$)

Unbounded programs

```
/1/  short g1, g2, t;  
/2/  int main(){  
/3/    g1 = getShort() ;  
/4/    g2 = getShort() ;  
/5/    while(1){  
/6/        g1++;  
/7/        while(1){  
/8/            g2++;  
/9/            t = getShort();  
/10/         if(t == 0) break;  
/11/         assert(g1 != g2);  
/12/     }  
/13/ }  
/14/ return 0 ;  
/15/ }
```

Problem : How to give fuzzer a chance to fuzz the input?

Unbounded programs

```
/1/ short g1, g2, t, 11,12, 1BND = 3000;  
/2/ int main(){  
/3/   g1 = getShort() ;  
/4/   g2 = getShort() ;  
/5/   while(11 < 1BND){ //while(1)  
/6/       g1++;  
/7/       while(12 < 1BND){ //while(1)  
/8/           g2++;  
/9/           t = getShort();  
/10/          if(t == 0) break;  
/11/          assert(g1 != g2);  
/12/      }  
/13/ }  
/14/ return 0 ;  
/15/ }
```

Bound the loops to a small value.

Dynamically increase 1BND over fuzzing period


See if AFL can generate test input that can violate the
assertion

Restricted range of inputs

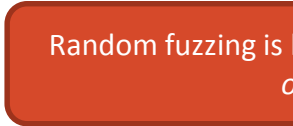
```

/1/ short g1=0,g2=15,g3=1;
/2/ int main(){
/3/     int i1;
/4/     while(1){
/5/         i1 = getShort() ;
/6/         if((i1 !=1) && (i1 !=2) && (i1 != 3) return -2;
/7/         if( g1 == 0 && g2 != 13 && i1 == 3) {g1 = 1; g3 = 3}
/8/         if( g1 != 2 && g2 > 2 && i1 == 1) {g2 = 3;}
/9/         .....
/10/        assert(g1 == 4 && g2 > 5 && g3 !=1)
/11/    }
/12/    return 0 ;
/13/ }

```



Value



Random fuzzing is on

Valid input range is (1,2,3)

Random fuzzing is likely to produce an input sequence with *out of range* numbers...

Restricted range of inputs

```
/1/ short g1=0,g2=15,g3=1;
/2/ int main(){
/3/   int i1;
/4/   while(1){ {<i1,T><g1,[10]><g2,[15]><g3,[1]>}
/5/       i1 = getShort() ;
/6/       if((i1 !=1) && (i1 !=2) && (i1 != 3) return -2;
/7/       if( g1 == 0 && g2 != 13 && i1 == 3) {g1 = 1; g3 = 3}
/8/       if( g1 != 2 && g2 > 2 && i1 == 1) {g2 = 3;}
/9/       .....
/10/      assert(g1 == 4 && g2 > 5 && g3 !=1)
/11/  }
/12/  return 0 ;
/13/ }
```

How to determine that the valid input range is (1,2,3) ?

How to fuzz the input such that the generated input is a random sequence formed from 1,2,3 ?

{<i1,[1,2,3]><g1,[10]><g2,[15]><g3,[1]>}

{<i1,[1,2,3]>...}

Perform *k*-interval analysis[pranalysis] and determine ranges of input variables at the error location.

Change the mutation engine to randomly choose the values within these ranges.

Other optimizations

- Optimized instrumentation by reducing number of instrumentation points.
- Algorithmic selection of techniques

Test-comp 2019

<https://test-comp.sosy-lab.org/2019/results/results-verified/>

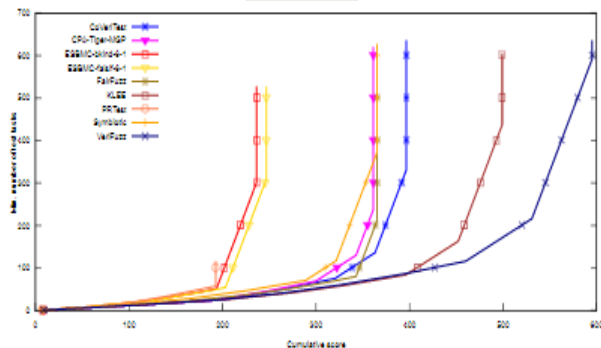
The background color is **gold for the winner**, **silver for the second**, and **bronze for the third**.

Ranking by Category (with Score-Based Quantile Plots)

What you can learn from a score-based quantile plot and how to interpret it, is described in the [competition report](#) on pages 12 and 13.

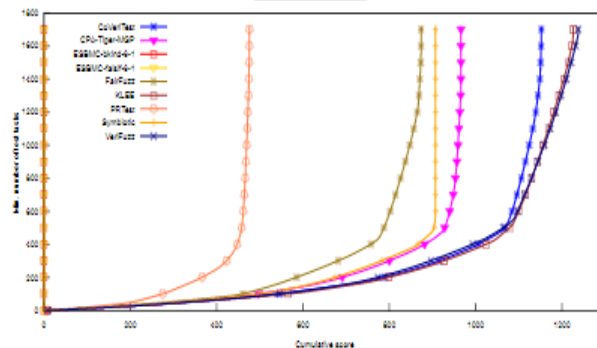
Cover-Error

1. VeriFuzz
2. KLEE
3. CoVeriTest



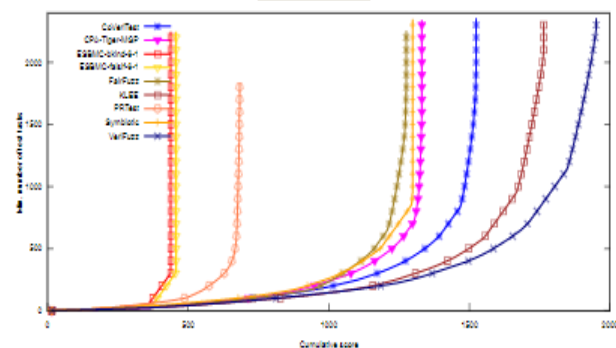
Cover-Branches

1. VeriFuzz
2. KLEE
3. CoVeriTest

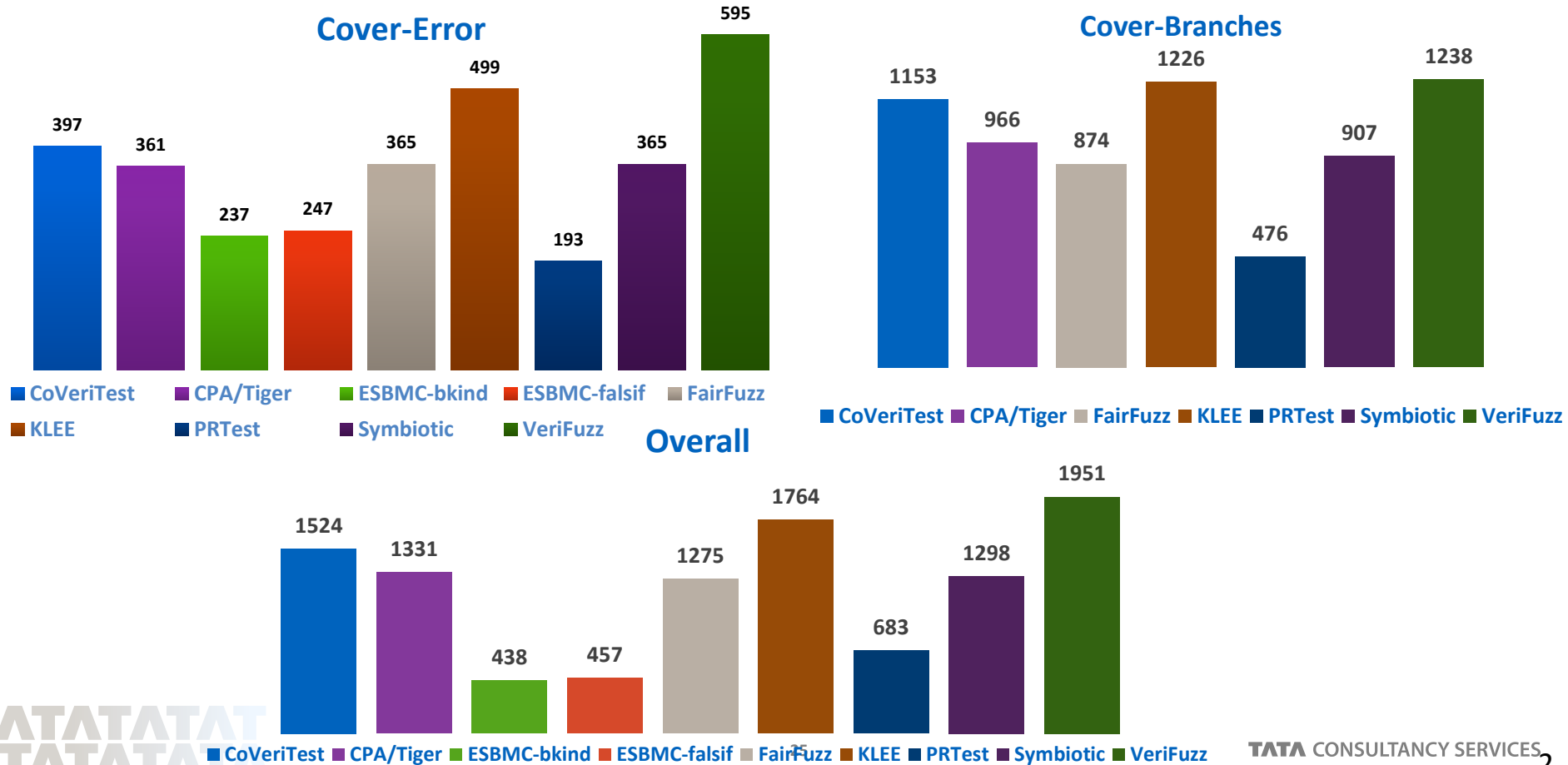


Overall

1. VeriFuzz
2. KLEE
3. CoVeriTest

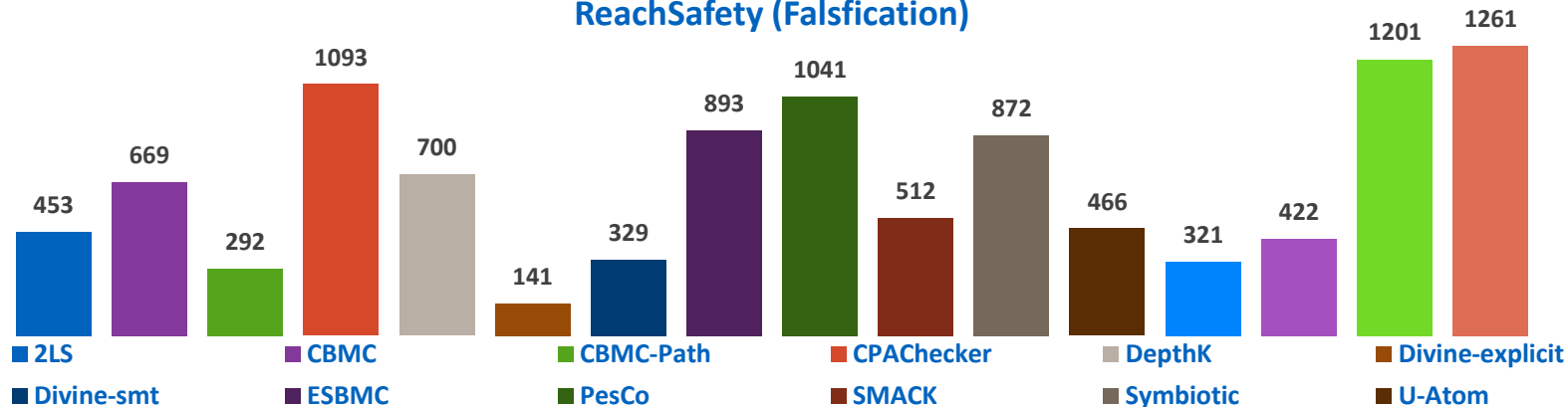


Test-comp 2019

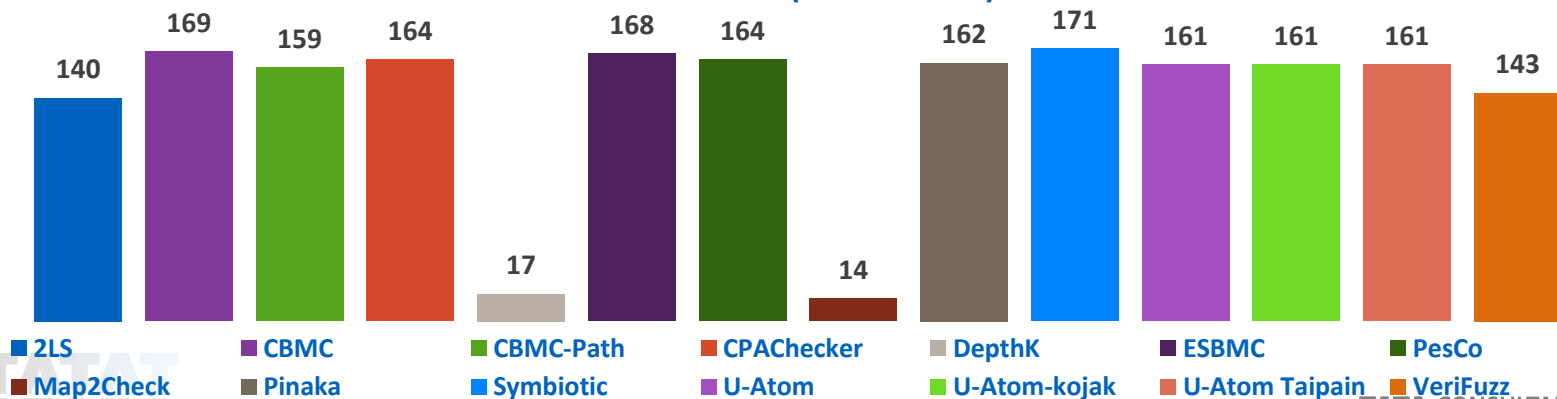


SV-COMP 2019

ReachSafety (Falsification)



NoOverflow(Falsification)



Thank you

