SOFTWARE RELIABILITY GROUP

Imperial College London

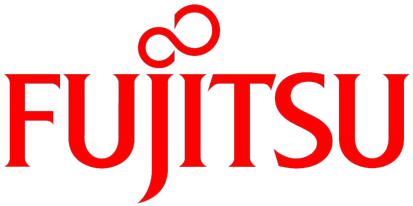
# Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing

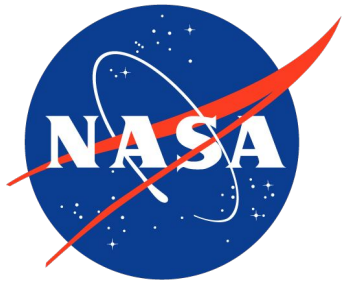
Timotej Kapus, Cristian Cadar
*Department of Computing*
*Imperial College London*

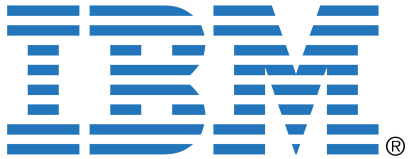
```
if (x > 2294967295) {
  assert(false);
}
printf("x: %u\n", x);
```

# Symbolic execution

- Used in industry:
  - IntelliTest
  - SAGE
  - KLOVER
  - SPF
  - Apollo
- Active research field

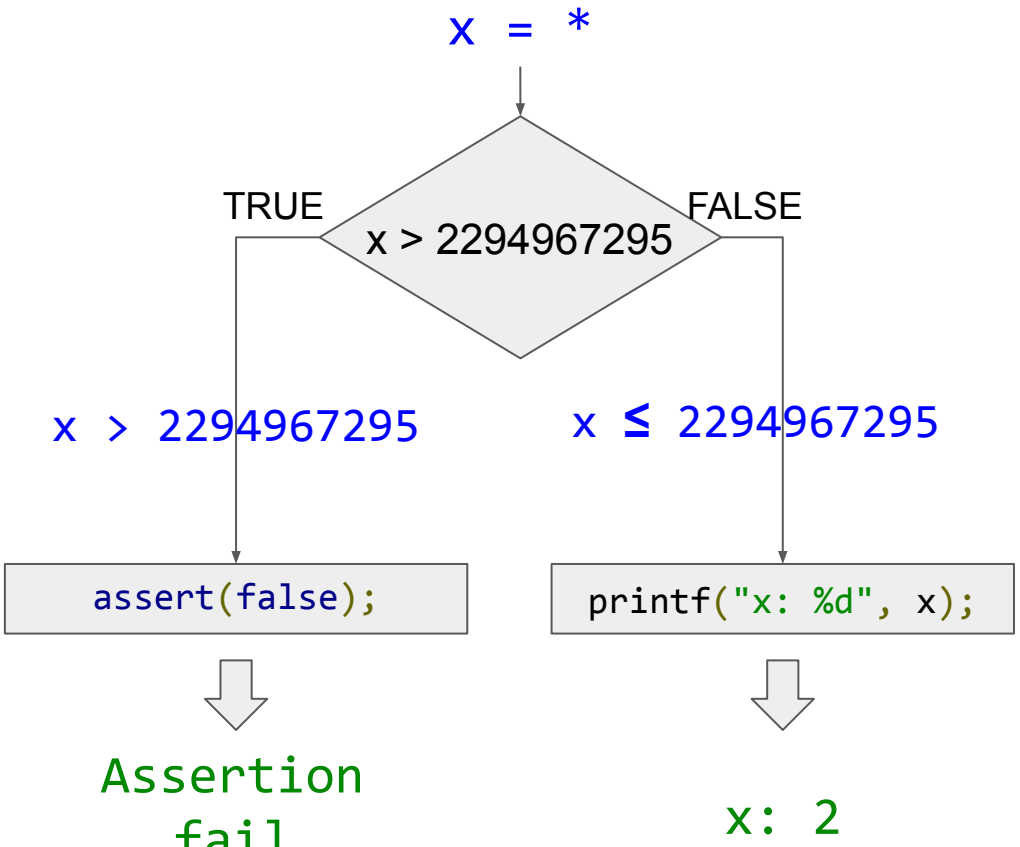



# Symbolic execution

```
1 unsigned int x = 5;
2 int main() {
3   if (x > 2294967295) {
4     assert(false);
5   }
6   printf("x: %u\n",x);
7 }
```

# Symbolic execution




```
1  unsigned int x = 5;
2  int main() {
3    make_symbolic(&x);
4    if (x > 2294967295) {
5      assert(false);
6    }
7    printf("x: %u\n",x);
8  }
```

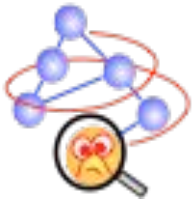
# Symbolic executors

- Many available open source
- Complex pieces of software
  - Accurate interpreter or precise instrumentation
  - Accurate constraint solving
  - Constraint gathering
  - Scheduling
  - Effective optimizations such as caching, fast solving, etc.

KLEE

CREST

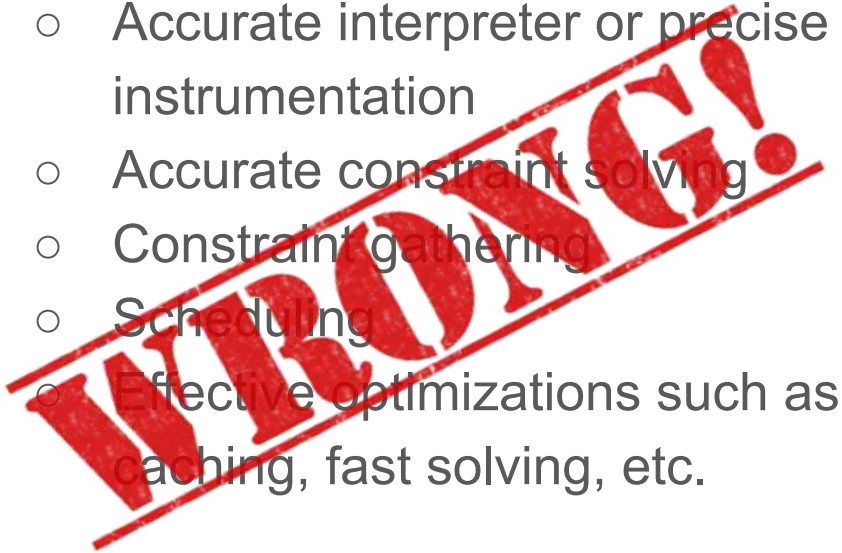
**Angr**



*JPF*

# Symbolic executors

- Many available open source
- Complex pieces of software
  - Accurate interpreter or precise instrumentation
  - Accurate constraint solving
  - Constraint gathering
  - Scheduling
  - Effective optimizations such as caching, fast solving, etc.




KLEE

CREST

Angr

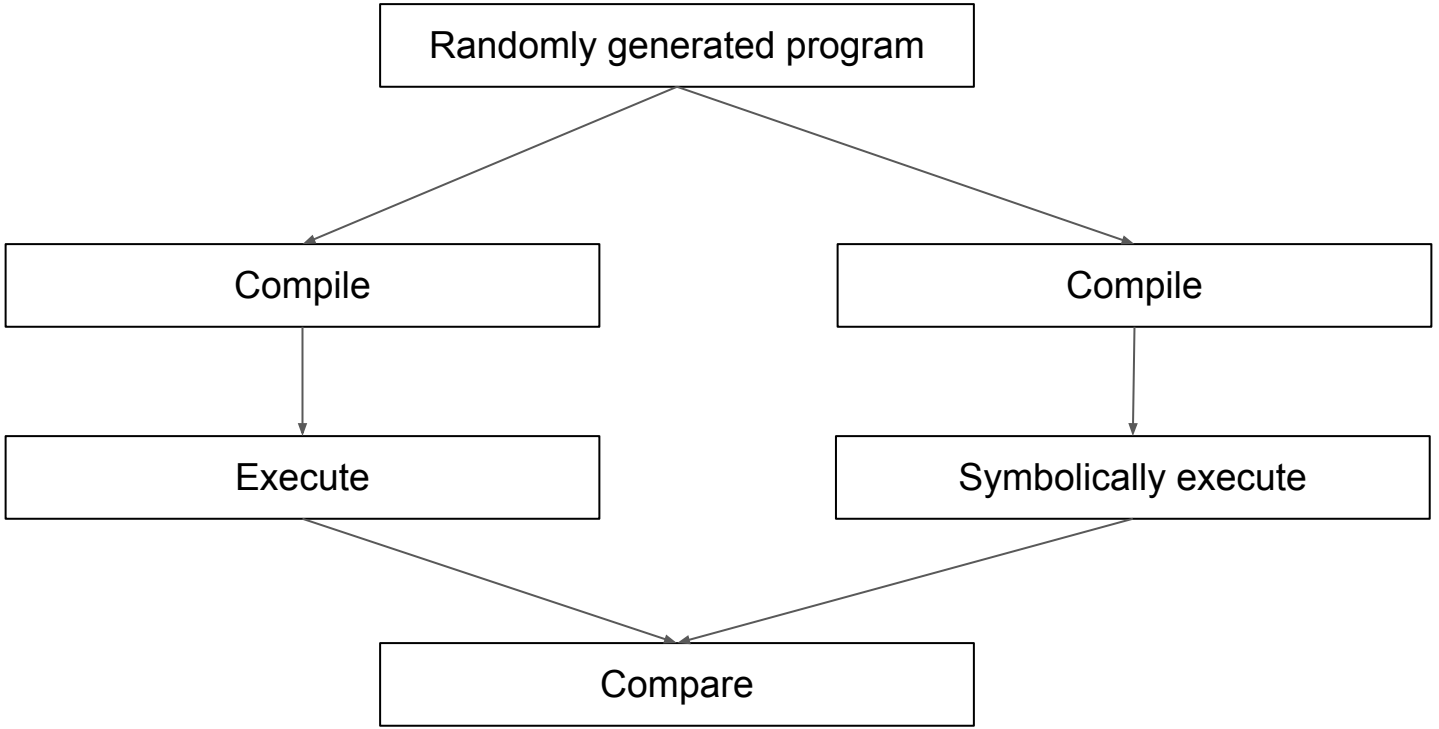
JPF

# Bugs in symbolic executors

- Particularly bad
- Lead to false sense of security
- Examples:
  - Missing a branch
  - Exploring spurious branches

```
1  unsigned int x = 5;
2  int main() {
3    make_symbolic(&x);
4    if(x > 2294967295) {
5      assert(false);
6    }
7    printf("x: %u\n",x);
8  }
```

# Differential testing of symbolic execution

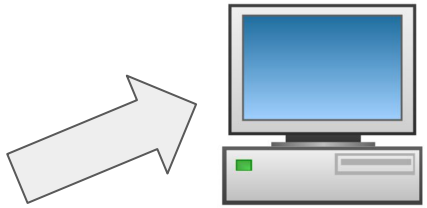


# Testing symbolic executors

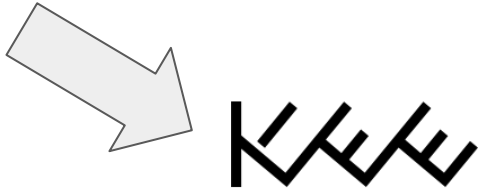
- Compare two executions (native/symbolic) in 3 different modes:
  - **Concrete** - tests interpretation/instrumentation
  - **Single Path** - tests constraint gathering and solving
  - **Multi Path** - tests scheduling, test case generation

# Concrete mode

```
1  unsigned int x = 5;
2  int main() {
3    if (x > 2294967295) {
4      assert(false);
5    }
6    printf("x: %u\n",x);
7  }
```



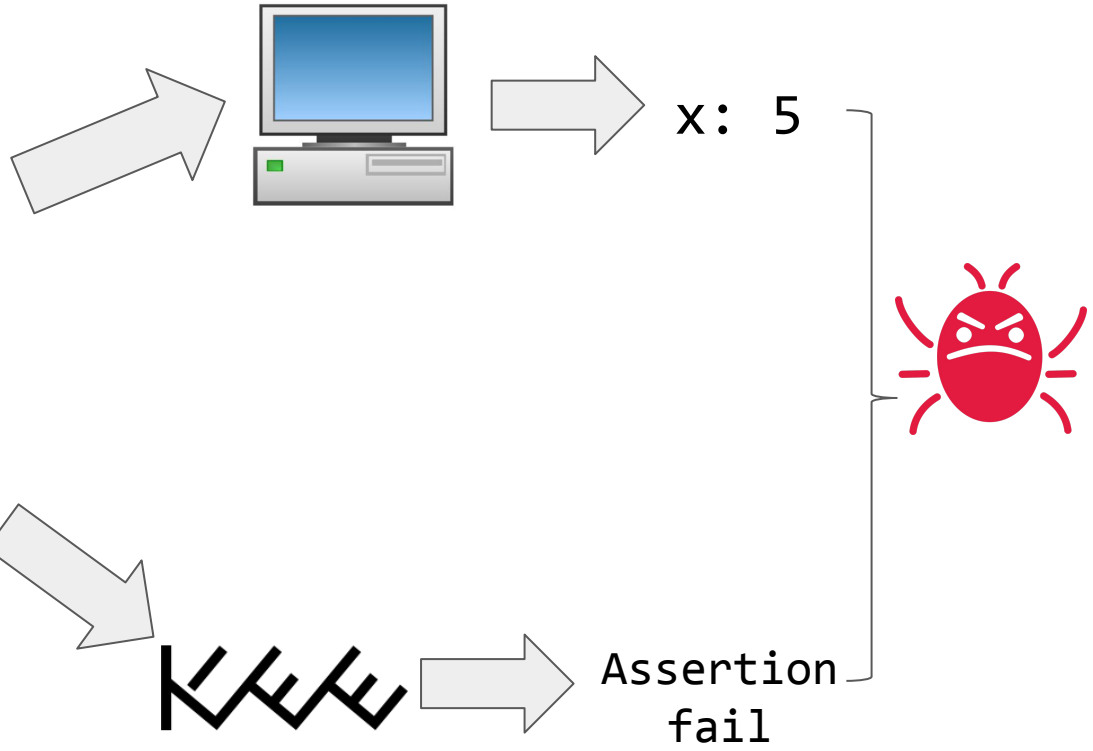
x: 5



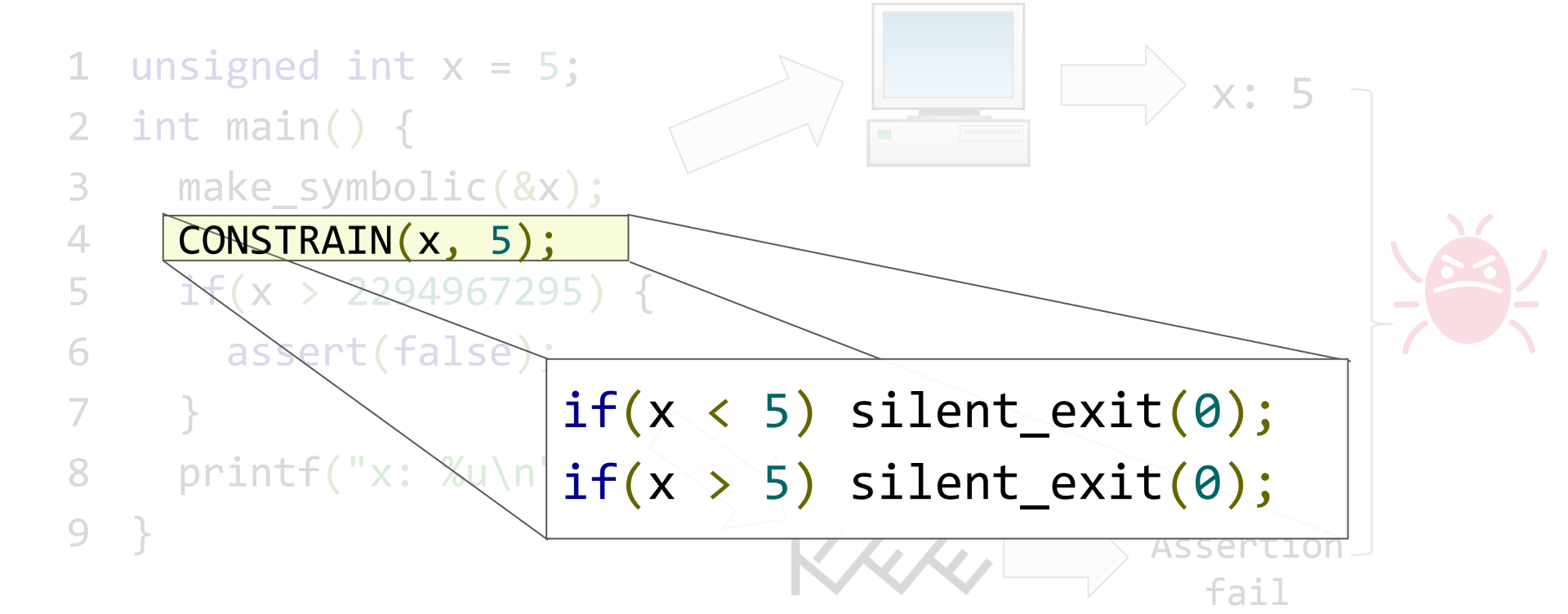
x: 343

# Single-Path mode

```
1  unsigned int x = 5;
2  int main() {
3    make_symbolic(&x);
4    CONSTRAIN(x, 5);
5    if(x > 2294967295) {
6      assert(false);
7    }
8    printf("x: %u\n",x);
9  }
```



# Single-Path mode: Constrainers



# Single-Path mode: Constrainers

```
1  unsigned int x = 5;
2  int main() {
3    make_symbolic(&x);
4    CONSTRAIN(x, 5);
5    if(x > 2294967295) {
6      assert(false);
7    }
8    printf("x: %u\n", x);
9  }
```



x: 5

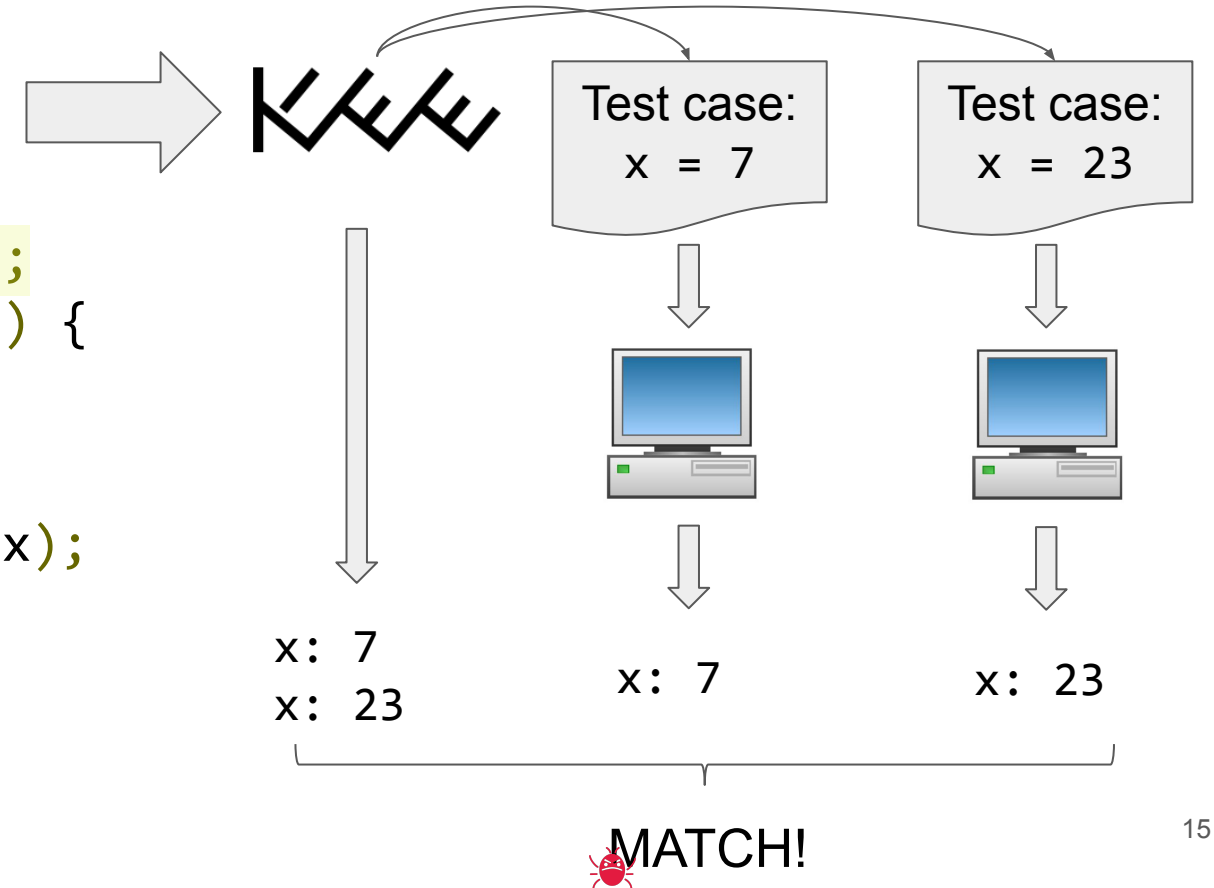


Assertion fail

```
if(x != 5) silent_exit(0);
```

# Multi-Path mode

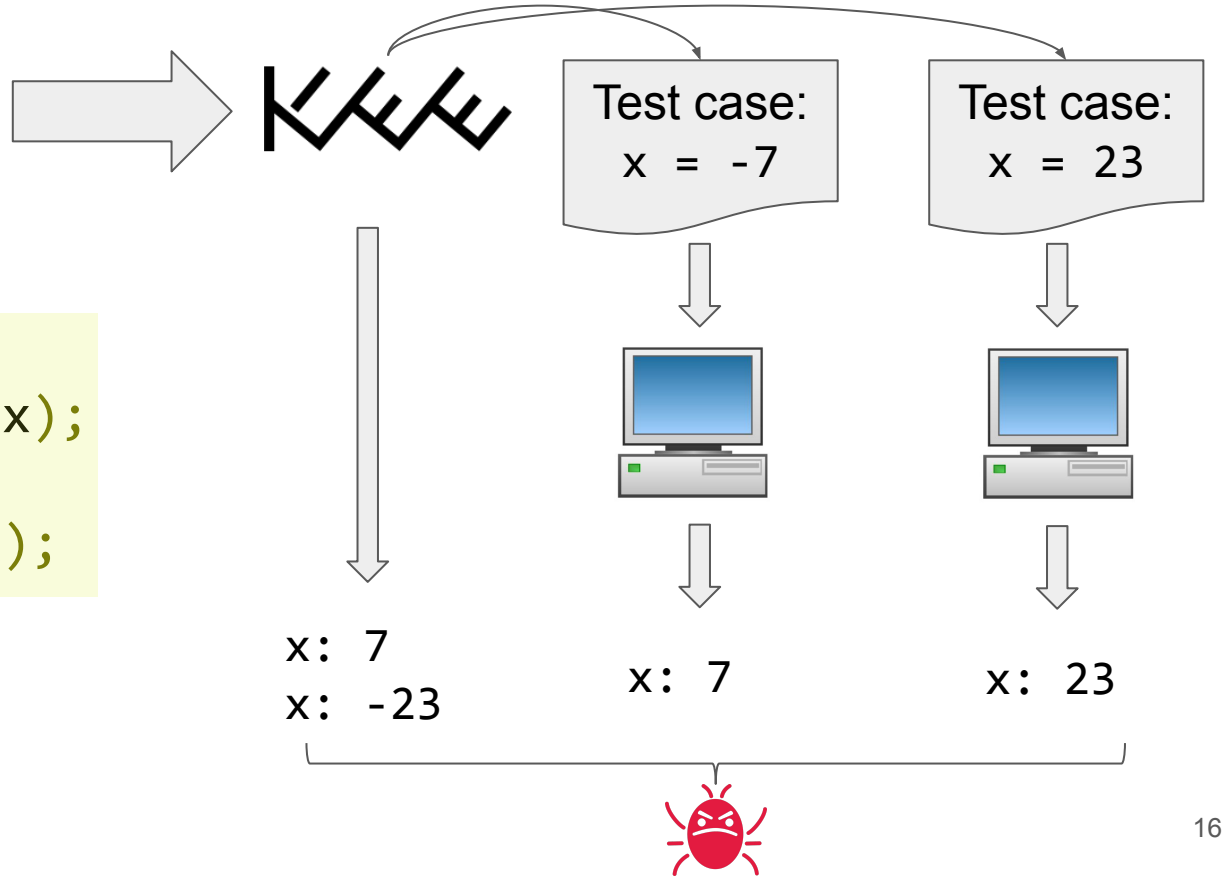
```
1  unsigned int x = 5;
2  int main() {
3    make_symbolic(&x);
4    if(x > 2294967295) {
5      assert(false);
6    }
7    printf("x: %u\n",x);
8  }
```



# Multi-Path mode

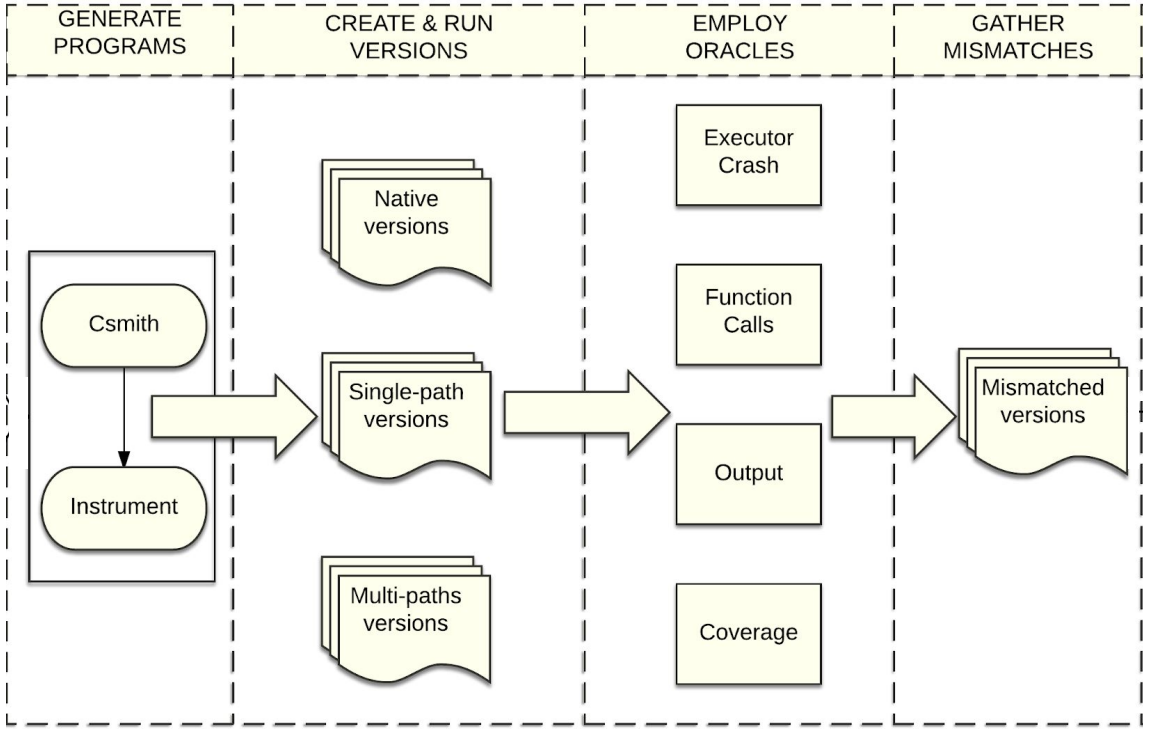
```
int x = 5;

void main() {
  make_symbolic(&x);
  if(x < 0)
    printf("x: %d", -x);
  else
    printf("x: %d", x);
}
```

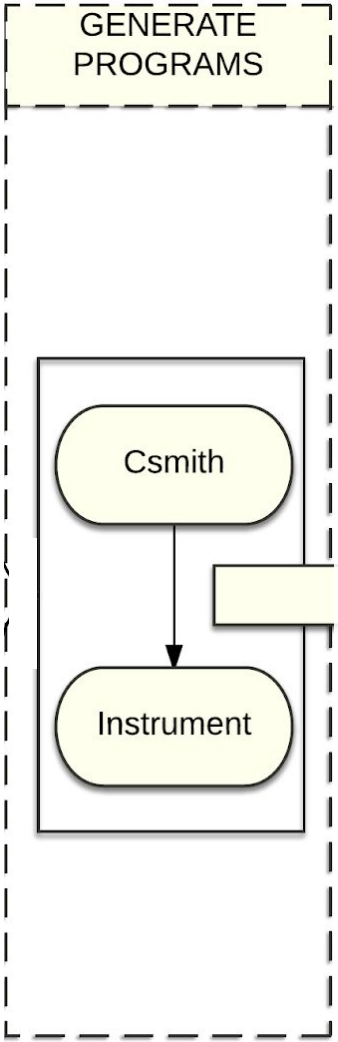


# Testing symbolic executors

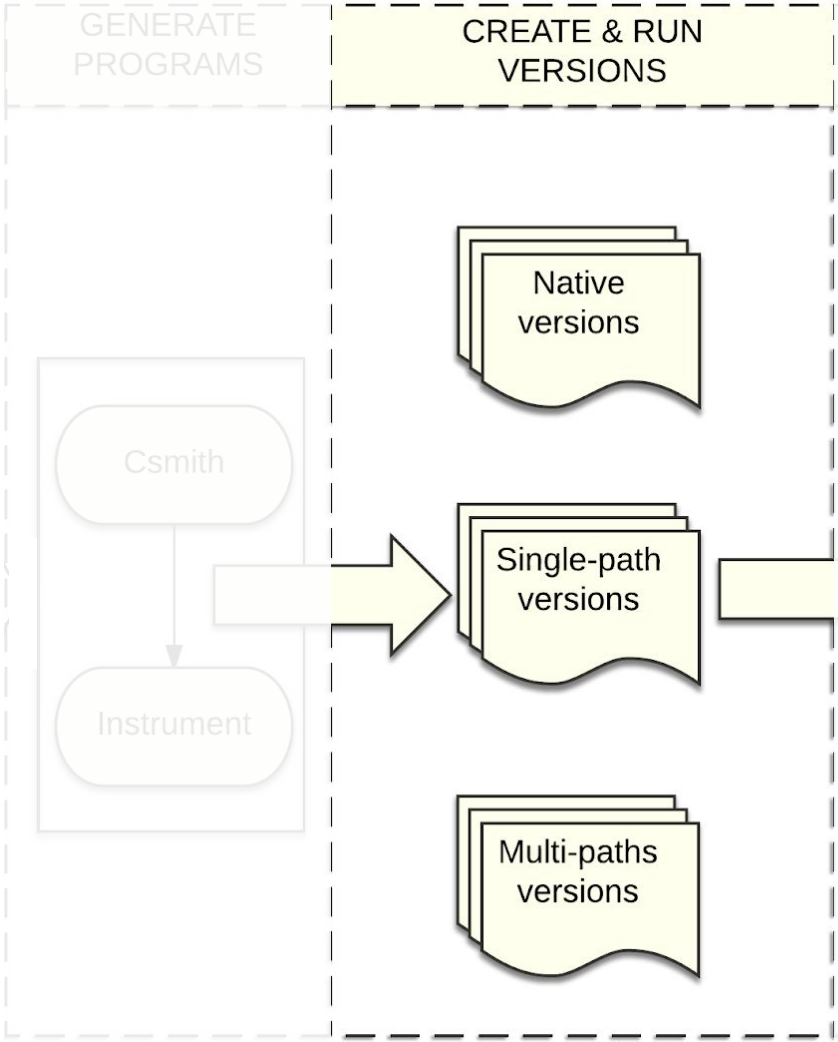
- Built a pipeline
- Run experiments in batches
- Avoid bugs found in previous batches



# Instrumentation supports




- Csmith
  - Random program generator
  - Found many bugs in compilers
  - Doesn’t generate programs with undefined behaviour
- Instrumentation supports:
  - Marking variables as symbolic
  - Oracles
  - Constraining

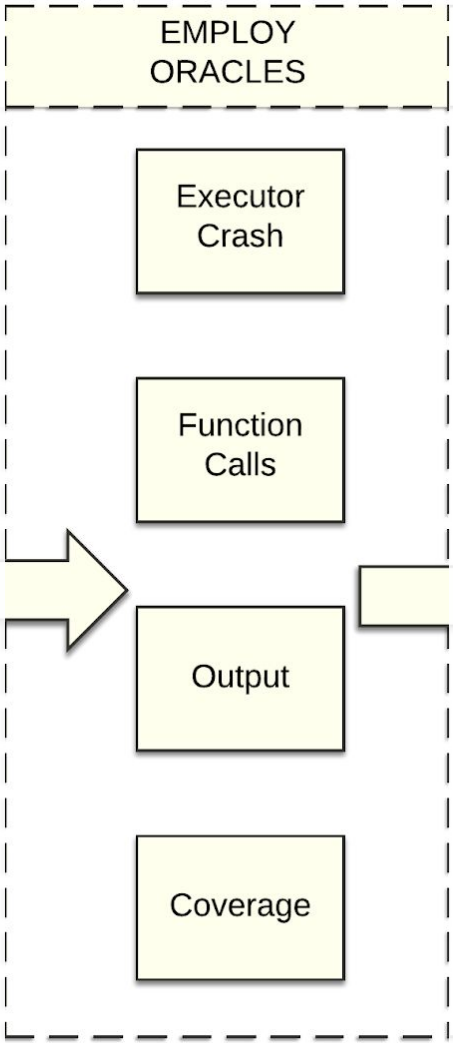



Versions correspond to mode:

- Concrete mode - native version
- Single path mode - single path version
- Multi path mode - multi path version

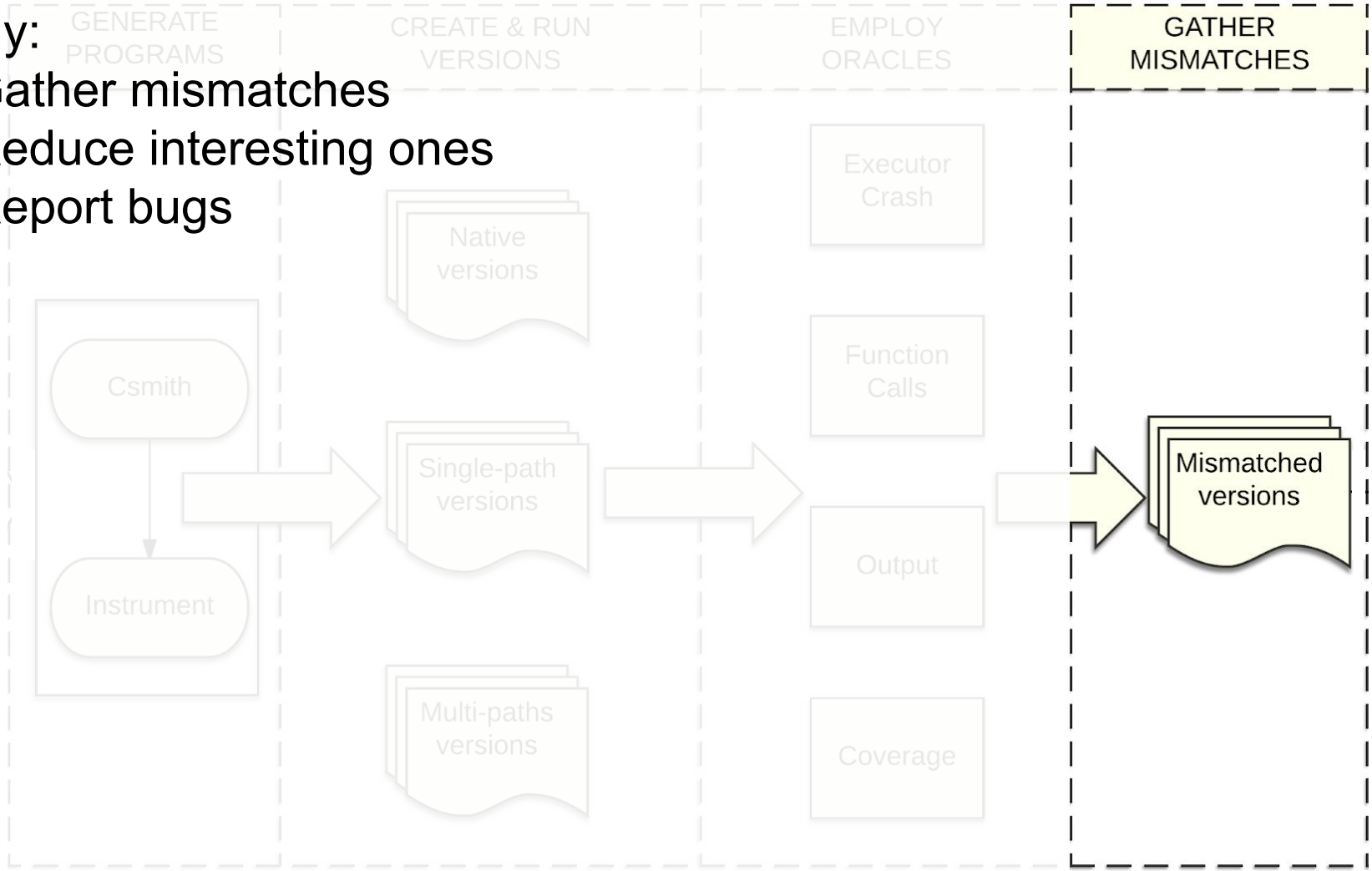
Oracles can check:

1. Executor doesn't crash
2. Function call chain
3. Output (values of all global variables) matches
4. Coverage achieved on the random program



Finally:

- Gather mismatches
- Reduce interesting ones
- Report bugs



# Case Studies

## KLEE


- Main case study
  - Familiarity
  - Flexibility
- Built on top of LLVM
- Keeps all paths in memory

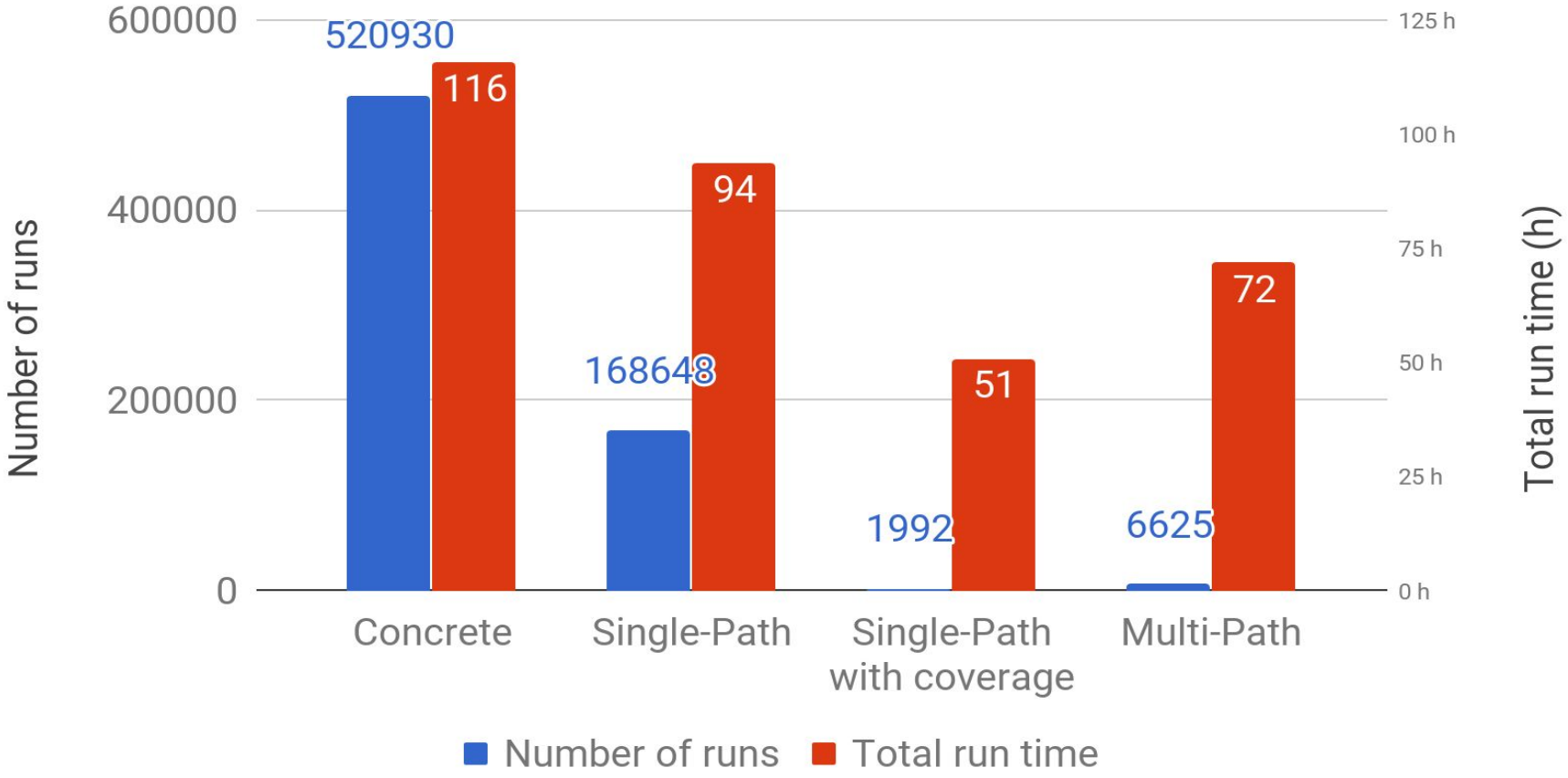
## CREST


- Concolic execution
- Instrumentation instead of interpretation
- Doesn’t generate test cases

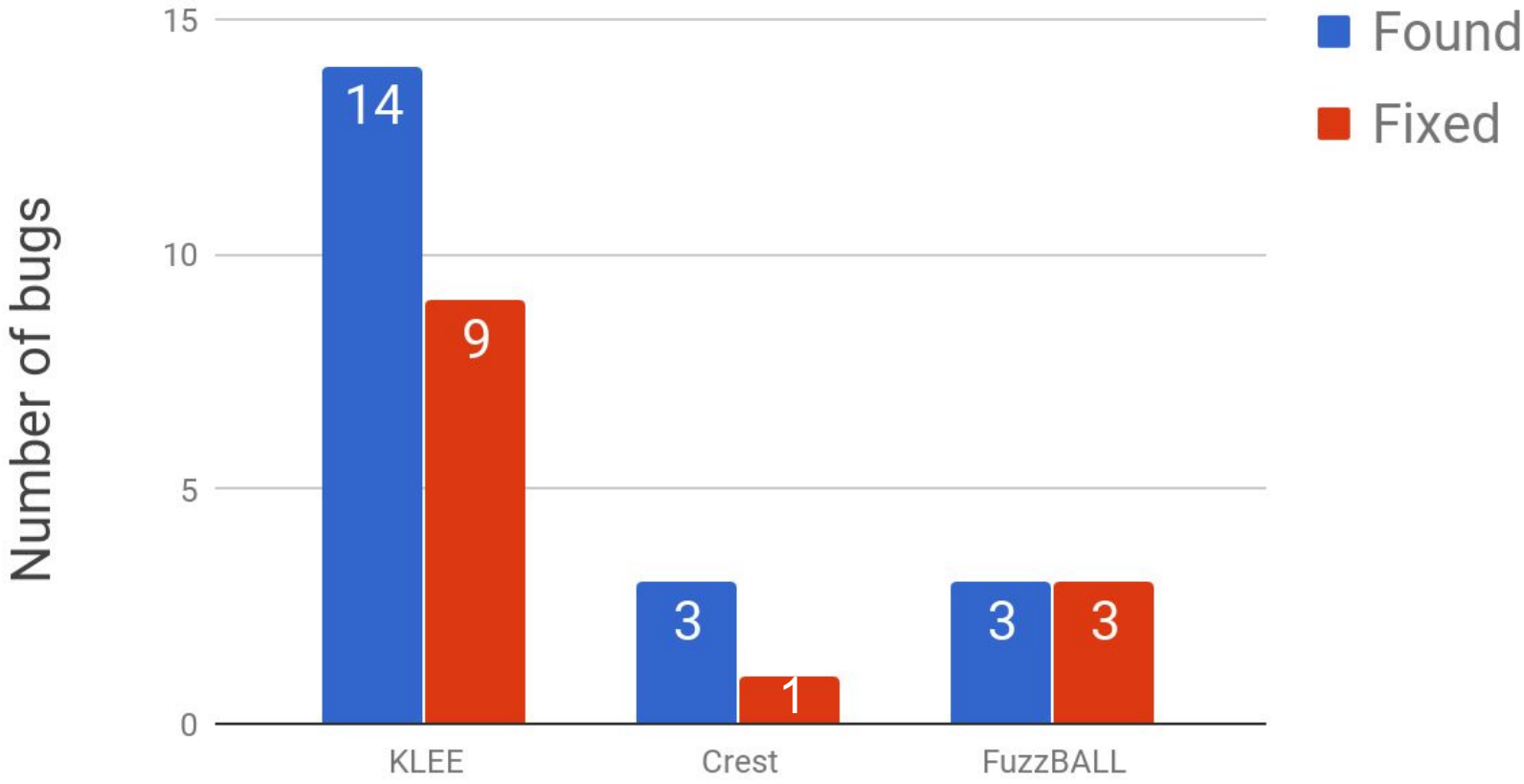
## FuzzBALL

- Binary level executor
- Doesn’t generate test cases

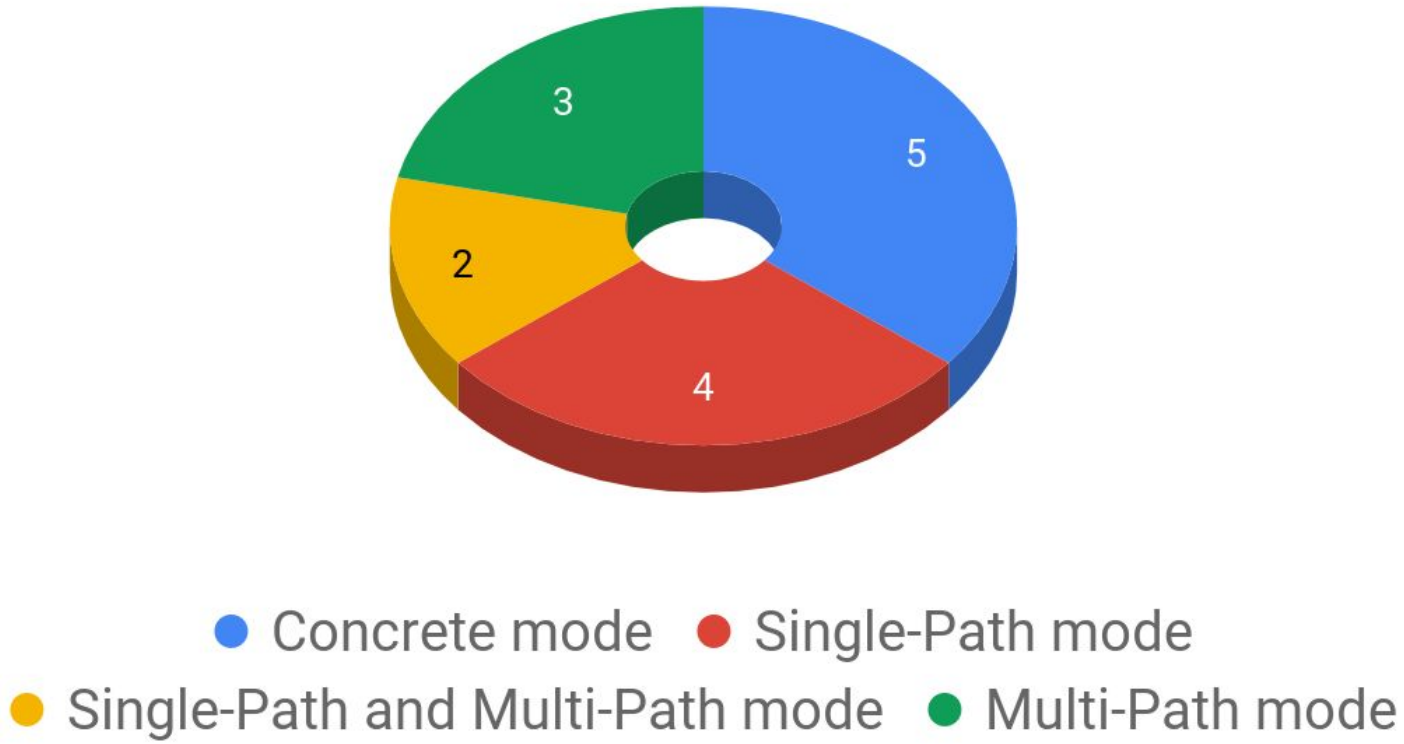
# KLEE summary of runs



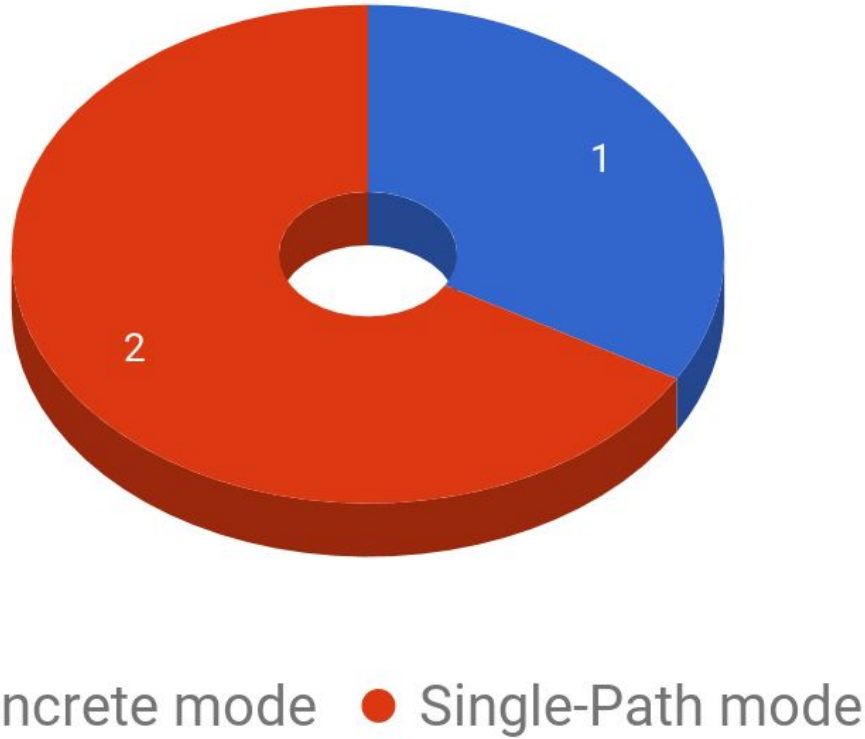
# Summary of bugs found



# KLEE bugs by mode

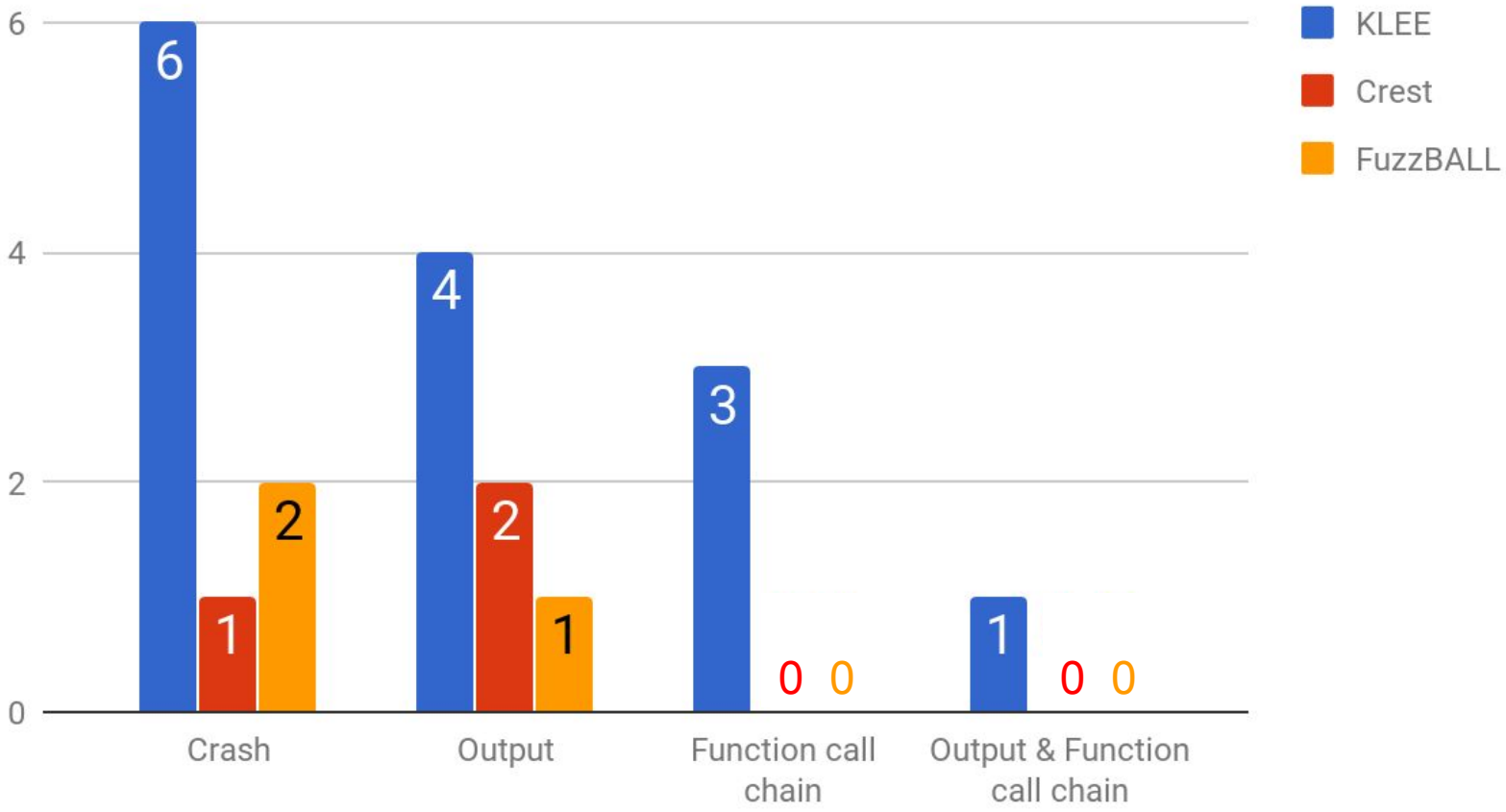


# Crest bugs by mode



# FuzzBALL bugs by mode

# Bugs found by oracle



# Example bug: *Crest*

```
1  unsigned int a;
2  int main() {
3    make_symbolic(&a);
4    if(a > 2294967295) {
5      assert(false);
6    }
7    printf("a: %d\n",a);
8  }
```

| Expected output | Actual output |
| --- | --- |
| a: 6<br>Assertion fail | a: 6<br>a: 23 |

# Example bug: *KLEE*

```
1  int g_10 = 0;
2  int main() {
3     make_symbolic(&g_10);
4     do {
5        printf("loop\n");
6        g_10 &= 2;
7     } while(!((3 ^ g_10) / 1));
8  }
```

| Expected output | Actual output |
| --- | --- |
| loop | loop<br>loop<br>loop<br>loop<br>loop<br>loop<br>loop<br>... |

# Example bug: *FuzzBALL*

```
1  unsigned int g_54 = 0;
2  unsigned int g_56 = 0;
3
4  void main ( void ) {
5      make_symbolic(&g_54);
6      CONSTRAIN(g_54, 0);
7      g_56 ^= 0 < g_54;
8      printf("g_56: %u\n", *(&g_56));
9  }
```

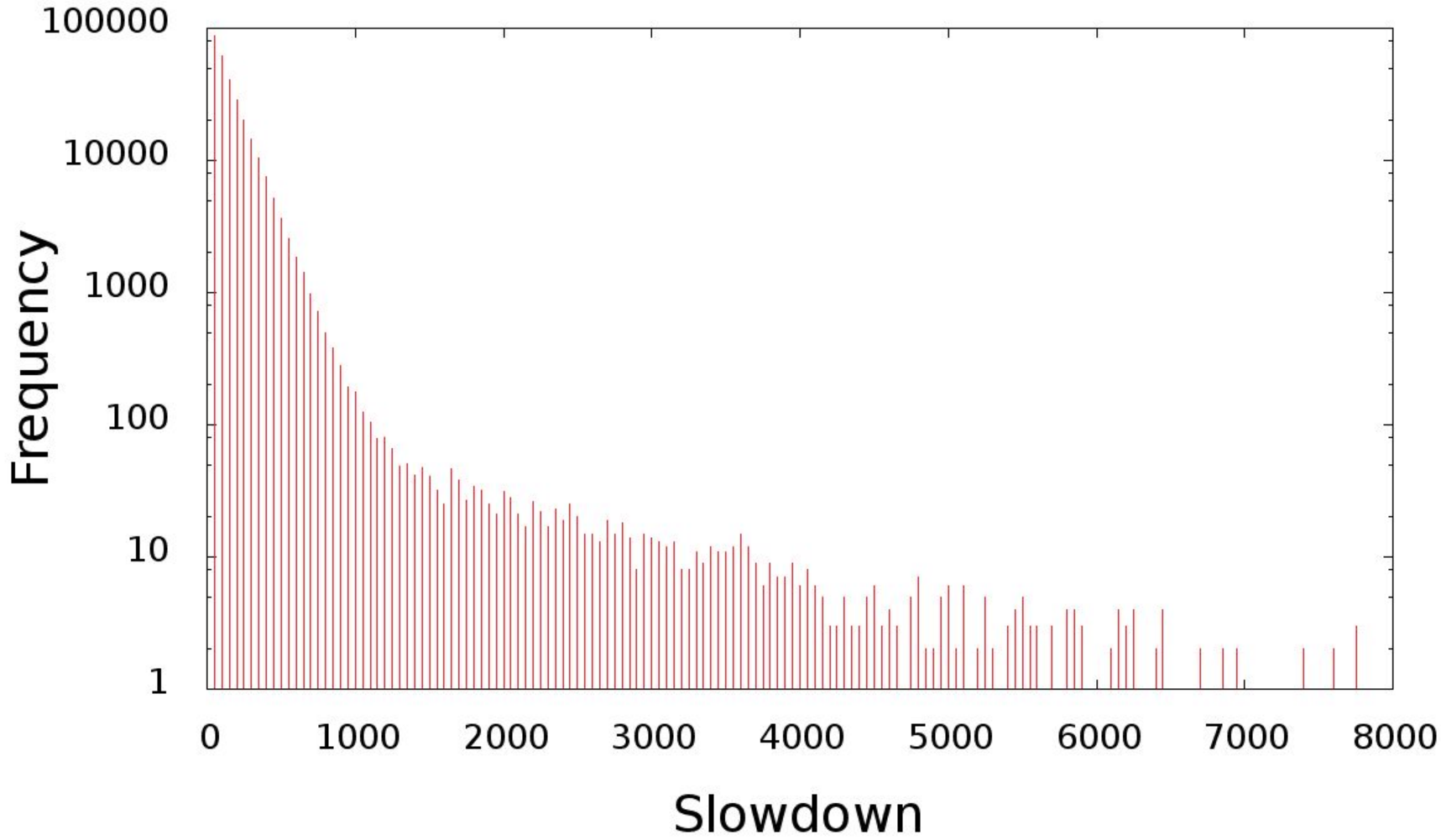
| Expected output | Actual output |
| --- | --- |
| `g_56: 0` | `Strange term cast (cast(t2:reg32t)L:reg8t)U: reg32t ^ 0xbc84814c:reg32t` |

# Conclusions

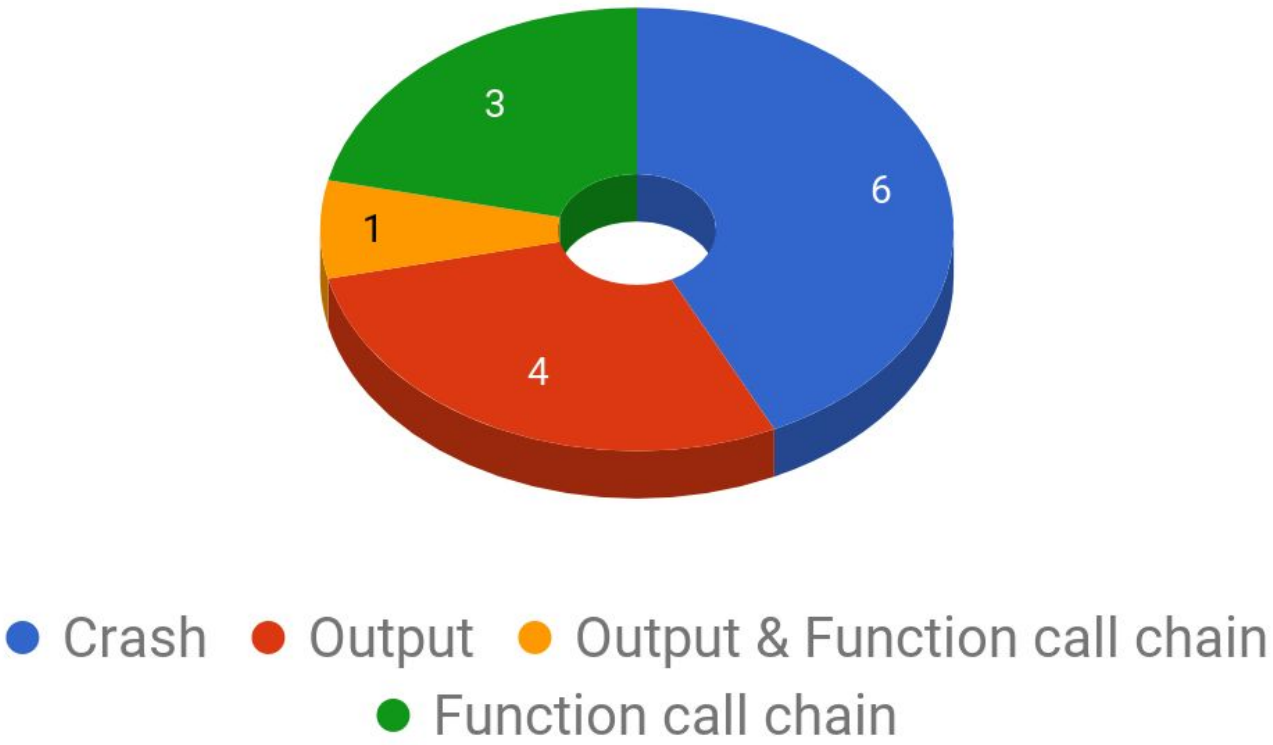
- Developed techniques that test many aspects of symbolic executors
- Applied them to 3 different symbolic executors
- Total bugs found:
  - 14 in *KLEE*
  - 3 in *Crest*
  - 3 in *FuzzBALL*

# Constrainers

| **Type** | **Constraint** |
|---|---|
| $<$ ,$>$ | $\neg(x < v) \wedge \neg(x > v)$ |
| $\leq, \geq$ | $x \leq v \wedge x \geq v$ |
| range | $\neg(x \leq v - 2) \wedge \neg(x \geq v + 3) \wedge$ $\neg(x = v - 1) \wedge \neg(x = v + 1) \wedge \neg(x = v + 2)$ |
| divisors | $\wedge_i \neg(x \mod d_i \neq 0) \wedge x > 1 \wedge x \leq v$ |


100000
10000
1000
100
10
1
Frequency
0
1000
2000
3000
4000
5000
6000
7000
8000
Slowdown

# KLEE bugs by oracle



# CREST bug

- 14 in *KLEE* (9 fixed)
- 3 in *Crest* (1 fixed)
- 3 in *FuzzBALL* (3 fixed)

- found within first 5000 runs of a batch

```
unsigned int a;
int main() {
  __CrestUInt(&a);
  printf("a: %d\n",a);
  if( a < 2294967295) {
    exit(0);
  }
}
```

# Single-Path Mode

*Compare native execution, with symbolic execution constrained to the exact same path as native execution.*

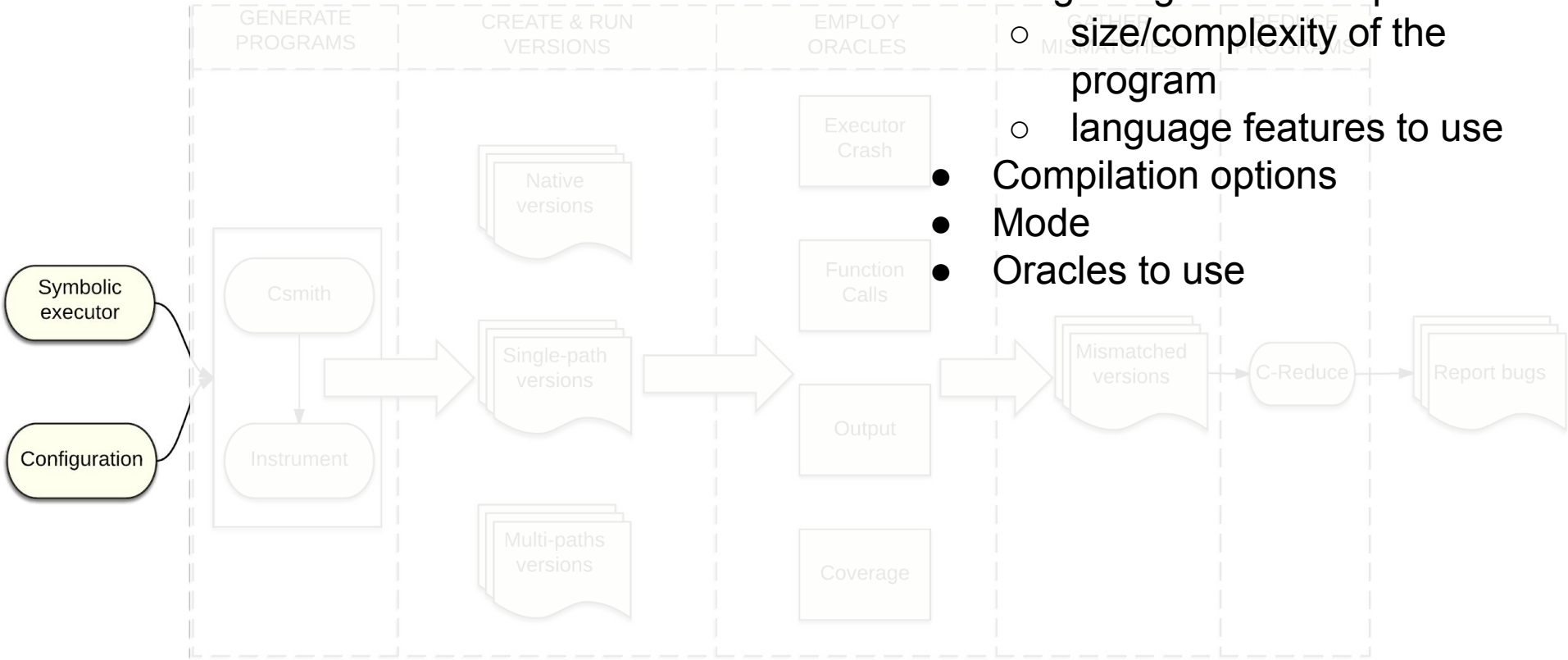
```
int x = 5;
make_symbolic(&x);
if (x < 5) silent_exit(0);
if (x > 5) silent_exit(0);
```

# Symbolic execution

- *Mark some inputs as symbolic*
- *Runs the program, while gathering constraints on the symbolic data*
- *Forks at branch points when both sides are feasible*
- *Upon hitting a terminal state (ie. error), solves the gathered constraints, to produce an input leading the program to the same state*

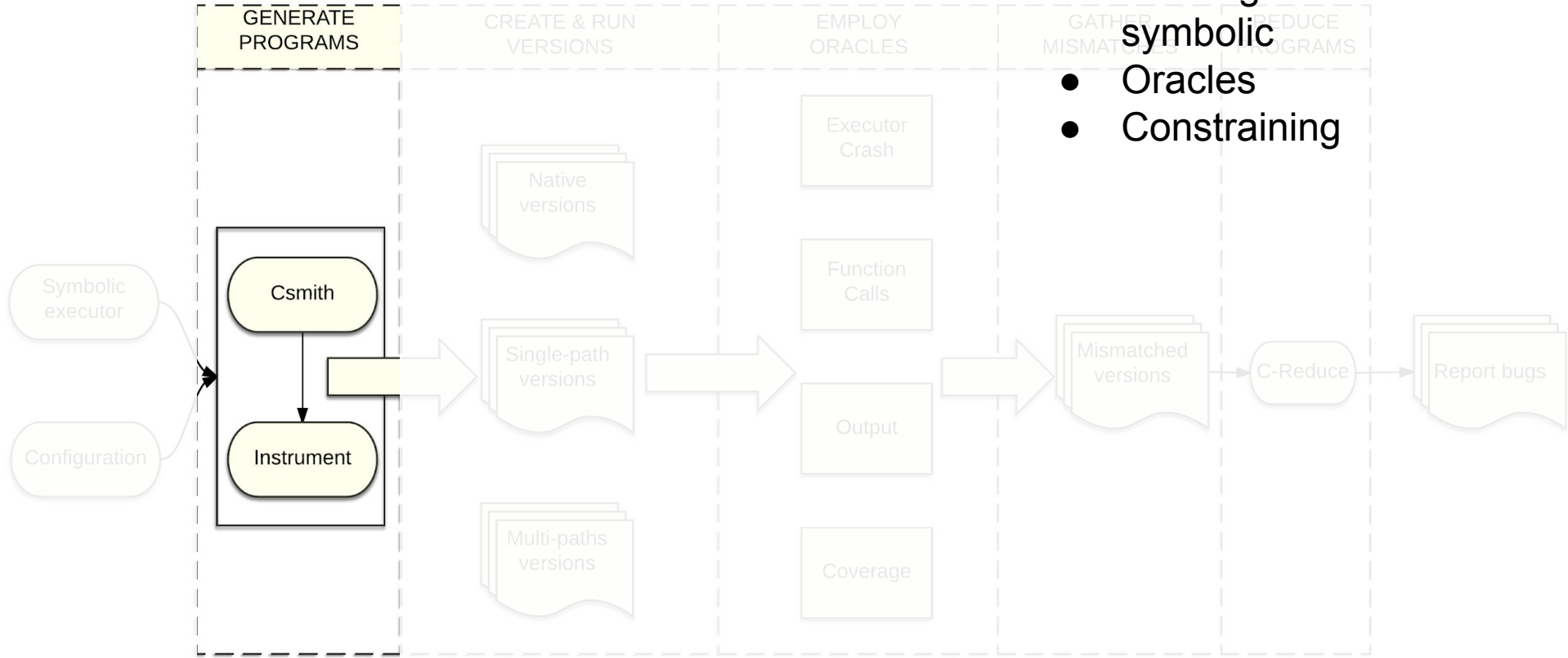
Configuration includes:

- Program generation options
  - size/complexity of the program
  - language features to use
- Compilation options
- Mode
- Oracles to use



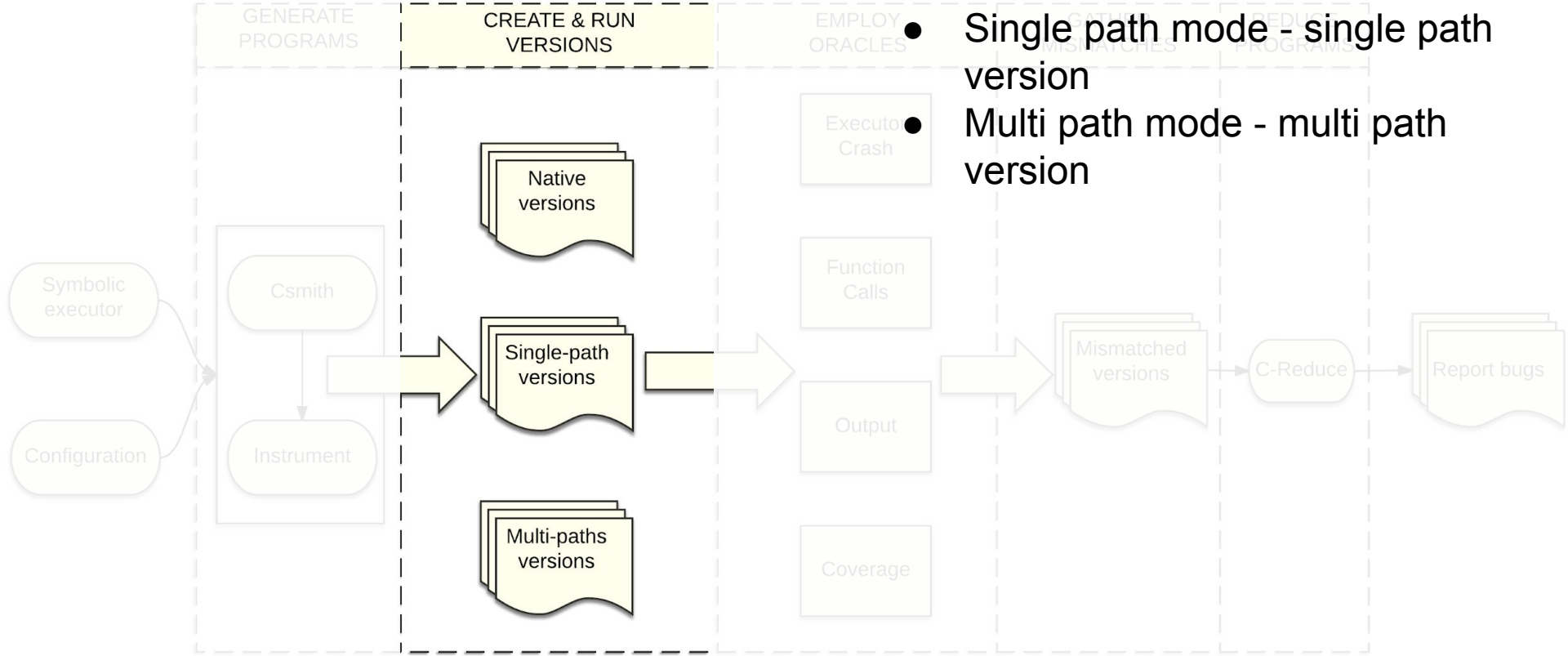
Instrumentation supports

- Marking variables as symbolic
- Oracles
- Constraining



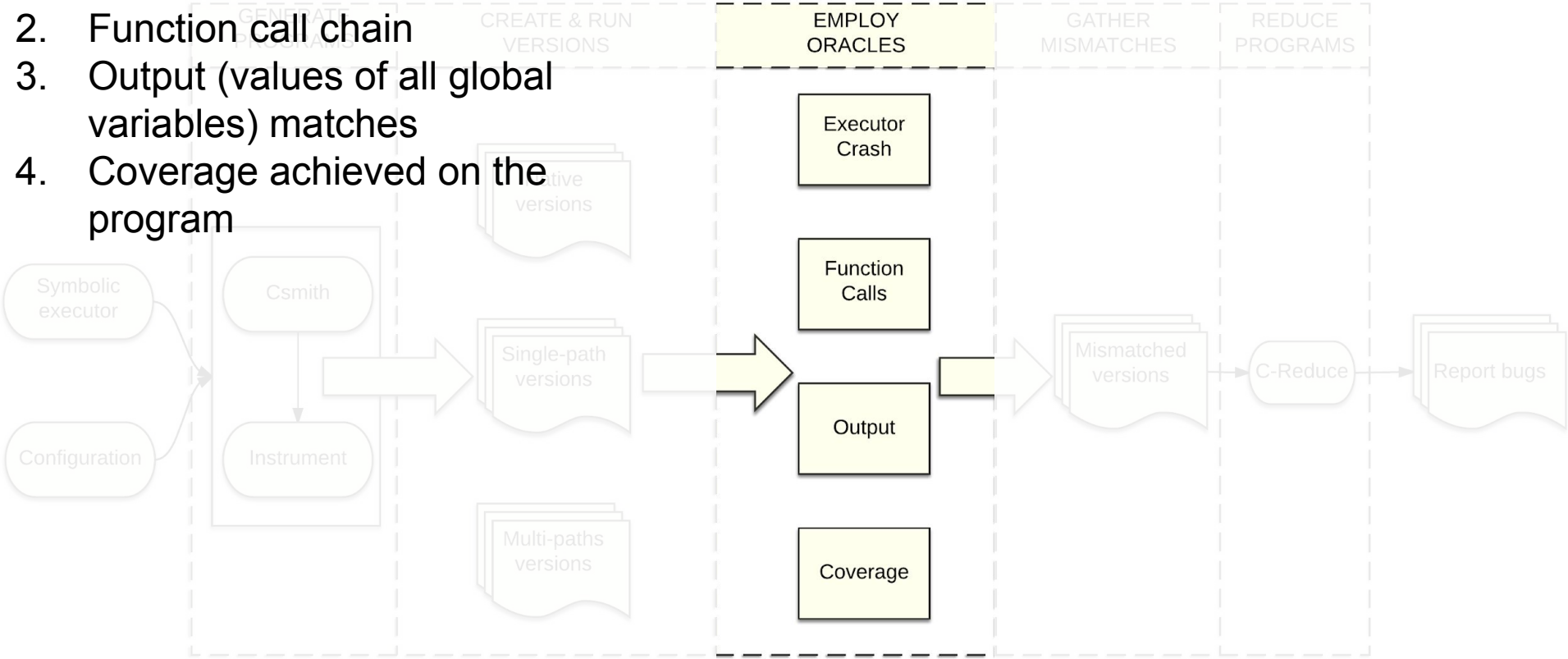
versions correspond to mode used:

- Concrete mode - native version
- Single path mode - single path version
- Multi path mode - multi path version



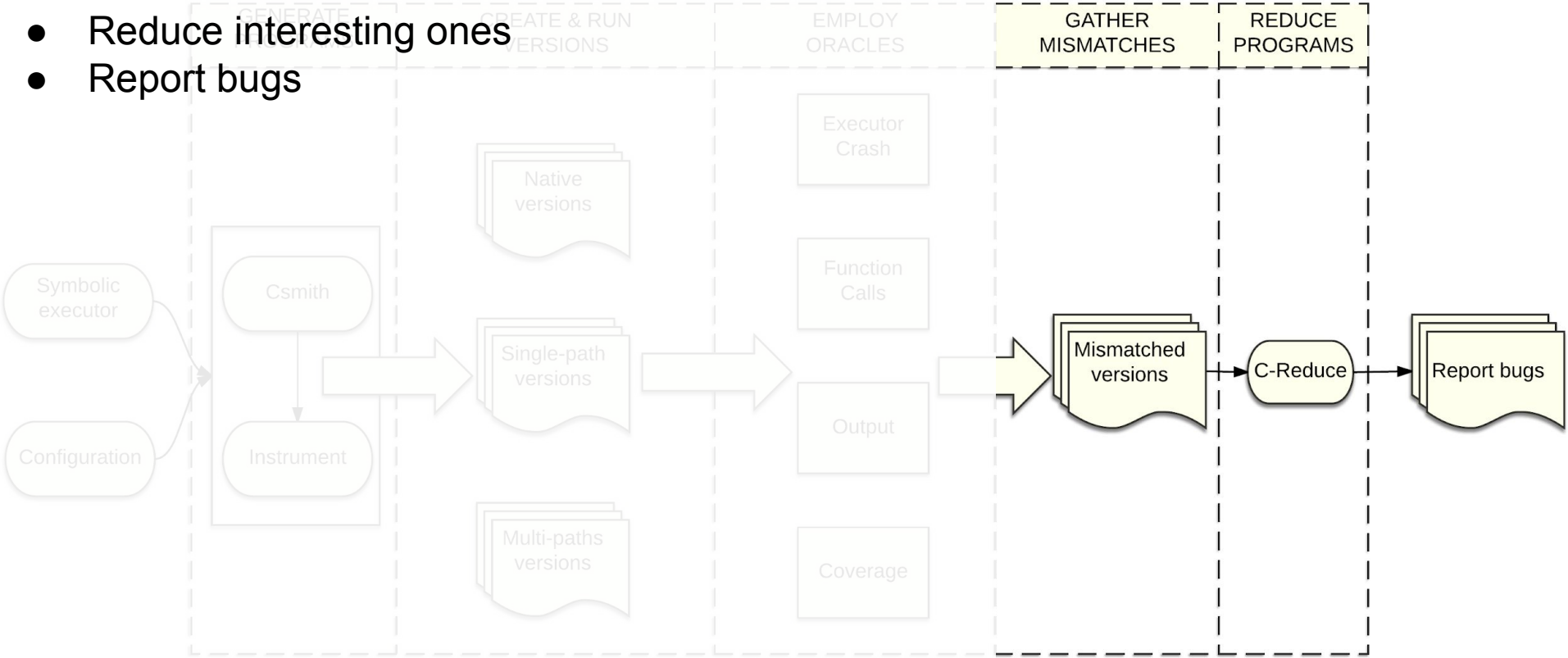
Oracles can check:

1. Executor doesn't crash
2. Function call chain
3. Output (values of all global variables) matches
4. Coverage achieved on the program



Finally:

- Gather mismatches
- Reduce interesting ones
- Report bugs



```
1 void foo(unsigned int x) {
2   if( x > 2294967295) {
3     assert(false);
4   }
5   printf("x: %u\n", x);
6 }
```

| Expected output | Actual output |
| --- | --- |
| `x: 6`<br>`Assertion fail` | `x: 6`<br>`x: 23` |