Chopped Symbolic Execution

David Trabish, Andrea Mattavelli, Noam Rinetzky and Cristian Cadar

ICSE 2018, Gothenburg, Sweden

Symbolic Execution: Introduction

Systematic approach for program path exploration

- Test input generation
- Bug finding
- Patch testing
- Cross checking

Symbolic Execution: Introduction

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

```
int x = 0;
int y = 0;
// inputs
int j, k;
```

Symbolic Execution: Introduction

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

```
int x = 0;
int y = 0;
// inputs
int j, k;
```

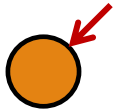
concrete inputs

k	j	x	y
0	0	**0**	**0**
0	1	**0**	**1**
0	2	**0**	**1**
10	0	**1**	**0**
11	0	**1**	**0**
70	30	**1**	**1**
71	40	**1**	**1**
72	50	**1**	**1**

Symbolic Execution: Introduction

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

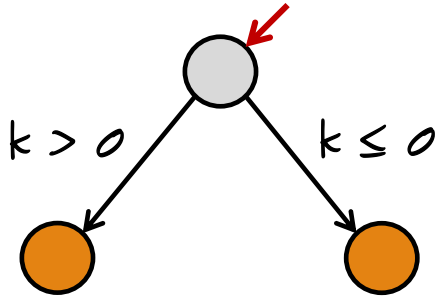
```
int x = 0;
int y = 0;
// inputs
int j, k;
```



Symbolic Execution: Introduction

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

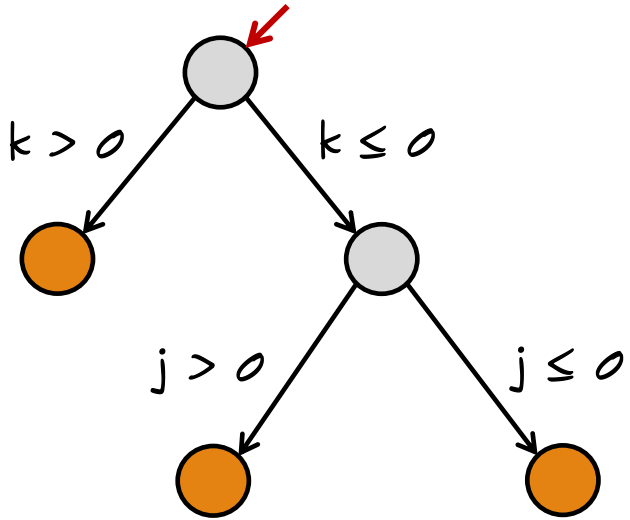
```
int x = 0;
int y = 0;
// inputs
int j, k;
```



Symbolic Execution: Introduction

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

```
int x = 0;
int y = 0;
// inputs
int j, k;
```



Symbolic Execution: Main Challenges

Path Explosion

Constraint Solving

Symbolic Execution: Main Challenges

Path Explosion

Motivating Example: CVE-2015-3622

- An ***out-of-bound-read*** vulnerability in **GNU libtasn1**
- GNU Libtasn1 is a library for decoding/encoding ASN.1 data
  - Used in *GnuTLS*

Motivating Example: CVE-2015-3622

```
int extract_octet(...) {
  len = decode_element(...);
  while (counter < str_len) {
    len = decode_element(...);
    if (len >= 0)
      append_value(...);
    else
      result = extract_octet(...);
  }
}
```

```
int decode_element(...) {
  // some checks here...
  int i = 1, k = data[0] & 0x7f;
  unsigned int answer = 0;
  while (i <= k && i < data_len) {
    answer += data[i] // error here!
    i++;
  }
  ...
}
```

Motivating Example: CVE-2015-3622

Two intertwined flows:

- **Parsing**
- **AST Building**

```
int extract_octet(...) {
  len = decode_element(...);
  while (counter < str_len) {
    len = decode_element(...);
    if (len >= 0)
      append_value(...);
    else
      result = extract_octet(...);
  }
}
```

Motivating Example: CVE-2015-3622

Key Observation:

- **Blue** flow performs **symbolically expensive** operations
- **Can be avoided** on most of the paths

Chopped Symbolic Execution: Main Idea

- Exploration skips **unwanted** functions
  - User specified
  - Expensive to analyze
- Resolves relevant side effects on demand

Chopped SE by Example

```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

```
int j; // symbolic
int k; // symbolic
int x = 0;
int y = 0;
```

Chopped SE by Example

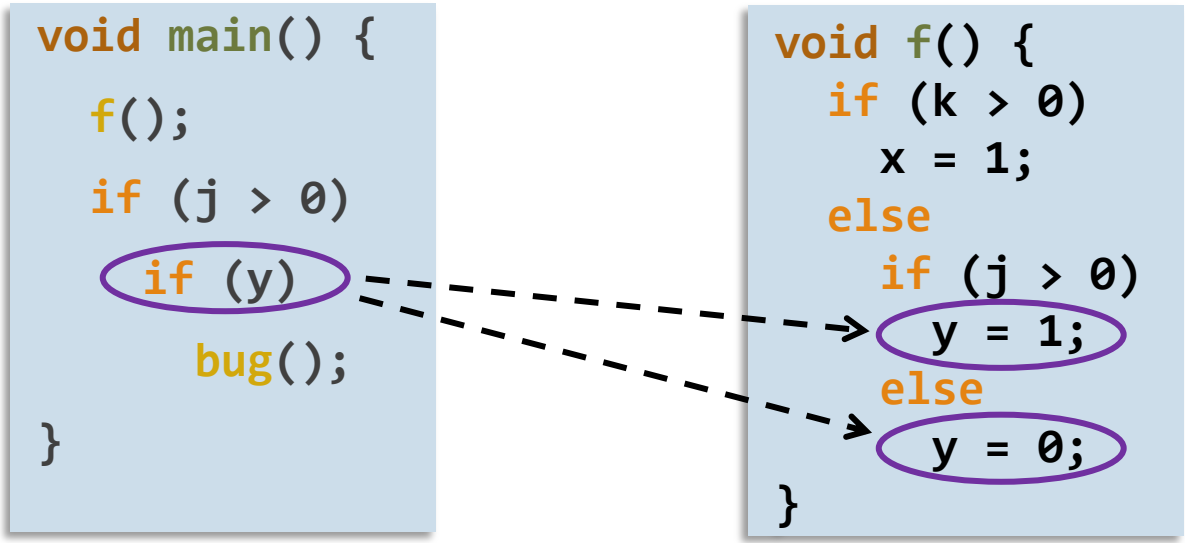
```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

Ref(main) = **{j, y}**

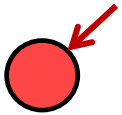
```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Mod(f) = **{x, y}**

Chopped SE by Example

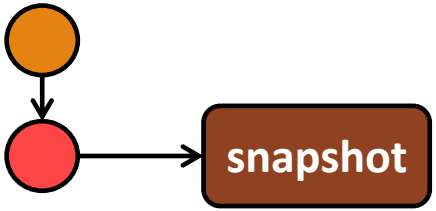


Chopped SE by Example



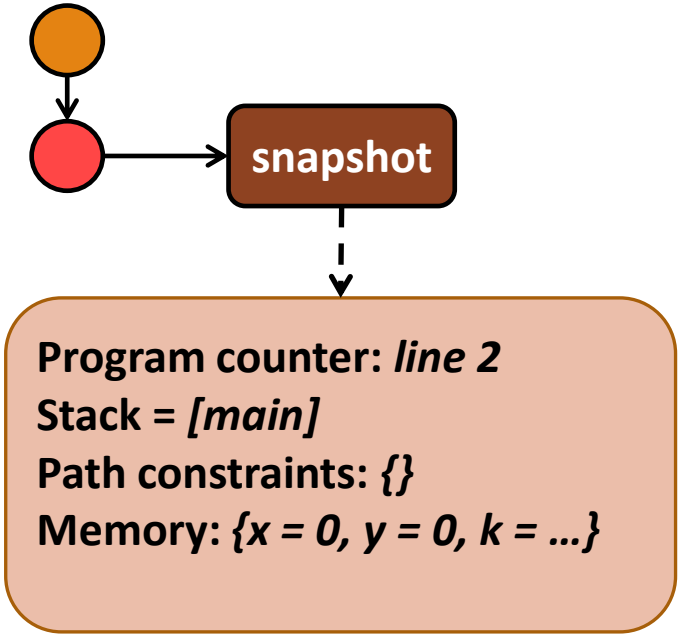

```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

Chopped SE by Example



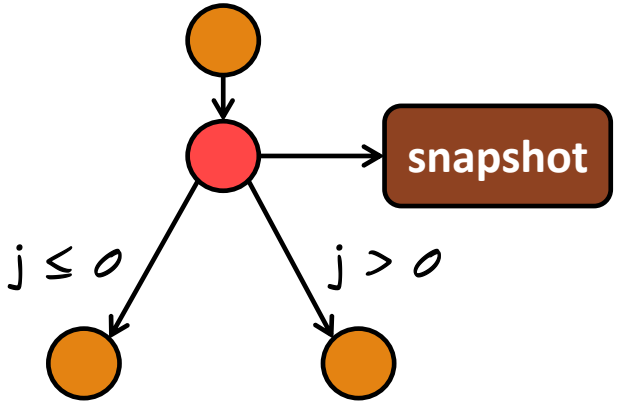

```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

Chopped SE by Example



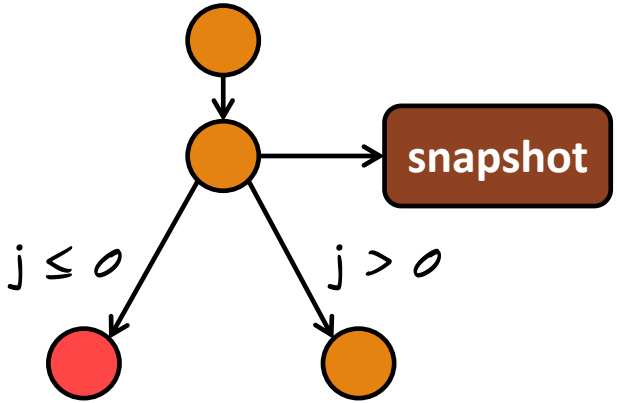

```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

Chopped SE by Example



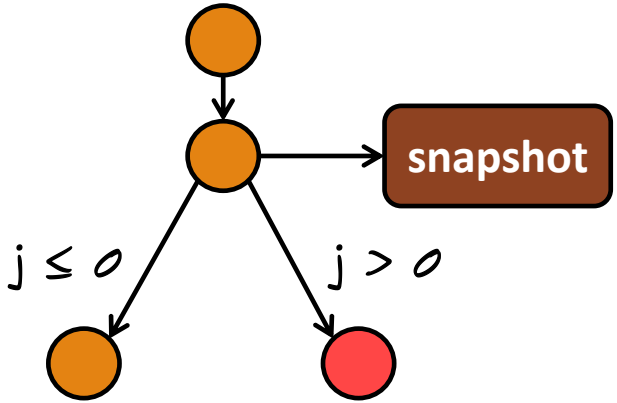

```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

Chopped SE by Example




```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

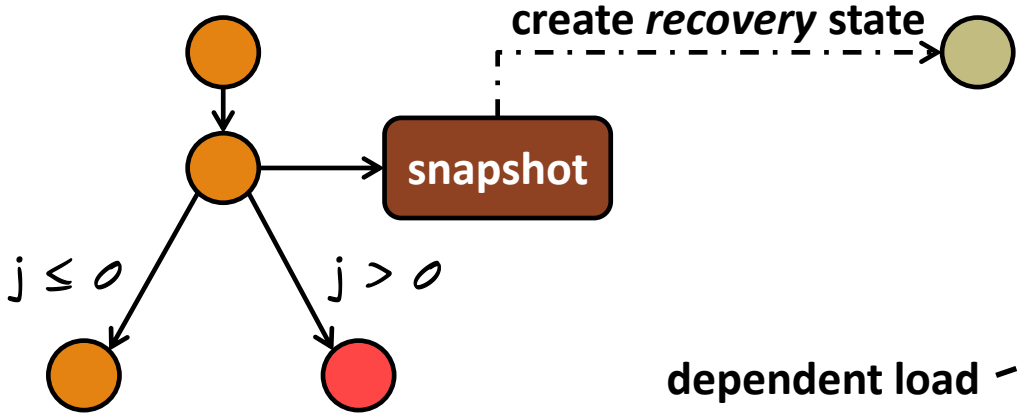
Chopped SE by Example




```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

dependent load

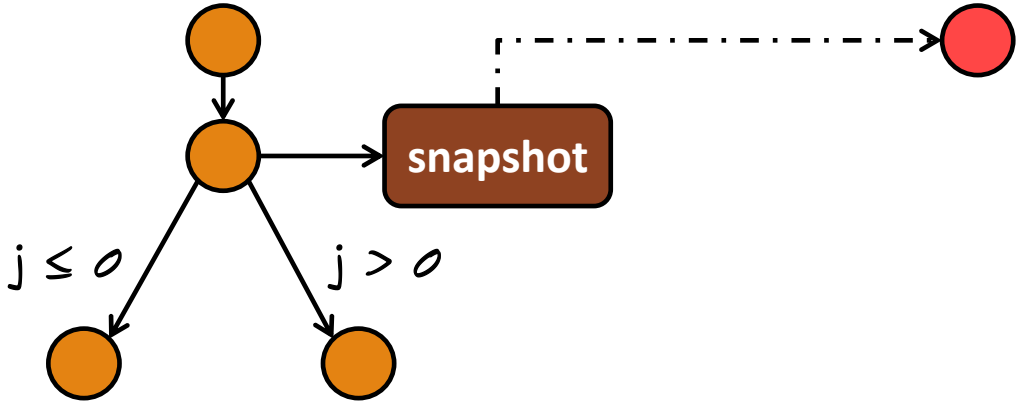
Chopped SE by Example




```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

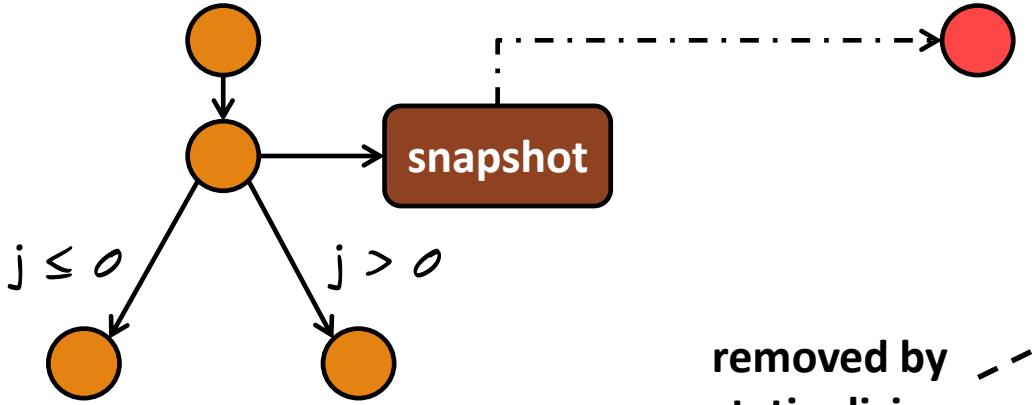
dependent load

Chopped SE by Example




```
void f() {
  if (k > 0)
    x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

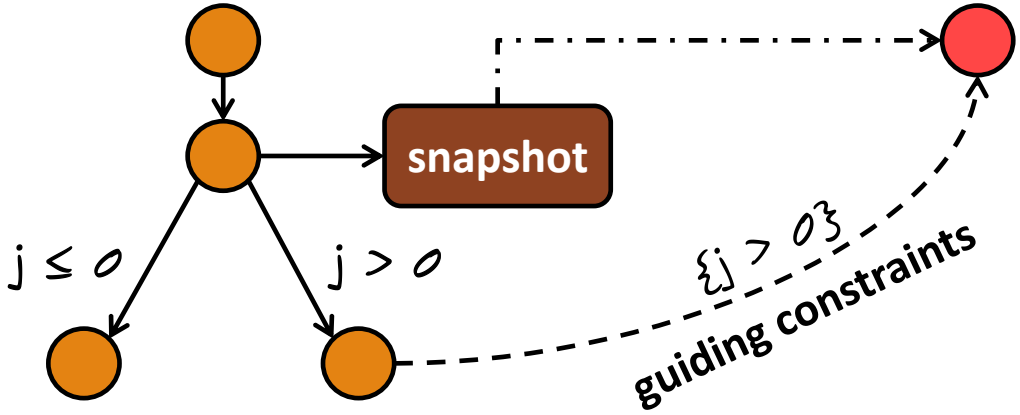
Chopped SE by Example



removed by
static slicing

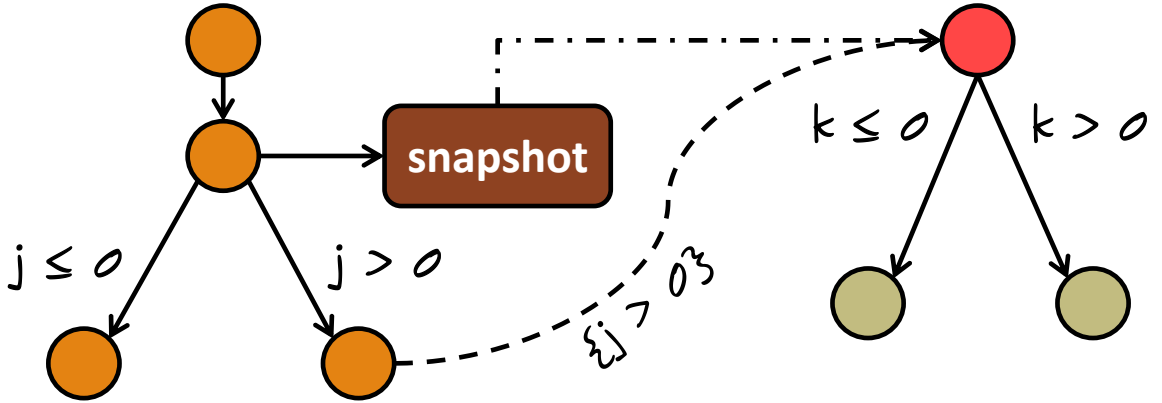
```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Chopped SE by Example



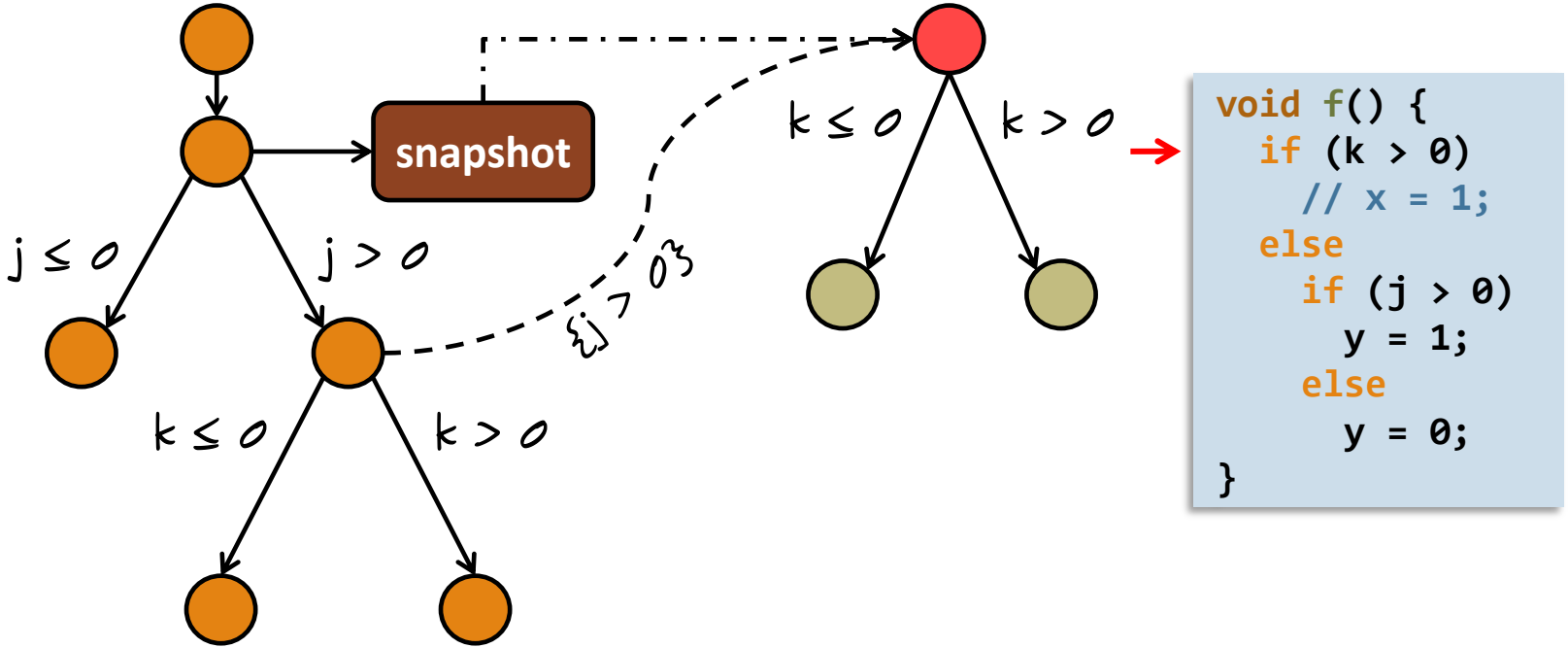

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Chopped SE by Example

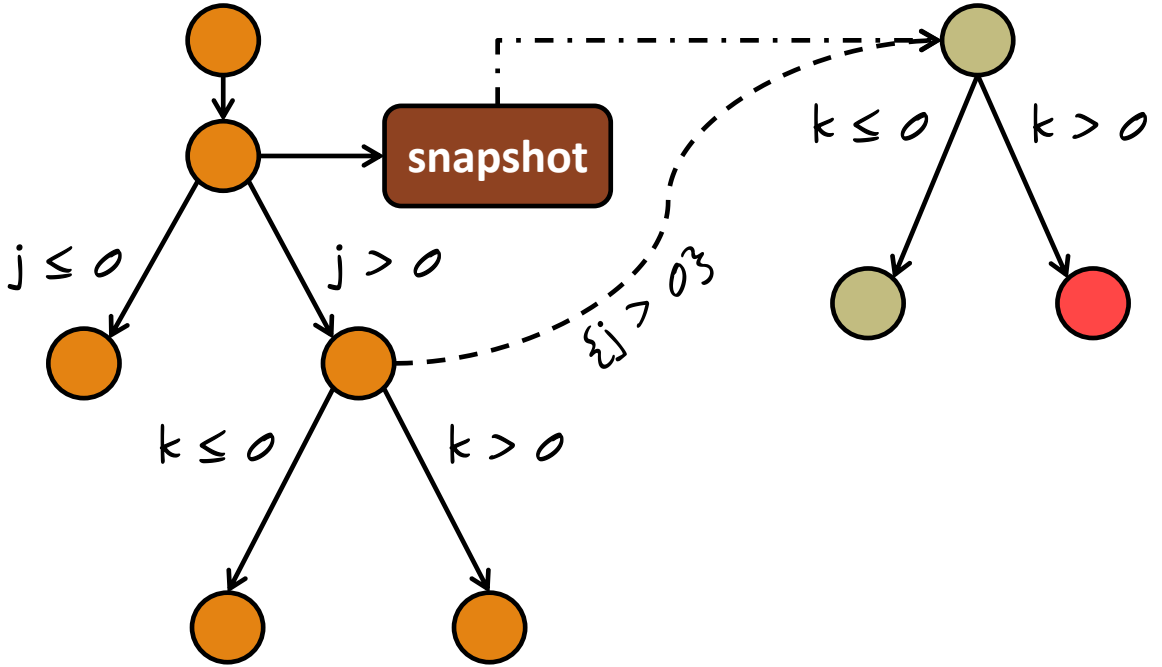



```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Chopped SE by Example

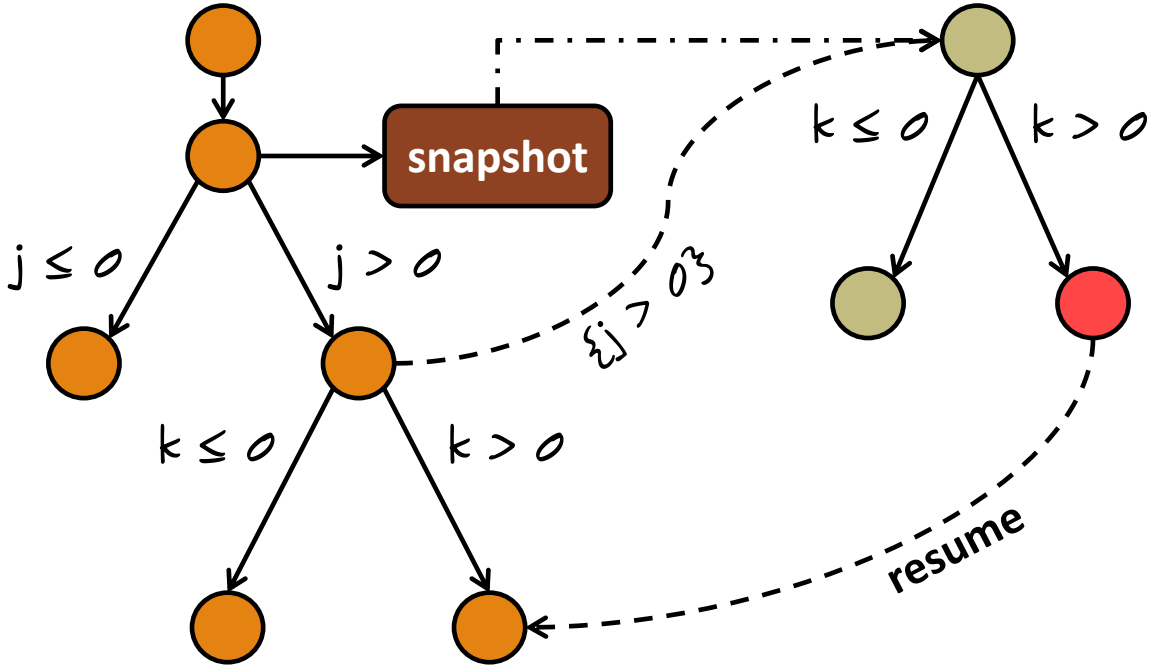


Chopped SE by Example



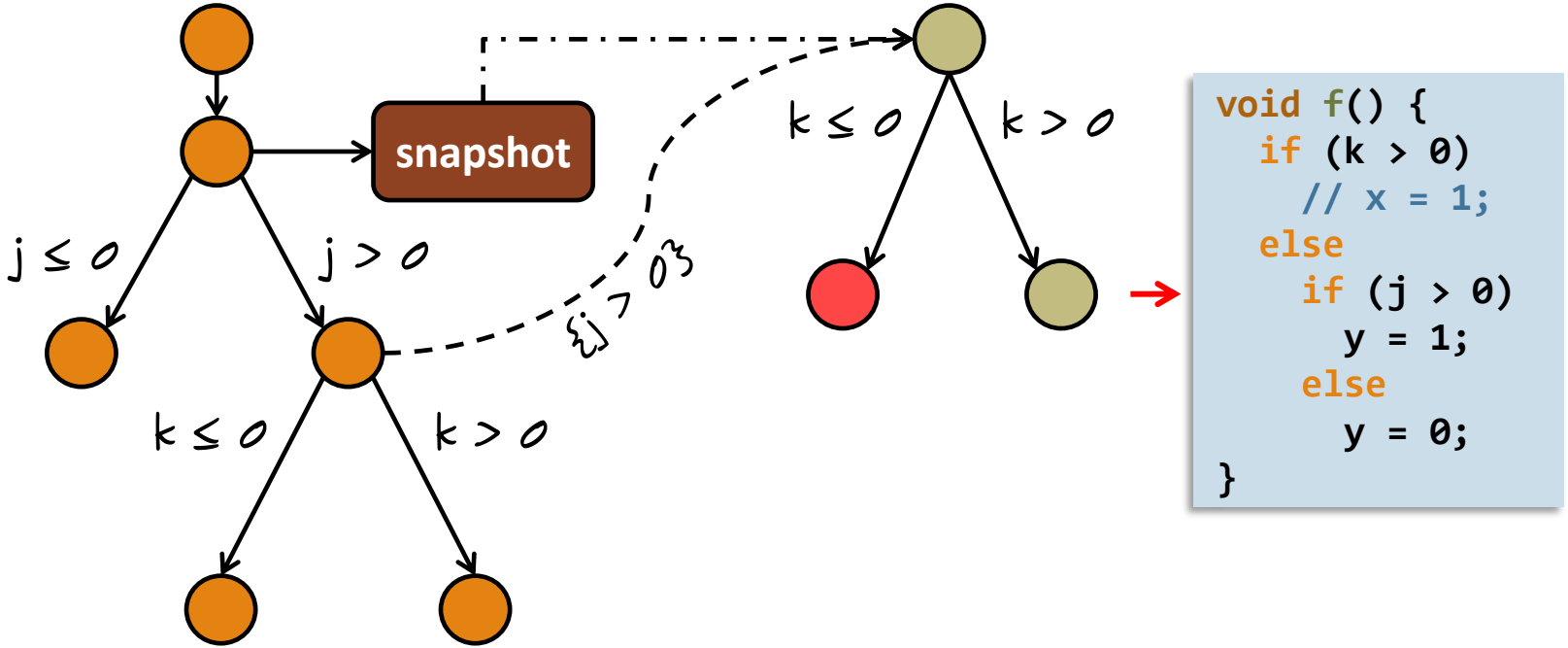

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Chopped SE by Example

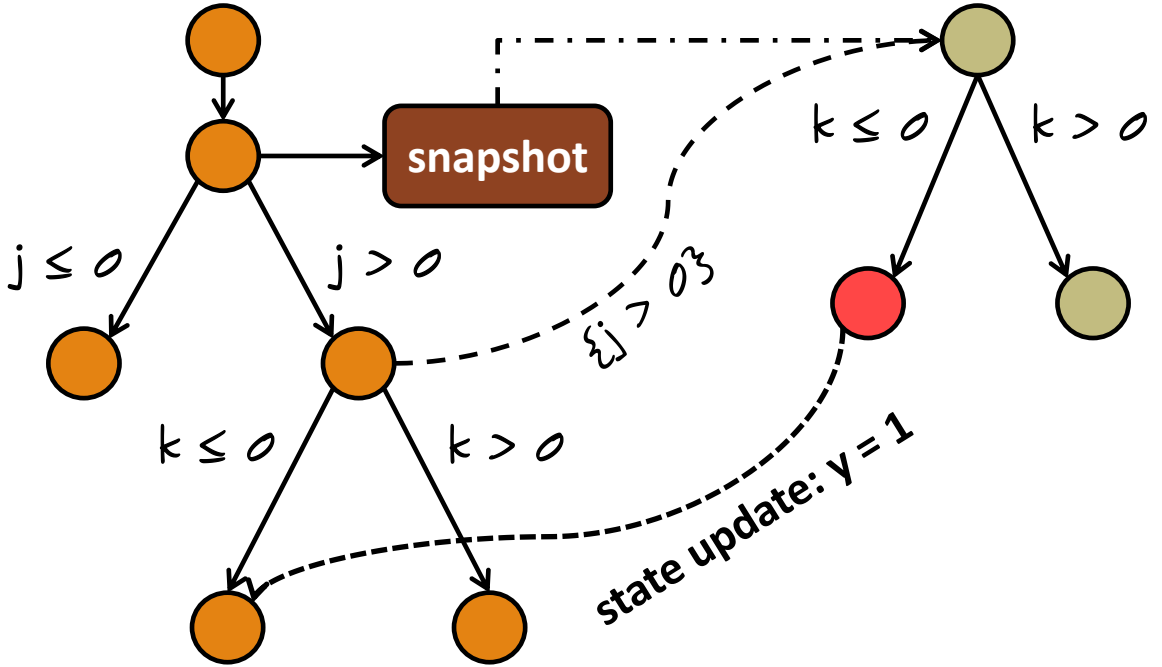



```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Chopped SE by Example

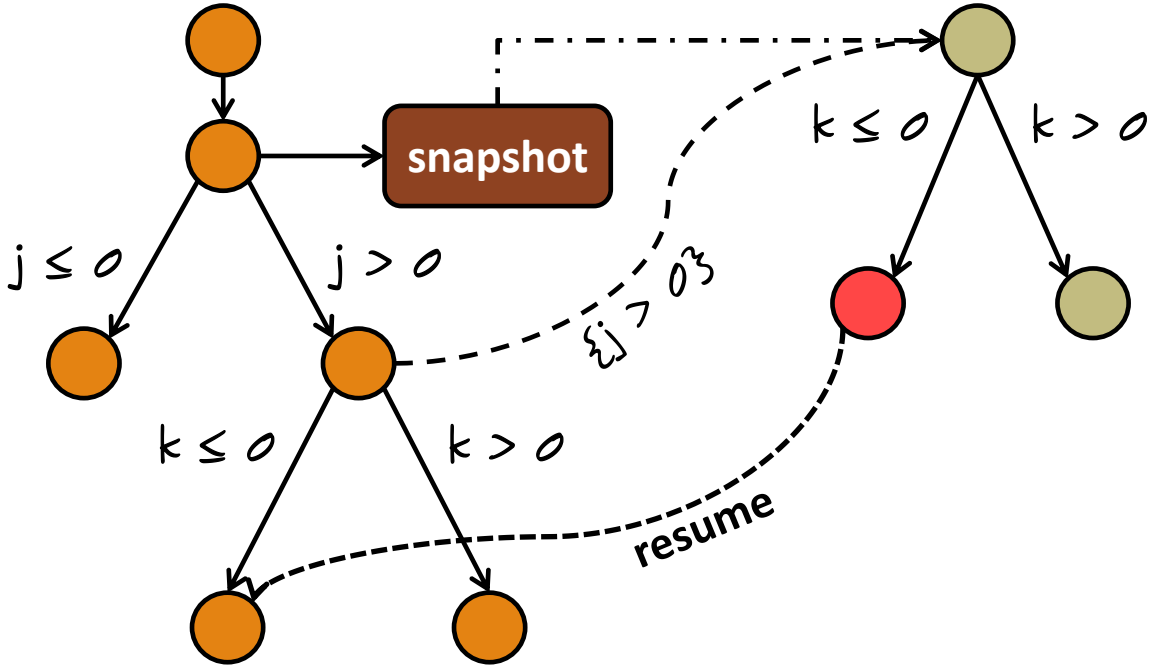


Chopped SE by Example



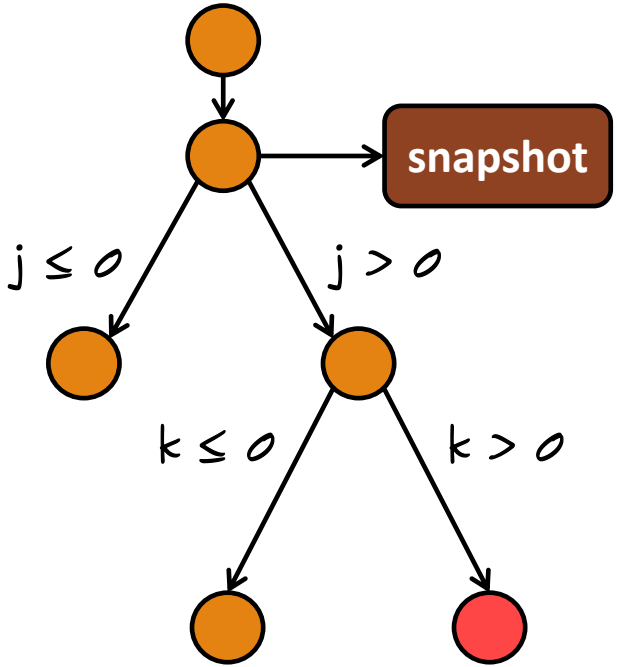

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Chopped SE by Example



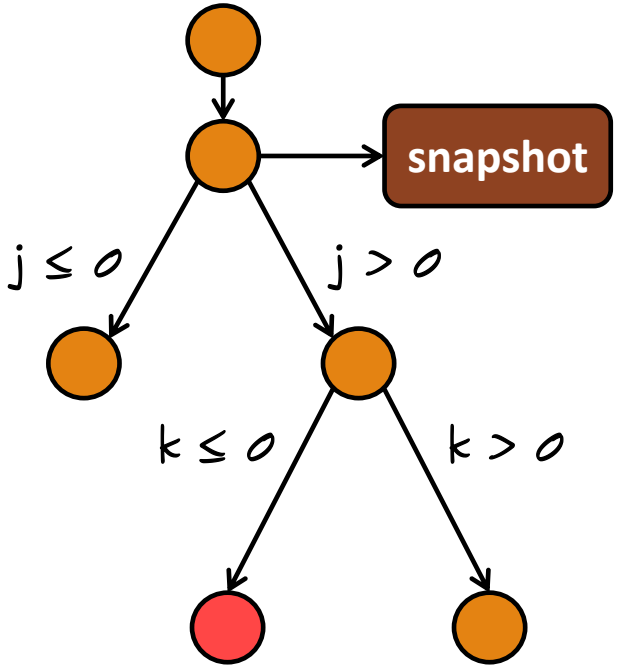

```
void f() {
  if (k > 0)
    // x = 1;
  else
    if (j > 0)
      y = 1;
    else
      y = 0;
}
```

Chopped SE by Example



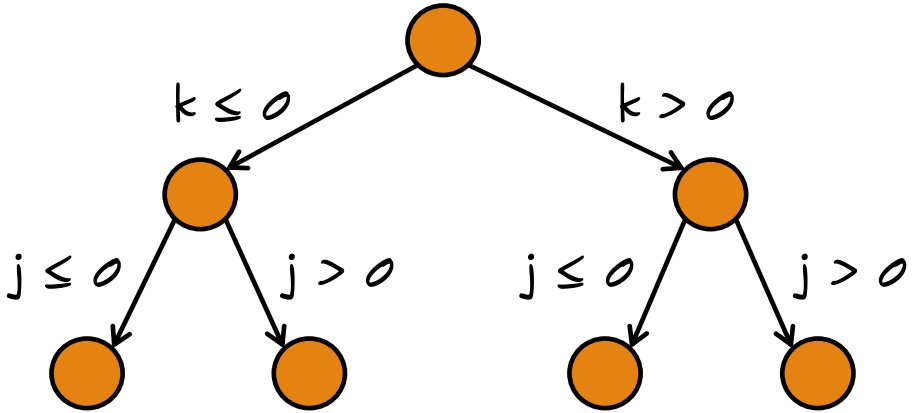

```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

Chopped SE by Example

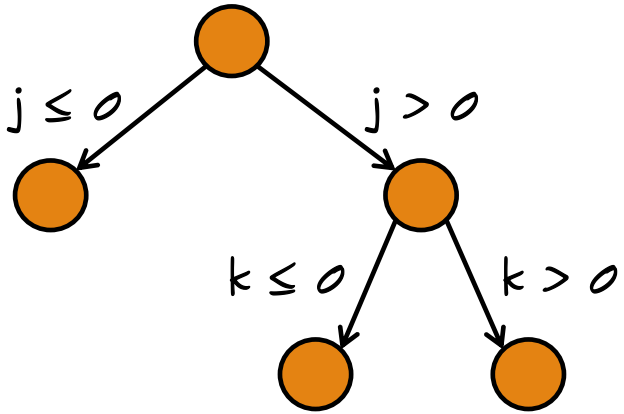



```
void main() {
  f();
  if (j > 0)
    if (y)
      bug();
}
```

Process Trees



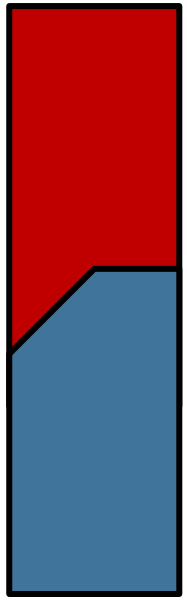
Standard SE



Chopped SE

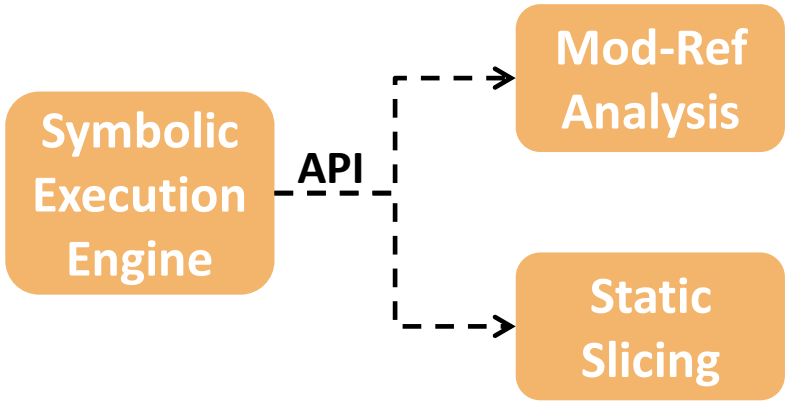
Chopping-Aware Search Heuristic

- The state space is divided into two parts:
  - **Normal** states
  - **Recovery** states
- **Normal states** are selected with **higher** probability



Implementation: CHOPPER

- Mod-Ref analysis
  - Based on the pointer analysis provided by the ***SVF*** project
- Static slicing
  - Based on the ***DG*** project
- Symbolic Execution Engine
  - Built on top of ***KLEE***



Experiments

- Bug reproduction
  - GNU libtasn1
- Test suite augmentation
  - GNU BC
  - LibYAML
  - GNU oSIP

Bug Reproduction

Benchmark: GNU libtasn1

Methodology:

- Creating a test driver for the *libtasn1* library
- Manually identifying the functions to skip
- Time limit of 24 hours
- Memory limit of 4 GB
- Execution is terminated once the error is discovered

Bug Reproduction

CVE \ Heuristic	Random	
	KLEE	CHOPPER
2012-1569	***OOM***	**00:02**
2014-3467 (1)	00:01	**00:01**
2014-3467 (2)	01:02	**00:06**
2014-3467 (3)	***Timeout***	**00:10**
2015-2806	01:07	**00:02**
2015-3622	***Timeout***	**00:01**

Bug Reproduction

CVE \ Heuristic	Random		DFS		Coverage	
	KLEE	CHOPPER	KLEE	CHOPPER	KLEE	CHOPPER
2012-1569	***OOM***	**00:02**	***OOM***	**00:03**	***OOM***	**00:01**
2014-3467 (1)	00:01	**00:01**	00:17	**00:01**	00:01	**00:01**
2014-3467 (2)	01:02	**00:06**	***Timeout***	**00:01**	01:34	**00:03**
2014-3467 (3)	***Timeout***	**00:10**	***Timeout***	**00:13**	***Timeout***	**00:10**
2015-2806	01:07	**00:02**	02:46	**00:12**	***OOM***	**00:01**
2015-3622	***Timeout***	**00:01**	***Timeout***	**00:18**	20:25	**00:01**

Test Suite Augmentation

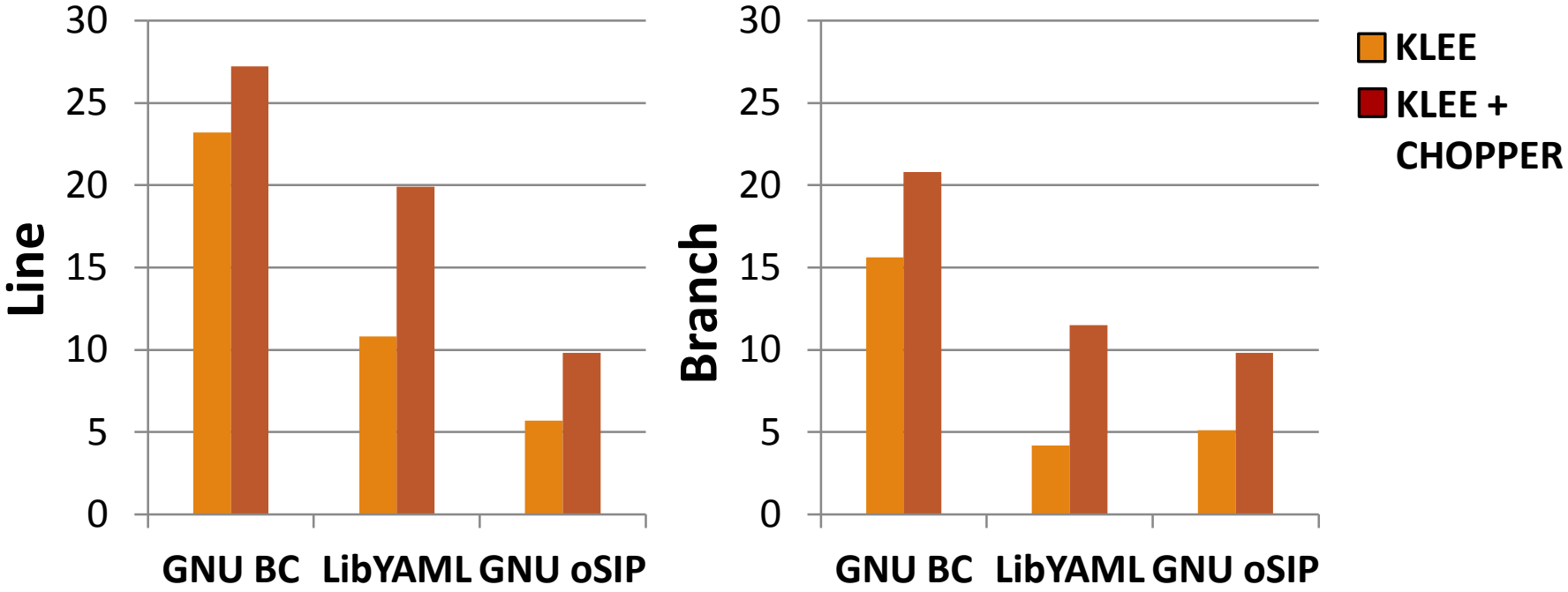
Benchmarks:

- GNU BC
- LibYAML
- GNU oSIP

Methodology:

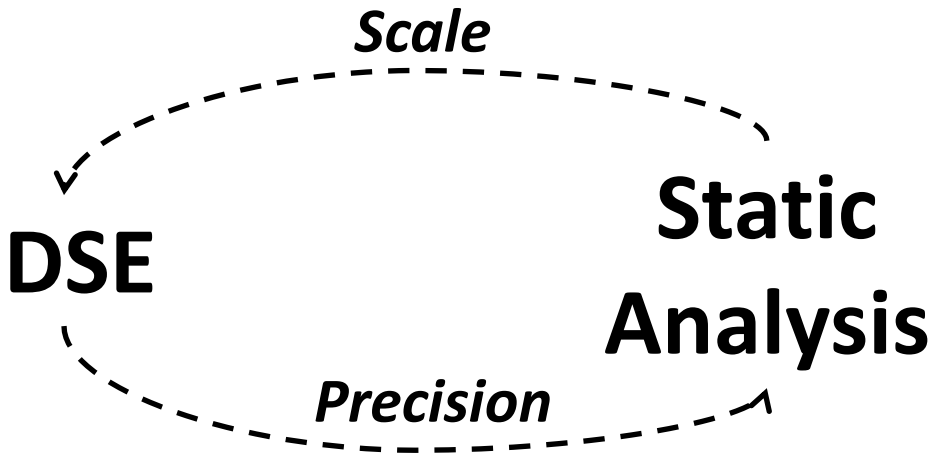
- Run KLEE with coverage based search heuristic for one hour
- Compute the structural coverage using *GCov*
- Identifying the functions to skip
- Run CHOPPER with coverage based search heuristic for one hour

Test Suite Augmentation



Conclusion

- Excluding parts of the CFG is useful in different scenarios
- DSE benefits from static analysis (and vice versa)



Thanks!