Verifying Systems Rules Using Rule-Directed Symbolic Execution

Heming Cui, Gang Hu, Jingyue Wu, Junfeng Yang
(ASPLOS'13)

Qingzhao Zhang, Qinyun Wang, Haizhong Zheng, Ruizhi You

Goal: Verifying Code Rules

- Code Rules
  - In one execution, a file must be closed after be opened.
  - In one execution, allocated memory must be freed once.
  - …
- Violation of code rules leads to …
  - Crashes
  - Resource leaks
  - Vulnerabilities
  - ...

Verifying Code Rules: Approaches

- Static Analysis
  - Data/control dependency analysis, etc
  - High coverage but high false positive rate
  - Hard to capture runtime effects
- **Symbolic Execution**
  - Execute programs on symbolic inputs
  - Low scalability but low false positive rate
  - Efficient on constraint solving

Background: Symbolic Execution

```
1 void f() {              // input = symbolic(x)
2     int a = input();    // a = x
3     b = a * 2;          // b = 2 * x
4     if (b == 12) {      // if (2 * x == 12)
5         fail();
6     } else {
7         ok();
8     }
9 }
```

If “fail” is a violation of code rules...

Q: When the program will fail?
A: x == 6

Challenge: Symbolic Execution

- Limitation: path explosion
  - Conventional symbolic execution explores all feasible paths.
  - The number of paths grows exponentially as the program size increases.
- Problems
  - Some paths may be stuck.
  - The symbolic execution may be slow on large programs.

Insight: Redundant Paths

- It is sufficient to explore a small portion of paths.
  - Only paths with specific events (e.g., open file) are valuable to explore
  - Only instructions with dependencies on events are valuable
- Solution: pruning invaluable paths
  - **Path slicing**

Background: Path Slicing

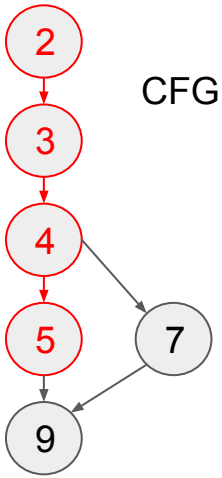
- One type of *program slicing*
- Given a control flow path, determine which edges are relevant with reaching a specific target.

```
1 void f() {
2     int a = input();
3     b = a * 2;
4     if (b == 12) {
5         fail();
6     } else {
7         ok();
8     }
9 }
```

Target: “fail” in Line 5

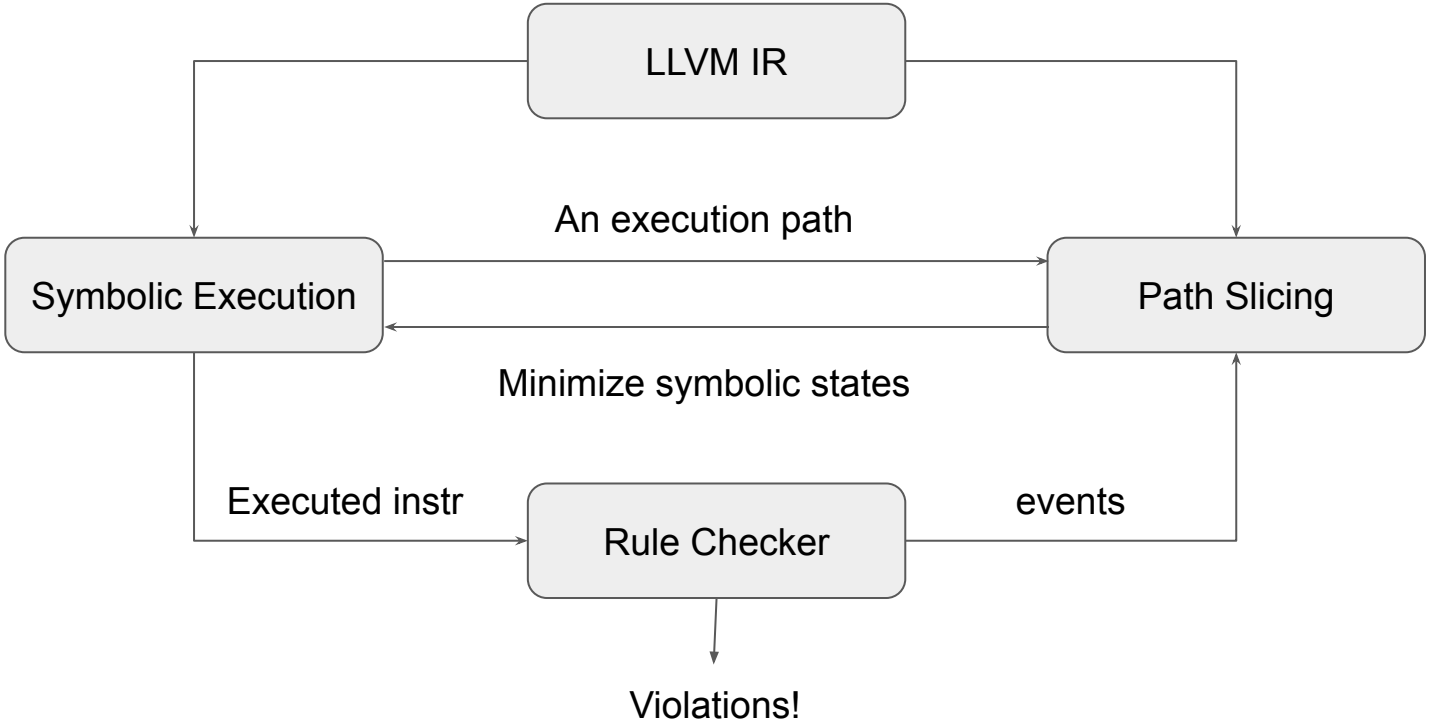
A slice to fail():

Line 2, 3, 4, 5



CFG

WOODPECKER Overview

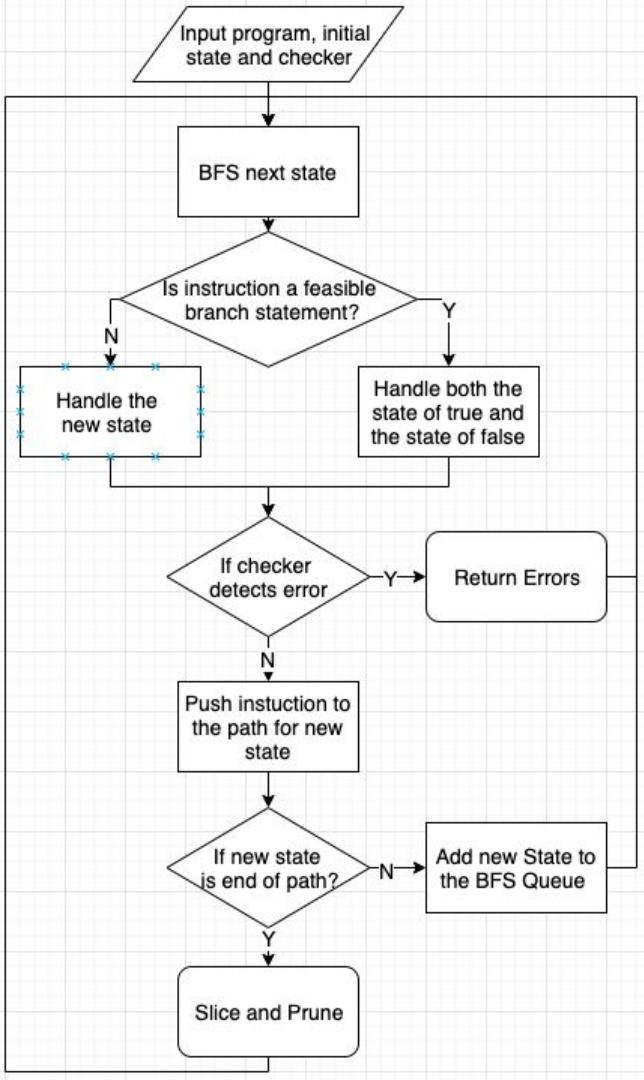


Rule Checker

- Identify events
  - Events are instructions of interests for code rules
    - E.g., open() & close() for File-open-close rule
- Detect violations
  - A violation is a specific ordered sequence of events
    - E.g. open() -> No close()
    - E.g. open() -> close() -> close()

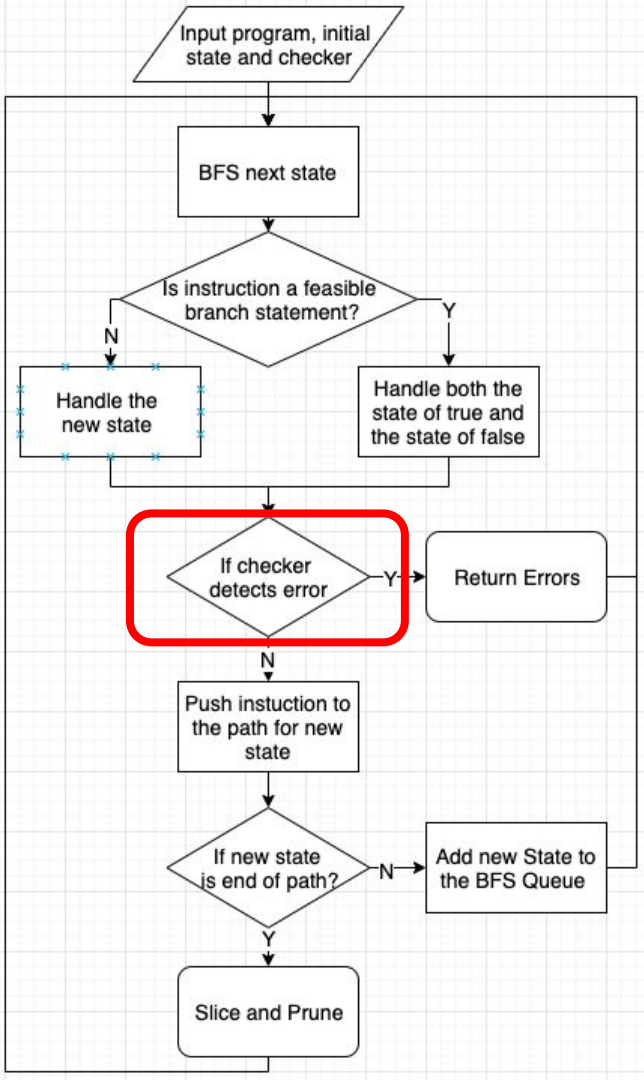
WOODPECKER's Search Algorithm

- **Traverse different execution paths**
  - **Traverse CFG, for each instructions**
    - Check violation of Rules
    - Check the end of path
  - If it is an violation
    - Report the violation
  - If it is the end of a path
    - Start slicing and pruning



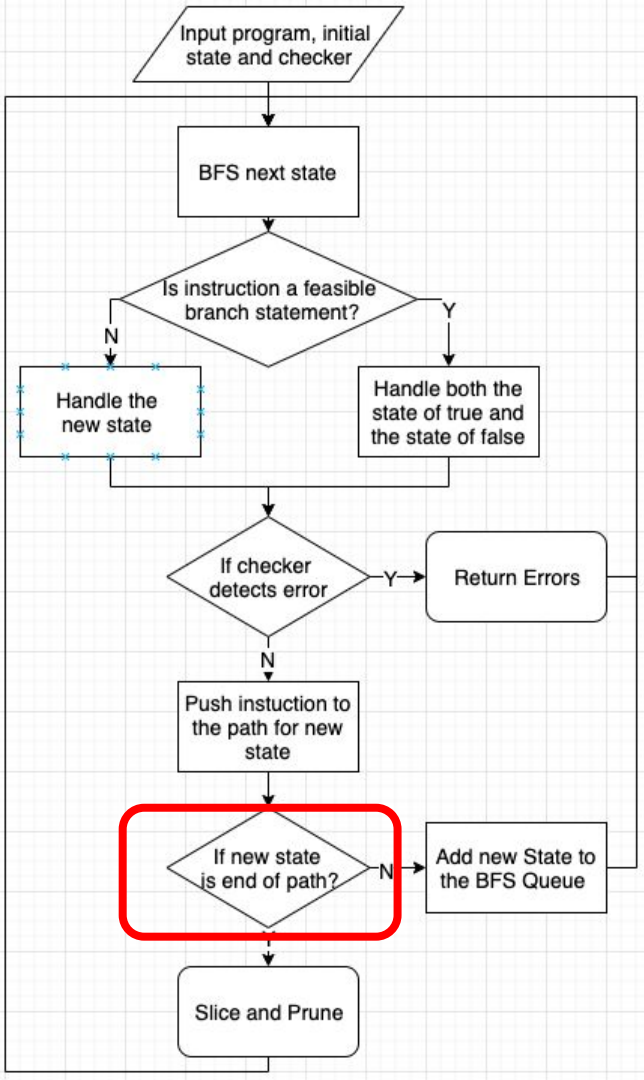
WOODPECKER's Search Algorithm

- **Traverse different execution paths**
  - **Traverse CFG, for each instructions**
    - Check violation of Rules
    - Check the end of path
  - If it is an violation
    - Report the violation
  - If it is the end of a path
    - Start slicing and pruning



WOODPECKER's Search Algorithm

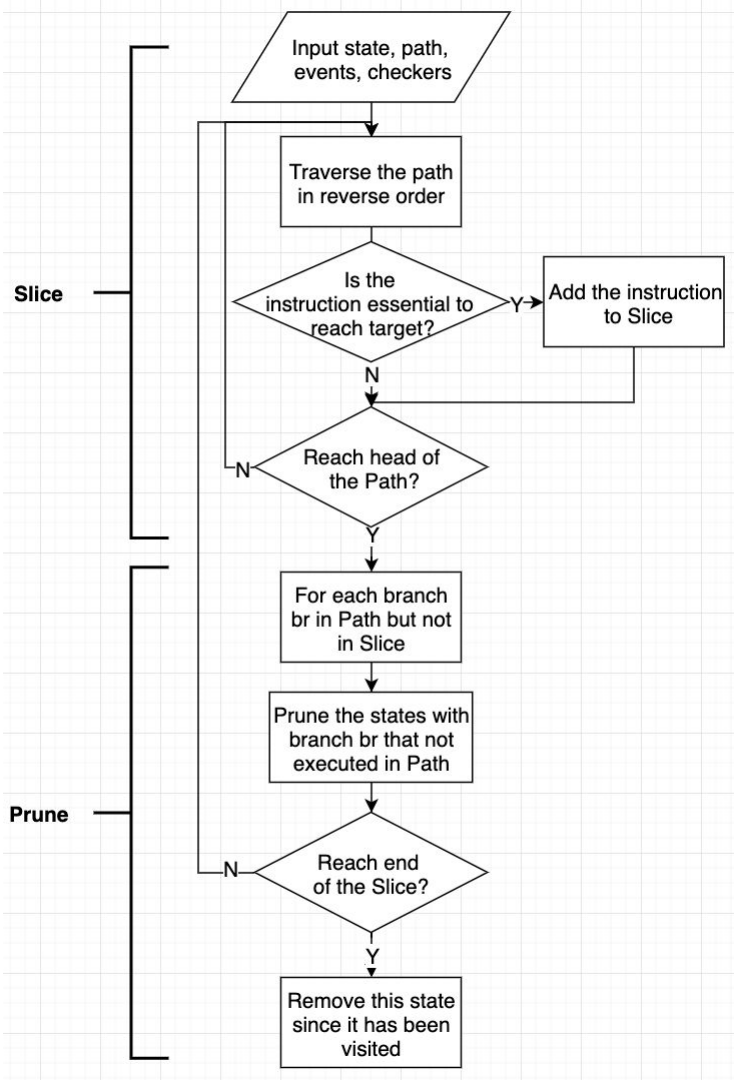
- **Traverse different execution paths**
  - **Traverse CFG, for each instructions**
    - Check violation of Rules
    - Check the end of path
  - If it is an violation
    - Report the violation
  - If it is the end of a path
    - Start slicing and pruning



Slice and Prune

- Path slicing
  - Backward searching the path.
  - Find data/control dependencies among instructions.
  - Extract in-slice instructions.
- Path Pruning
  - Remove symbolic states which are not in the slice.

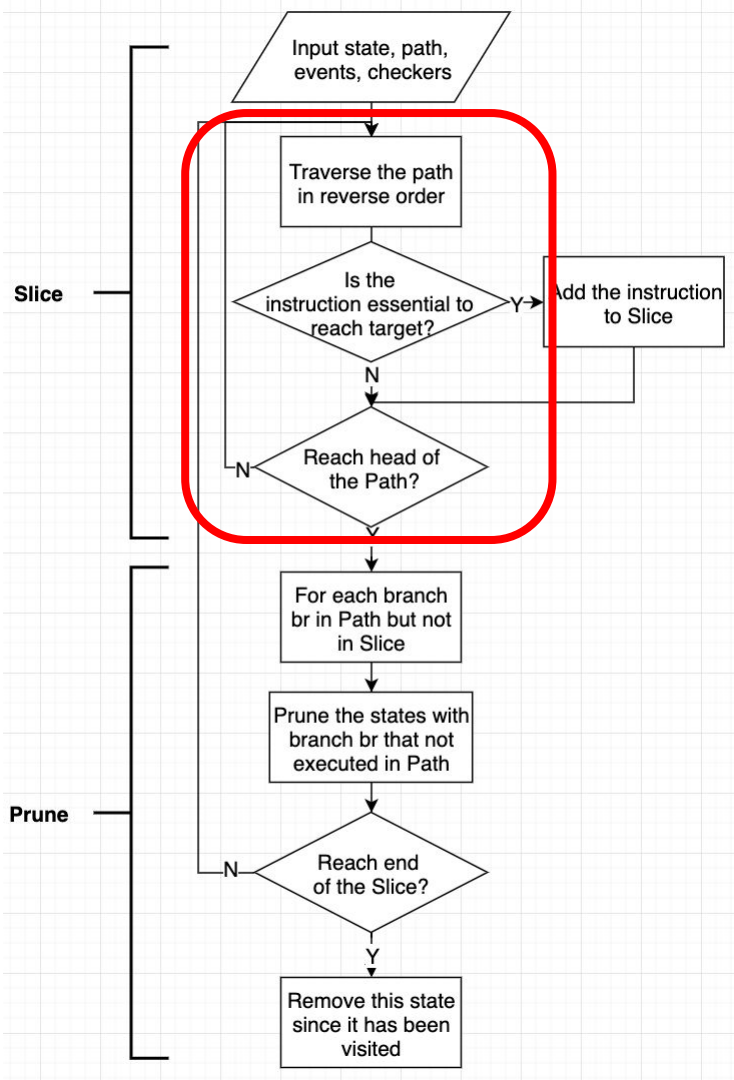
- Definition: In-slice Instruction
  - An event is dependent on the instruction.
  - (Branch Instr) Off-path branch contains other events.



Slice and Prune

- Path slicing
  - Backward searching the path.
  - Find data/control dependencies among instructions.
  - Extract in-slice instructions.
- Path Pruning
  - Remove symbolic states which are not in the slice.

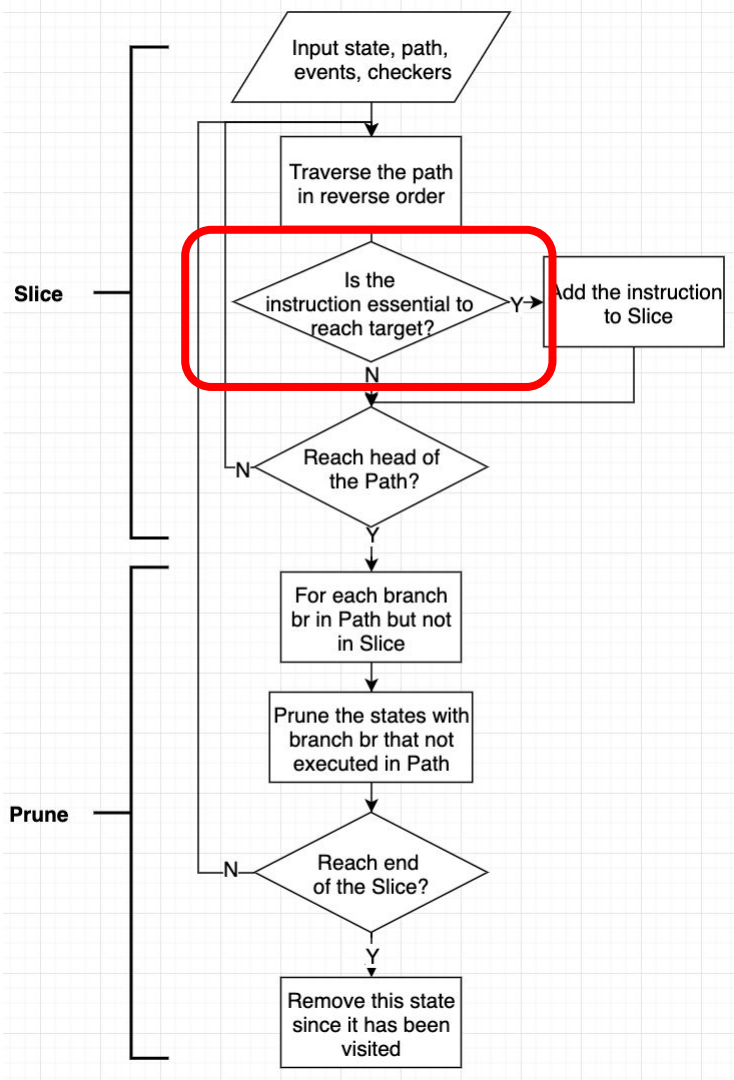
- Definition: In-slice Instruction
  - An event is dependent on the instruction.
  - (Branch Instr) Off-path branch contains other events.



Slice and Prune

- Path slicing
  - Backward searching the path.
  - Find data/control dependencies among instructions.
  - Extract in-slice instructions.
- Path Pruning
  - Remove symbolic states which are not in the slice.

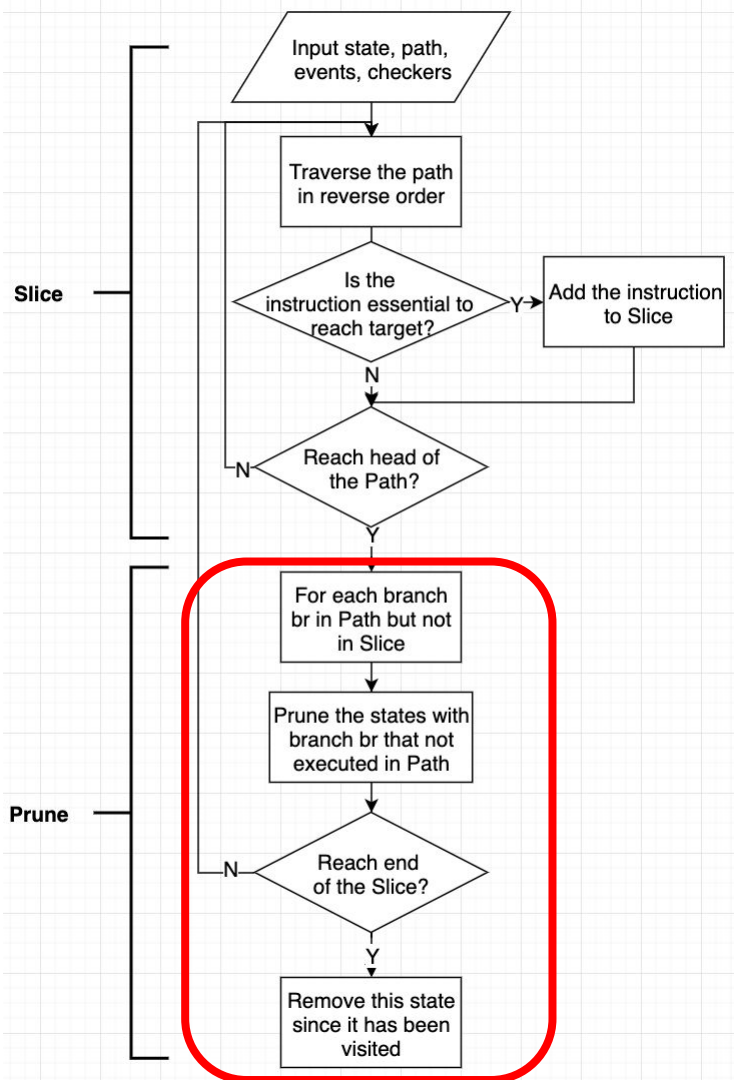
- Definition: In-slice Instruction
  - An event is dependent on the instruction.
  - (Branch Instr) Off-path branch contains other events.



Slice and Prune

- Path slicing
  - Backward searching the path.
  - Find data/control dependencies among instructions.
  - Extract in-slice instructions.
- Path Pruning
  - Remove symbolic states which are not in the slice.

- Definition: In-slice Instruction
  - An event is dependent on the instruction.
  - (Branch Instr) Off-path branch contains other events.



An example: *cat* in GNU *coreutils*

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {
15:       if(c < 32 && c != '\n') { // non-printable char
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

Determine input source

Handle non-printable char & print char

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {
15:       if(c < 32 && c != '\n') { // non-printable char
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

Open files

Close files

Determine input source

Handle non-printable char & print char

An example:
cat in GNU *coreutils*
Open-close rule:
Opened file must be closed

Baseline method: KLEE

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {
15:       if(c < 32 && c != '\n') { // non-printable char
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

Path explosion on the loop:

KLEE explores **698,116** paths in 1 hour.

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {
15:       if(c < 32 && c != '\n') { // non-printable char
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

Apply WOODPECKER

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)                                    True
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file           True
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;                            False
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {              True → False (Second iteration)
15:       if(c < 32 && c != '\n') { // non-printable char    False
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)               True
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);                             False
24:   return 0;
25: }
```

Apply WOODPECKER

1. Find a path between *entry* and *exit* with search algorithm.

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)                                   True
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file          True
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;                           False
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {             True → False (Second iteration)
15:       if(c < 32 && c != '\n') { // non-printable char   False
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)              True
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);                            False
24:   return 0;
25: }
```

Apply WOODPECKER

1. Find a path between *entry* and *exit*.
2. Path slicing

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)                                   True
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file          True
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;                           False
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {   True → False
15:       if(c < 32 && c != '\n') { // non-printable char   False
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)              True
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

Apply WOODPECKER

3. Pruning

Check the other branches of branch instructions **not** in path slicing set.

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)                                   True
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file          True
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;                           False
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {   True → False
15:       if(c < 32 && c != '\n') { // non-printable char   False
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)              True
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

Apply WOODPECKER

3. Pruning

```
3:        argind = 1;
4:        infile = "-";
6:true    argind < argc
7:        infile = argv[argind];
8:true    strcmp(infile, "-")
9:        input_desc = fopen(infile);
12:       if(input_desk) continue;
~~14:true   (c=fgetc(input_desc)) != EOF~~
~~15:false  c<32 && c!= '\n'~~
~~19:       putchar(c);~~
~~14:false  (c=fgetc(input_desc)) != EOF~~
21:true   infile[0]!='-' || infile[1] != 0
22:       fclose(input_desc);
23:false  ++argind < argc
~~24:       return 0;~~
```

Concerning events

Pruned instructions

Instructions explored by WOODPECKER

Evaluation

- Path verification efficiency. Comparison with **KLEE.**
- Rule violation detection.
- Cost of pruning.

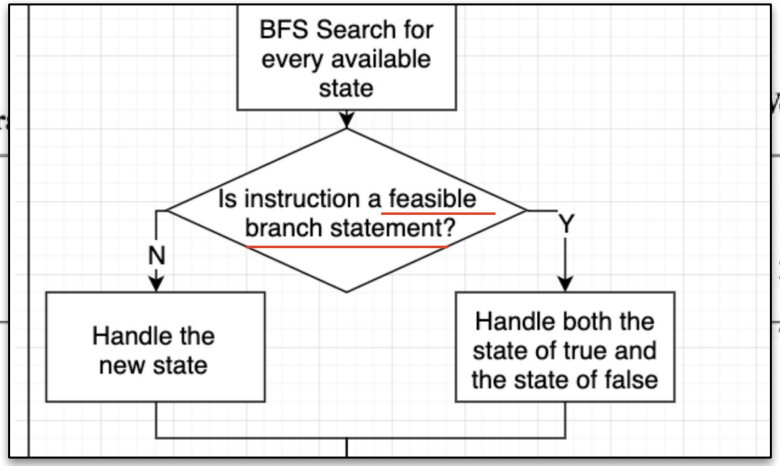
Evaluation

Checkers	Lines of Code	Programs Checked	Programs Verified		Relevant Paths Verified		Redundant Paths	
			WOODPECKER	KLEE	WOODPECKER	KLEE	WOODPECKER	KLEE
Assertion	102	57	13	3	195,268	45,763	69,795	195,178
Memory leak	399	103	32	7	1,024,676	176,475	451,836	1,657,721
Open-close	211	72	19	4	528,676	82,883	203,407	512,439
File access	344	120	40	12	1,694,393	377,181	496,651	2,141,111
Data loss	533	35	7	7	132,136	89,779	22,996	117,225
Total	1,589	387	111	33	3,575,149	772,081	1,244,685	4,623,674

Table 1: *Summary of verification results.*

Evaluation

Checkers	Lines of Code	Progr	Verified KLEE	Redundant Paths WOODPECKER	Redundant Paths KLEE
Assertion	102		45,763	69,795	195,178
Memory leak	399		176,475	451,836	1,657,721
Open-close	211		82,883	203,407	512,439
File access	344		377,181	496,651	2,141,111
Data loss	533		89,779	22,996	117,225
Total	1,589		772,081	1,244,685	4,623,674



Evaluation

- ***search efficiency****:* the percentage of relevant paths explored over all paths ever forked.

Time Limit	Programs Verified			Paths Verified		
	W	K	W- K	W	K	W/K
1 hour	73	7	67	2,776,499	532,222	5.2
2 hours	104	31	73	6,933,817	662,558	10.5
4 hours	112	39	73	14,437,294	847,621	17.0

Programs	mem leak	open-close	data loss
`coreutils`	40	13	0
`shadow`	11	5	1
`tar`	4	0	0
`sed`	3	0	0
`CVS`	3	1	2
`git`	19	4	7
Total	80	23	10

Evaluation

- *search efficiency:* the per

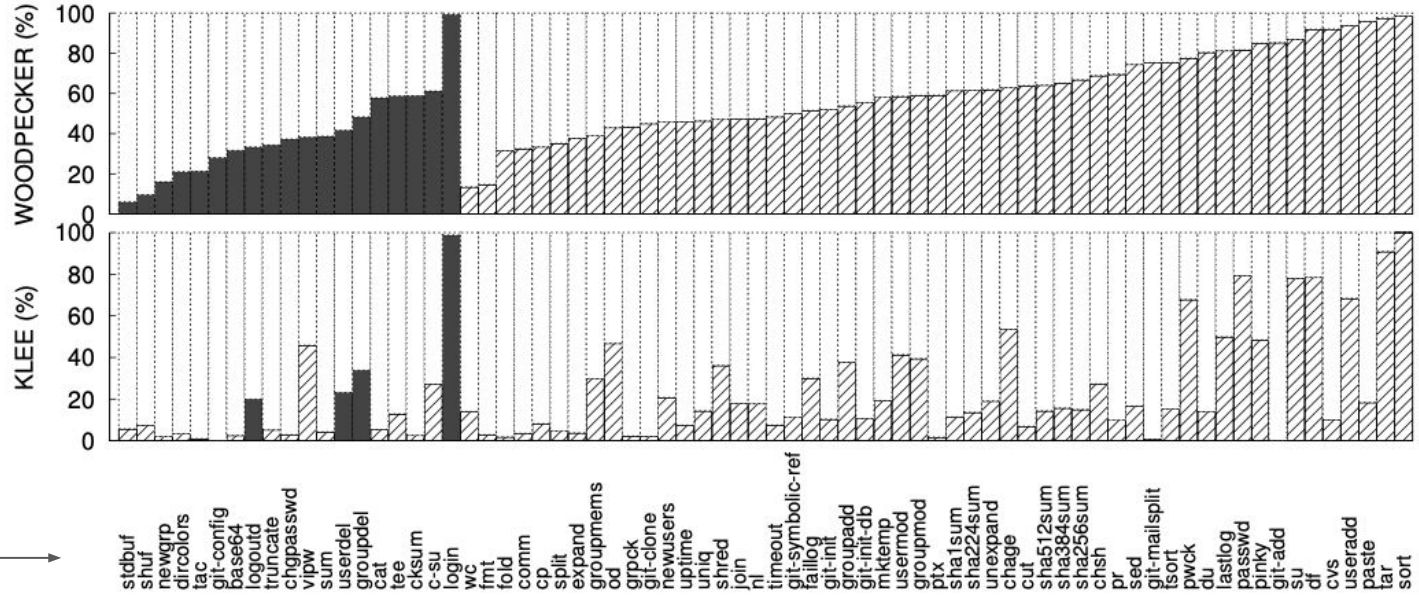
Time Limit	Programs Verified W	K	W- K
1 hour	73	7	67
2 hours	104	31	73
4 hours	112	39	73

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {
15:       if(c < 32 && c != '\n') { // non-printable char
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

em leak	open-close	data loss
40	13	0
11	5	1
4	0	0
3	0	0
3	1	2
19	4	7
80	23	10

Evaluation

Percentage of Relevant Path Explored Among All



Programs run against

Figure 8: *Search efficiency with the open-close checker.* WOODPECKER's median search efficiency of the hatched bars is 59.2%, whereas KLEE's is 15.3%.

Limitation

Overhead of pruning and alias analysis

Programs	Pruning (%)	Alias (%)
`coreutils`	0.82	0.49
`shadow`	1.98	0.11
`tar`	4.59	5.87
`sed`	0.69	0.50
`CVS`	4.36	4.64
`git`	8.99	11.71

Related Work

- Static Analysis
  - Check rules on (potentially infeasible) program paths.
  - Can aggressively trade off soundness for low false positive.
- Symbolic execution
  - Error detections, tests generation, buggy execution reproduce, path verification etc.
  - Complementary to previous symbolic execution.
  - Leverage power search heuristics in existing systems.
- Program Slicing
  - Dynamic - Static
  - Path slicing

Strengths & Weakness

- Strengths
  - Designed a light-weight symbolic execution schema for verifying code rules by leveraging path slicing.
  - Outperforms KLEE in efficiency.

- Weakness
  - Absent heuristics
    - The choice of search mechanisms (i.e., DFS/BFS)
  - Not supporting simultaneous multi-rules checking
    - WoodPecker should be extended to supported checking multiple rules to avoid redundant work.