Combining Static Analysis Error Traces with Dynamic Symbolic Execution (Experience Paper)

Frank Busse • Pritam M. Gharat • Cristian Cadar • Alastair F. Donaldson
Software Reliability Group • Multicore Programming Group
off-the-shelf static analyser

traces

off-the-shelf symbolic executor

concrete inputs for true positives

developers
What we did:

- combined traces for **two bug classes** from **two static analysis engines** with **one symbolic execution engine** in a particular way

What we didn't:

- compare **static analysis vs. (dynamic) symbolic execution**
- **generalise results** to any combination of static analysis with symbolic execution
- **benchmark static analysers**
```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H') {
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```
Example

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H') {
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
            --in1;
        }
    }

    int result = *p0;
    free(p0);
    return result;
}
```
```c
int main (int argc, char *argv[]) {
    uint8_t *in1 = argv[1][0];
    uint8_t *in2 = argv[1][1];
    uint8_t *in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```
Example

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];
    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;
    while (in1 > 'H' - 2) {
        if (in1 == 'H') {
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }
    int result = *p0;
    free(p0);
    return result;
}
```
Example Static Analysis Traces

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == 'l')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```

```bash
$ scan-build clang example.c
example.c:23:15: warning: Use of memory after it is freed [unix.Malloc]
    int result = *p0;
          ^~~
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
```
Example

Static Analysis Traces

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```
Infeasible Traces

if (do_sandbox) {
    shadow_node = make_array();
    sub = make_string(argv0, strlen(argv0));
    val = make_number(0.0);
    assoc_set(shadow_node, sub, val);
}

for (i = argc0, j = 1; i < argc; i++, j++) {
    sub = make_string(AWKNUM j);
    val = make_string(argv[i], strlen(argv[i]));
    val->flags = USER_INPUT;
    assoc_set(ARGV_node, sub, val);
}

if (do_sandbox) {

    Assuming the condition is true →

    Taking false branch →

    Loop condition is true. Entering loop body →

    Assuming the condition is true →
Example

Dynamic Symbolic Execution

```
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H') {
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        }
    --in1;
}

    int result = *p0;
    free(p0);
    return result;
}
```

> klee [...] example.bc --sym-arg 3
KLEE: Using STP solver backend
KLEE: ERROR: example.c:23: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 23361
KLEE: done: completed paths = 554
KLEE: done: generated tests = 4
```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H') {
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!'')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```
Example

Dynamic Symbolic Execution

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H')
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!')
                    free(p1);
            }
        --in1;
    }

    int result = *p0;
    free(p0);
    return result;
}
```
Example
Instrumentation

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (in1 > 'H' - 2) {
        if (in1 == 'H') {
            if (in2 == 'i') {
                p1 = p0;
                if (in3 == '!'')
                    free(p1);
            }
            --in1;
        }
    }

    int result = *p0;
    free(p0);
    return result;
}
```
Example
Instrumentation

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));  \[1\]
    *p0 = in1;

    while (in1 > 'H' - 2) { \[2\] \[3\] \[4\] \[8\]
        if (in1 == 'H') { \[5\]
            if (in2 == 'i') { \[6\]
                p1 = p0;
                if (in3 == '!') {
                    free(p1);
                }
            } --in1;
        }
    }

    int result = *p0; \[9\]
    free(p0);
    return result;
}
```
Constraint Enforcement

Ignore noop
Try add constraint if feasible
Require constraint has to hold
Example
Constraint Enforcement

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (INSTR_LINE_13(in1 > 'H' - 2)) {
        if (INSTR_LINE_14(in1 == 'H')) {
            if (INSTR_LINE_15(in2 == 'i')) {
                p1 = p0;
                if (INSTR_LINE_17(in3 == '!'))
                    free(p1);
            }
        }
        --in1;
    }

    int result = *p0; INSTR_LINE_23();
    free(p0);
    return result;
}
```

> klee [...] example.bc --sym-arg 3
KLEE: Using STP solver backend
KLEE: ERROR: example.c:23: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 23361
KLEE: done: completed paths = 554
KLEE: done: generated tests = 4

> klee [...] example.bc --sym-arg 3
KLEE: Using STP solver backend
KLEE: ERROR: example_run.c:25: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 1230
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
Example
Constraint Enforcement

```c
int main (int argc, char *argv[]) {
    uint8_t in1 = argv[1][0];
    uint8_t in2 = argv[1][1];
    uint8_t in3 = argv[1][2];

    uint8_t *p0, *p1;
    p0 = malloc(sizeof(uint8_t));
    *p0 = in1;

    while (INSTR_LINE_13(in1 > 'H' - 2)) {
        if (INSTR_LINE_14(in1 == 'H'))
            if (INSTR_LINE_15(in2 == 'i')) {
                p1 = p0;
                if (INSTR_LINE_17(in3 == '!'))
                    free(p1);
            }
    }

    int result = *p0; INSTR_LINE_23();
    free(p0);
    return result;
}
```
Targeted Search Heuristic

- drives execution engine towards instrumented lines
- skips unreachable steps
- terminates states that can't reach final step
- prioritises states that
  - reached more steps
  - are closer to next step
Targeted Search Heuristic

- drives execution engine **towards** instrumented lines
- **skips** unreachable steps
- **terminates** states that can't reach final step
- **prioritises** states that
  - reached **more steps**
  - are **closer to next step**
Targeted Search Heuristic

- drives execution engine towards instrumented lines
- skips unreachable steps
- terminates states that can't reach final step
- prioritises states that
  - reached more steps
  - are closer to next step

Algorithm 1: Targeted search heuristic

```plaintext
Data: activePathIDs : Step → {PathID}
Data: states : Step × PathID → (State)
Data: instructionPathIDs : Instruction × Step → pathID
Data: maxActiveStep: maximum step number among states
Data: maxStep: maximum step number in program

Function update(currentState, newStates, terminatedStates):
  1. updateCurrent (currentState)
  2. foreach state : newStates do
     3. insert (state)
  3. foreach state : terminatedStates do
     4. remove (state)

Function insert(state):
  1. state.distance ← computeDistance(state)
  2. if state.distance = ∞ then
     3. while state.distance = ∞ ∧ state.lastStep < maxStep do
        4. state.lastStep ← state.lastStep + 1
        5. state.distance ← computeDistance(state)
  6. state.pathID ←
     7. instructionPathIDs[state.pc][state.lastStep]
  8. if state.distance = ∞ then
     9. terminate(state)
  10. else
     11. states[state.lastStep][state.pathID].add(state)
     12. activePathIDs[state.lastStep].add(state.pathID)

Function select() → State:
  1. nextPathID ←
     2. activePathIDs[maxActiveStep].selectRoundRobin()
  3. candidates ←
     4. states[maxActiveStep][nextPathID].selectByDistance()
  5. return candidates.pickRandomly()
```
Evaluation

We investigated

- historical SA bug reports
- CoREBench¹
- 25 applications/suites > 7 yrs old

In short

- (almost) no historical reports
- known bugs not found
- true positives trivial and/or bug class not supported by KLEE

---

¹ https://www.comp.nus.edu.sg/~release/corebench/

---

<table>
<thead>
<tr>
<th>Application</th>
<th>Released</th>
<th>Relevant reports</th>
<th>False positives</th>
<th>True positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>APR</td>
<td>1.5.2</td>
<td>8 2</td>
<td>8 2</td>
<td>0 0</td>
</tr>
<tr>
<td>flex</td>
<td>2.5.39</td>
<td>13 17</td>
<td>12 7</td>
<td>1 10</td>
</tr>
<tr>
<td>awk</td>
<td>4.1.2</td>
<td>124 70</td>
<td>20 20</td>
<td>0 0</td>
</tr>
<tr>
<td>bc</td>
<td>1.06</td>
<td>11 0</td>
<td>11 0</td>
<td>0 0</td>
</tr>
<tr>
<td>Binutils</td>
<td>2.25.1</td>
<td>0 38</td>
<td>0 14</td>
<td>0 6</td>
</tr>
<tr>
<td>combine</td>
<td>0.40</td>
<td>1 10</td>
<td>1 10</td>
<td>0 0</td>
</tr>
<tr>
<td>Coreutils</td>
<td>8.24</td>
<td>25 5</td>
<td>20 5</td>
<td>0 0</td>
</tr>
<tr>
<td>datamash</td>
<td>1.0.6</td>
<td>0 1</td>
<td>0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>Diffutils</td>
<td>3.3</td>
<td>6 0</td>
<td>6 0</td>
<td>0 0</td>
</tr>
<tr>
<td>Findutils</td>
<td>4.4.2</td>
<td>6 2</td>
<td>6 1</td>
<td>0 1</td>
</tr>
<tr>
<td>grep</td>
<td>2.2.1</td>
<td>20 8</td>
<td>20 8</td>
<td>0 0</td>
</tr>
<tr>
<td>Gzip</td>
<td>1.6</td>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>Libtasn1</td>
<td>4.5</td>
<td>1 4</td>
<td>1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>M4</td>
<td>1.4.17</td>
<td>9 2</td>
<td>8 2</td>
<td>1 0</td>
</tr>
<tr>
<td>Make</td>
<td>4.1</td>
<td>3 2</td>
<td>3 2</td>
<td>0 0</td>
</tr>
<tr>
<td>oSIP</td>
<td>4.1.0</td>
<td>1 6</td>
<td>1 6</td>
<td>0 0</td>
</tr>
<tr>
<td>sed</td>
<td>4.2</td>
<td>6 7</td>
<td>3 7</td>
<td>3 0</td>
</tr>
<tr>
<td>Trueprint</td>
<td>5.4</td>
<td>0 7</td>
<td>0 6</td>
<td>1 0</td>
</tr>
<tr>
<td>ImageMagick</td>
<td>6.9.4-8</td>
<td>10 11</td>
<td>10 3</td>
<td>0 8</td>
</tr>
<tr>
<td>JasPer</td>
<td>1.900.1</td>
<td>9 3</td>
<td>9 1</td>
<td>0 2</td>
</tr>
<tr>
<td>libjpeg</td>
<td>9a</td>
<td>17 2</td>
<td>17 2</td>
<td>0 0</td>
</tr>
<tr>
<td>LibTIFF</td>
<td>3.9.7</td>
<td>6 12</td>
<td>6 3</td>
<td>0 9</td>
</tr>
<tr>
<td>libxml2</td>
<td>2.9.2</td>
<td>33 91</td>
<td>20 20</td>
<td>0 0</td>
</tr>
<tr>
<td>tcpdump</td>
<td>4.7.4</td>
<td>0 2</td>
<td>0 0</td>
<td>0 2</td>
</tr>
<tr>
<td>Vorbis Tools</td>
<td>1.4.0</td>
<td>1 19</td>
<td>1 1</td>
<td>0 18</td>
</tr>
</tbody>
</table>
Evaluation

We investigated

- historical SA bug reports
- CoREBench¹
- 25 applications/suites > 7yrs old

In short

- (almost) no historical reports
- known bugs not found
- true positives trivial and/or bug class not supported by KLEE

→ We had to artificially inject bugs.

<table>
<thead>
<tr>
<th>relevant reports</th>
<th>true/false positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSA</td>
<td>Infer</td>
</tr>
<tr>
<td>311</td>
<td>322</td>
</tr>
<tr>
<td>CSA</td>
<td>Infer</td>
</tr>
<tr>
<td>6/183</td>
<td>60/122</td>
</tr>
</tbody>
</table>

Infer reports many missing NULL-checks for malloc(), strdup(), localtime(), …

¹ https://www.comp.nus.edu.sg/~release/corebench/
Bug Injection

- **two bug types**
  - null-pointer dereferences
  - use-after-free errors
- **1–4 events** along path
- only in hard to reach instructions (KLEE needs more than 10min to cover instruction)
- 10 applications from Coreutils 8.31
Bug Injection

- **two bug types**
  - null-pointer dereferences
  - use-after-free errors
- **1–4 events** along path
- only in hard to reach instructions (KLEE needs more than 10min to cover instruction)
- 10 applications from Coreutils 8.31

```c
int _i = 1, *xtmp = &_i, *ytmp = &_i;
xtmp = (int *)malloc(sizeof(int));
ytmp = xtmp; // 0-2 aliases for 1-4 event bugs
free(xtmp); printf("%d", *ytmp); // use-after-free
```
Bug Injection

- 297 one-event bugs
- 632 two-event bugs
- 478 three-event bugs
- 357 four-event bugs

55 bugs for further evaluation

Typical trace lengths:
- **CSA** 10–20 steps (max. 55)
- **Infer** 1–5 steps
Results
Instrumented Code

- targeted heuristic finds more bugs in less time
- require performs worst
- try only slightly better than ignore
- intermediate steps rarely beneficial
The static analysis error traces in our experiments in general do not add (m)any benefits when combined with targeted symbolic execution.
Targeted KLEE on Instrumented Code

Figure 4: Total analysis time for KLEE for the injected bugs across all instrumentation strategies and search heuristics, as well as the Ignore-TargetLast special case and a Portfolio strategy. Numbers of bugs detected are shown above each error bar.