Triggering Deep Vulnerabilities Using Symbolic Execution

Dan Caselden, Alex Bazhanyuk, *Mathias Payer*, Stephen McCamant, Dawn Song, and many other awesome researchers, coders, and reverse engineers in the BitBlaze group at UC Berkeley

* images taken from original “Alice in Wonderland”

Preconditions

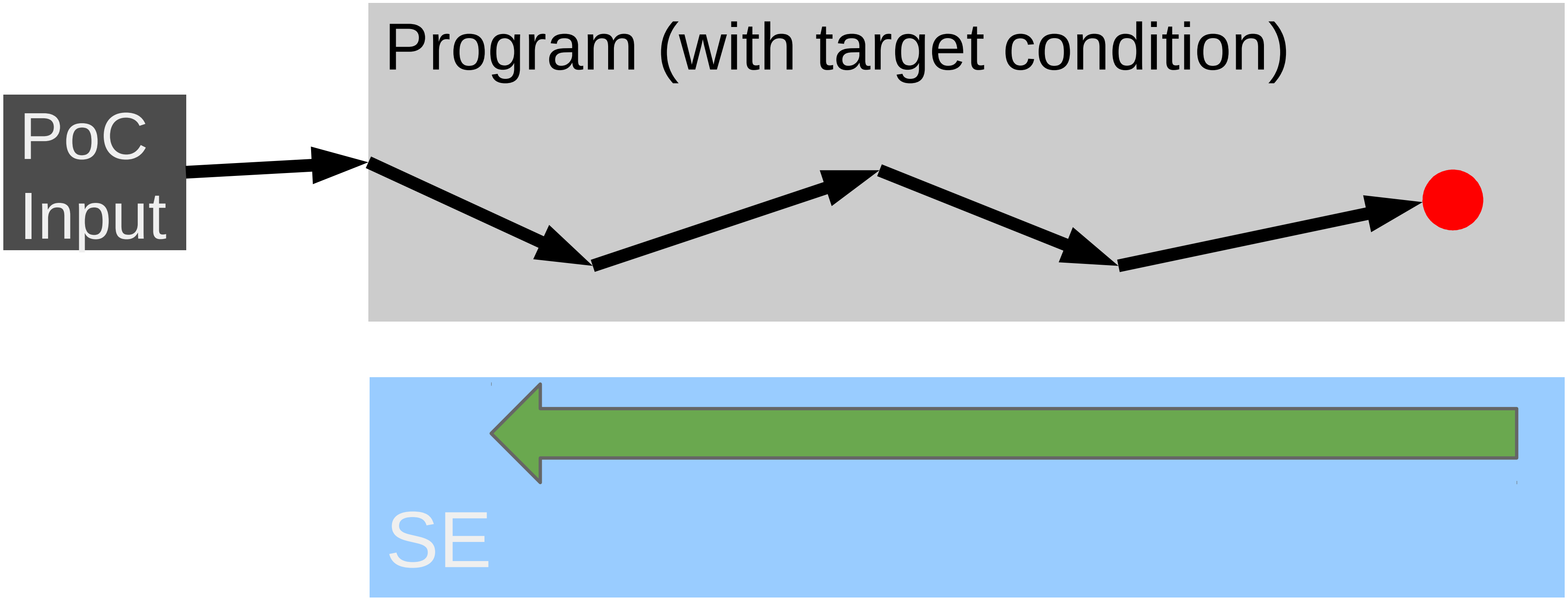
Finding bugs and crashes is easy

- Fuzzing, Bounded Model Checking, test cases

Exploit generation is hard

- Trigger for vulnerability?
- Input transformations?

Setup


PoC Input

Program (with target condition)

SE

Road map

Motivation

Definition and tools

State explosion

Scaling up

Divide and conquer

Binary analysis

The end

What is Symbolic Execution?

An abstract interpretation of code

- Symbolic values, not concrete

Agnostic to concrete values

- Values turn into formulas
- Constraints concretize formulas

Finds concrete input

- Triggers “***interesting***” condition

Using Symbolic Execution

Define set of conditions at code locations

- Symbolic Execution determines triggering input

Testing: finding bugs in applications

- Infer pre/post conditions and add assertions
- Use symbolic execution to negate conditions

Exploit generation: generate PoC input

- Vulnerability condition is predefined

Symbolic Execution Tools

FuzzBALL

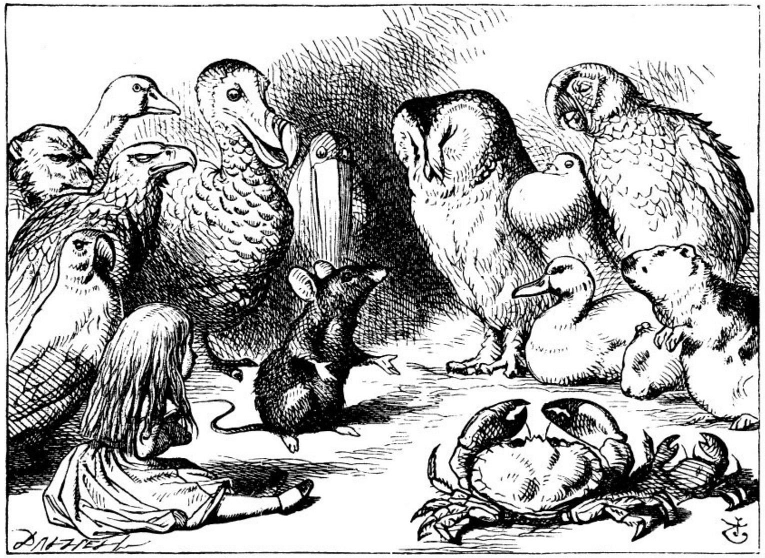
- PoC exploits for given vulnerability conditions
- http://bitblaze.cs.berkeley.edu/fuzzball.html

S2E: Selective Symbolic Execution

- Automatic testing of binary code
- http://dslab.epfl.ch/proj/s2e

KLEE

- Bug finding in source code
- http://ccadar.github.io/klee/

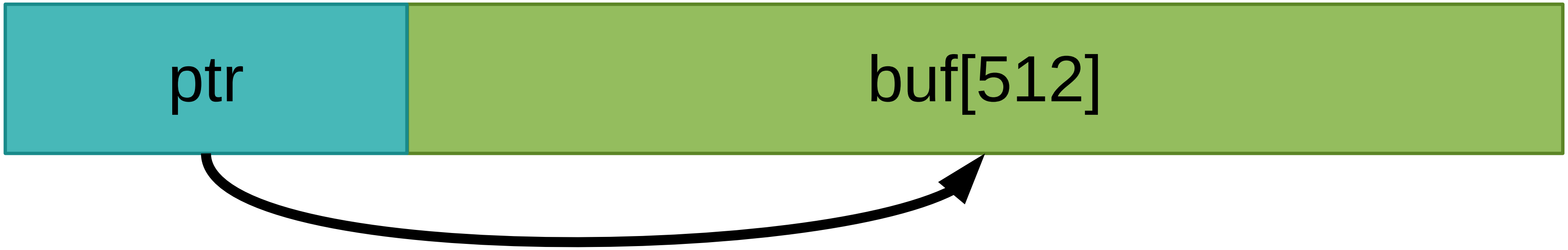


Example #1: Vortex Wargame*

```
#include <...>
void print(unsigned char *buf, int len); // print state (for debugging)

#define e(); if(((unsigned int)ptr & 0xff000000)==0xca000000){win();}

int main() {
    unsigned char buf[512];
    unsigned char *ptr = buf + (sizeof(buf)/2);
    unsigned int x;

    while((x = getchar()) != EOF) {
        switch(x) {
            case '\n': print(buf, sizeof(buf)); continue; break;
            case '\\': ptr--; break;
            default: e(); if(ptr > buf + sizeof(buf)) continue; ptr++[0] = x;
} } }
```

* http://www.overthewire.org/wargames/

Example #1: Vortex Wargame*



```
switch (input) {
        case '\n': debug()     // print debug information
        case '\': ptr--;       // decrement ptr
        default:
            if (ptr & 0xff000000 == 0xca000000) win();
            if (ptr < buf[len]) ptr++[0] = input;
}
```

* http://www.overthewire.org/wargames/

Example #1: Vortex Wargame*

p

Problem size: 3^n

```
switch (input) {
        case '\n': debug()      // print debug information
        case '\\': ptr--;       // decrement ptr
        default:
            if (ptr & 0xff000000 == 0xca00
            if (ptr < buf[len]) ptr++[0] = inp
}
```

Demo!

* http://www.overthewire.org/wargames/

Road map

Motivation

Definition and tools

State explosion

Scaling up

Divide and conquer

Binary analysis

The end

Does Symbolic Exec. scale?

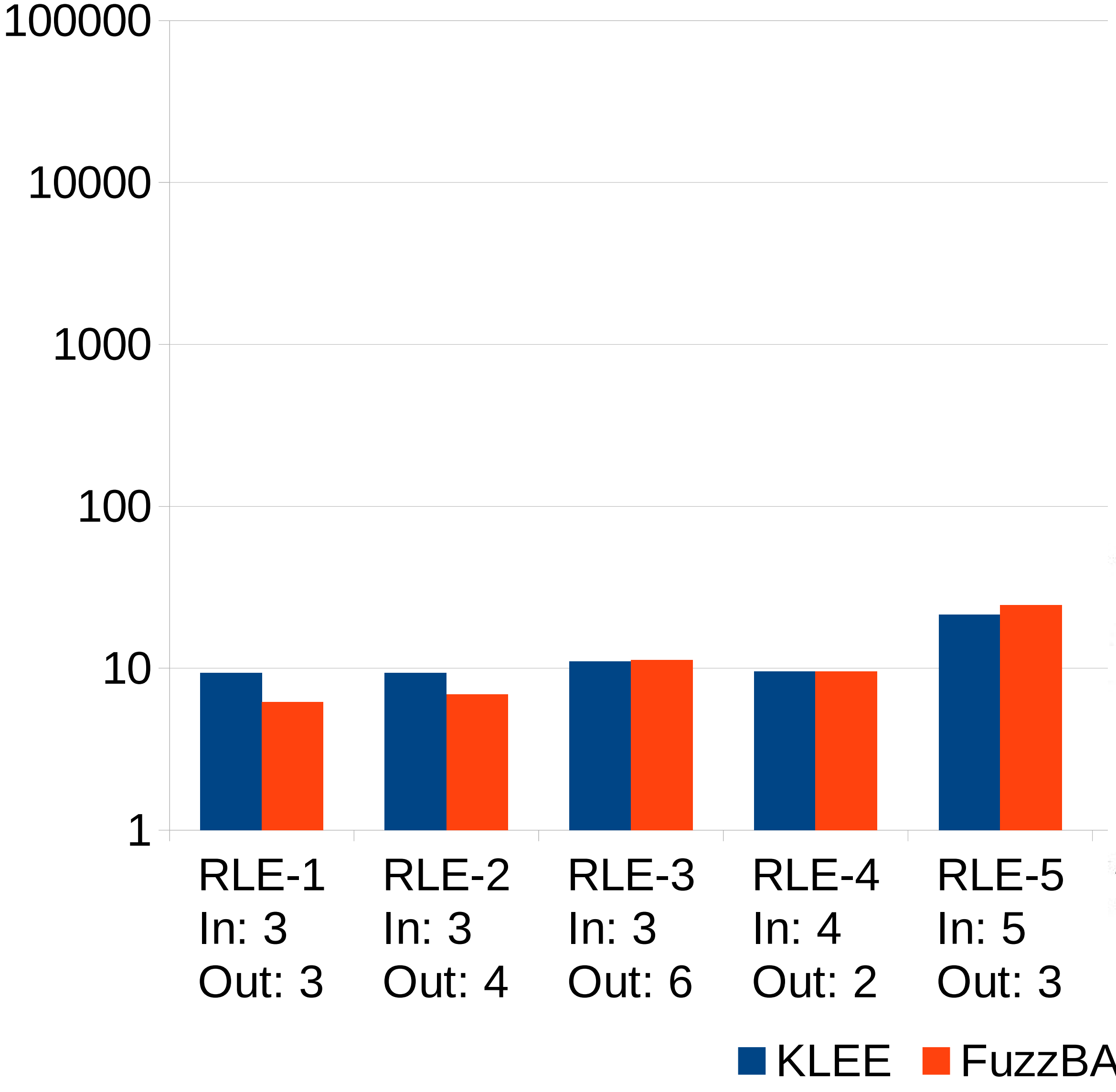
Run Length Encoding: compression

- Decode and expand input string
- Output buffer is given
- Symbolic Execution produces input
- Different input/output length

Evaluate performance of

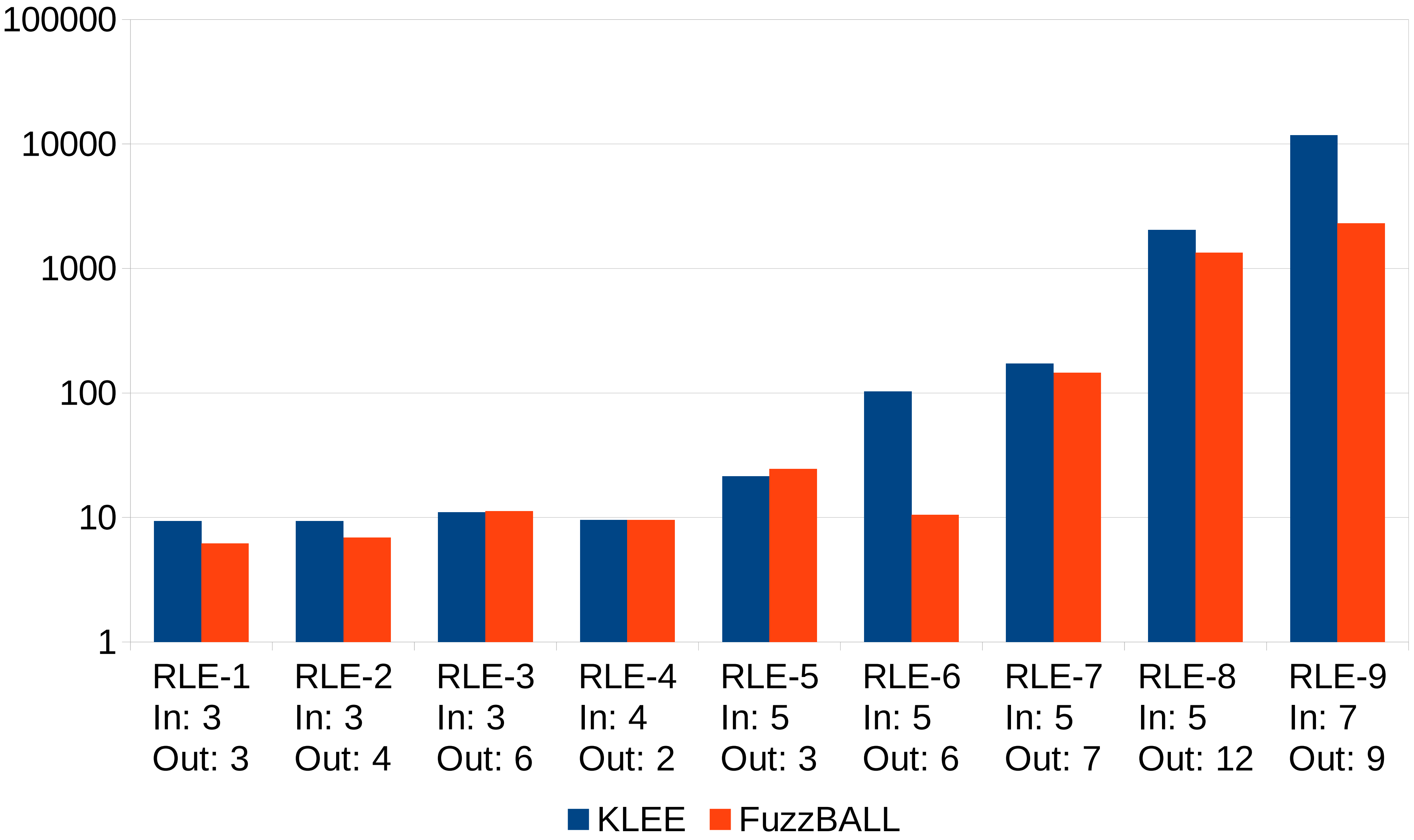
- KLEE
- FuzzBALL

RLE encoding: limitations*



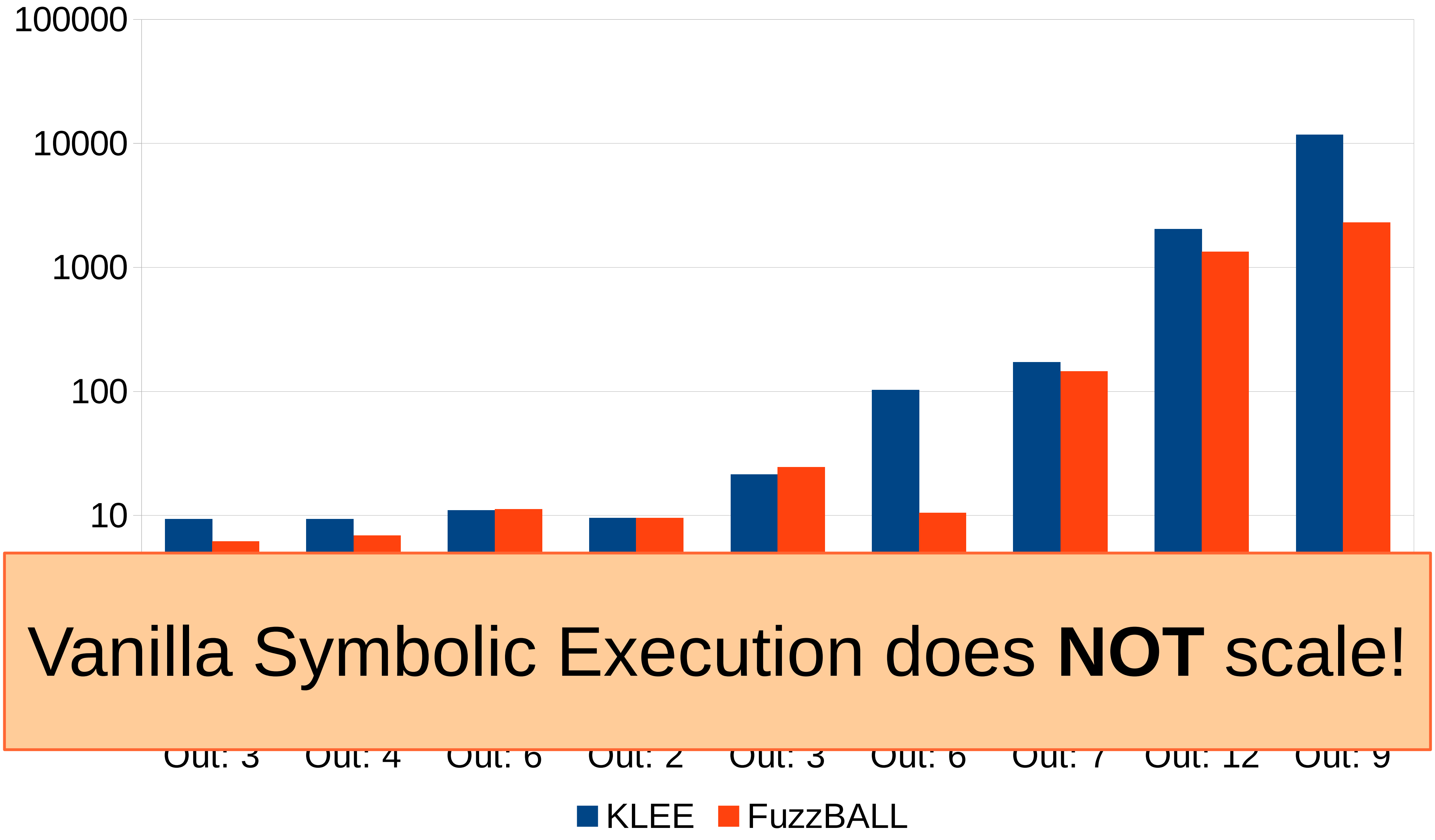
* Detailed results from TR Berkeley/EECS-2013-125

RLE encoding: limitations*



* Detailed results from TR Berkeley/EECS-2013-125

RLE encoding: limitations*



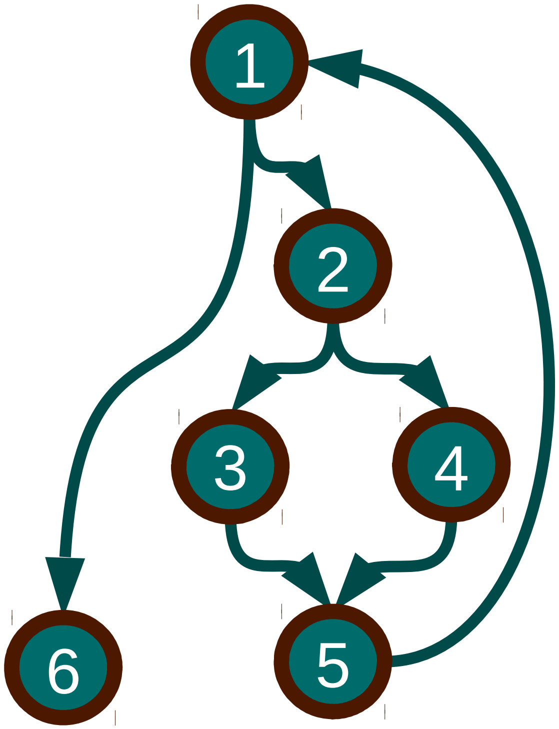
Vanilla Symbolic Execution does **NOT** scale!

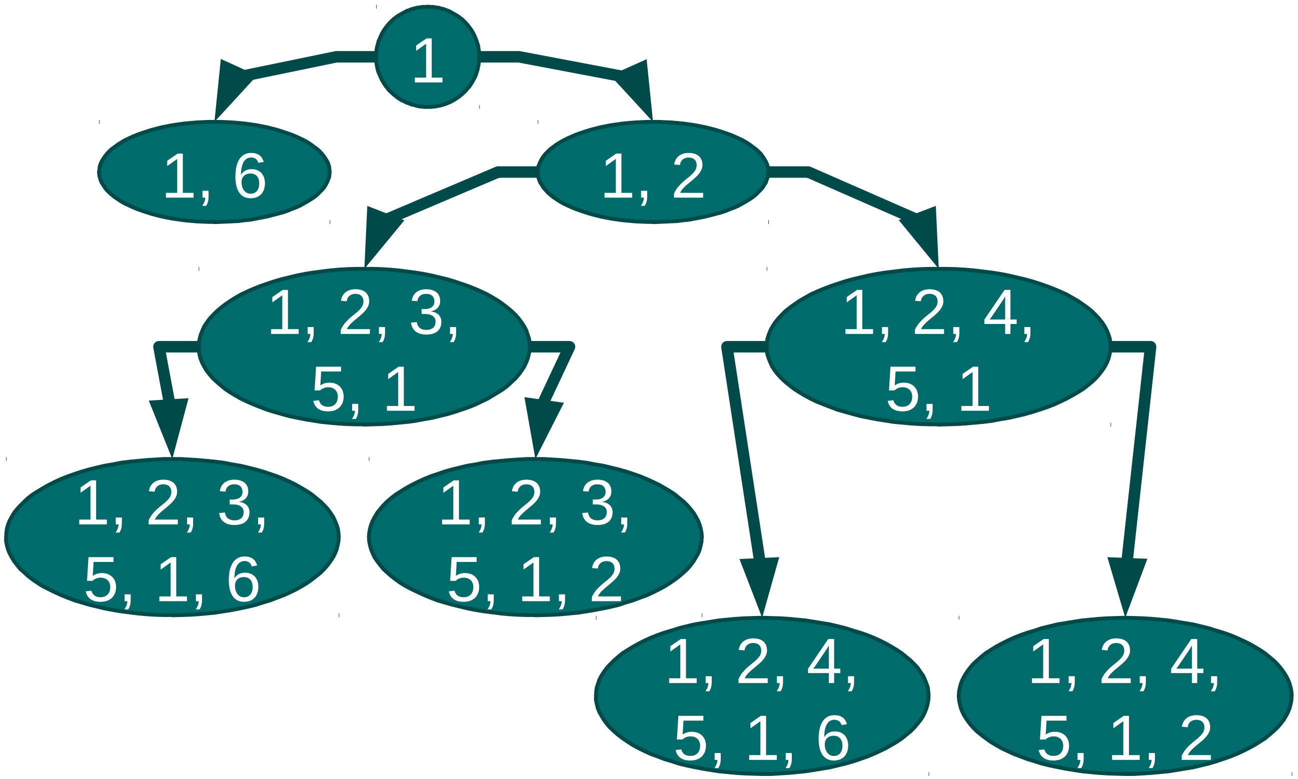
* Detailed results from TR Berkeley/EECS-2013-125

State explosion

At each decision point

- Number of paths doubles (fork)
- Updated or added constraints





Reasons for state explosion

Too much input/output data

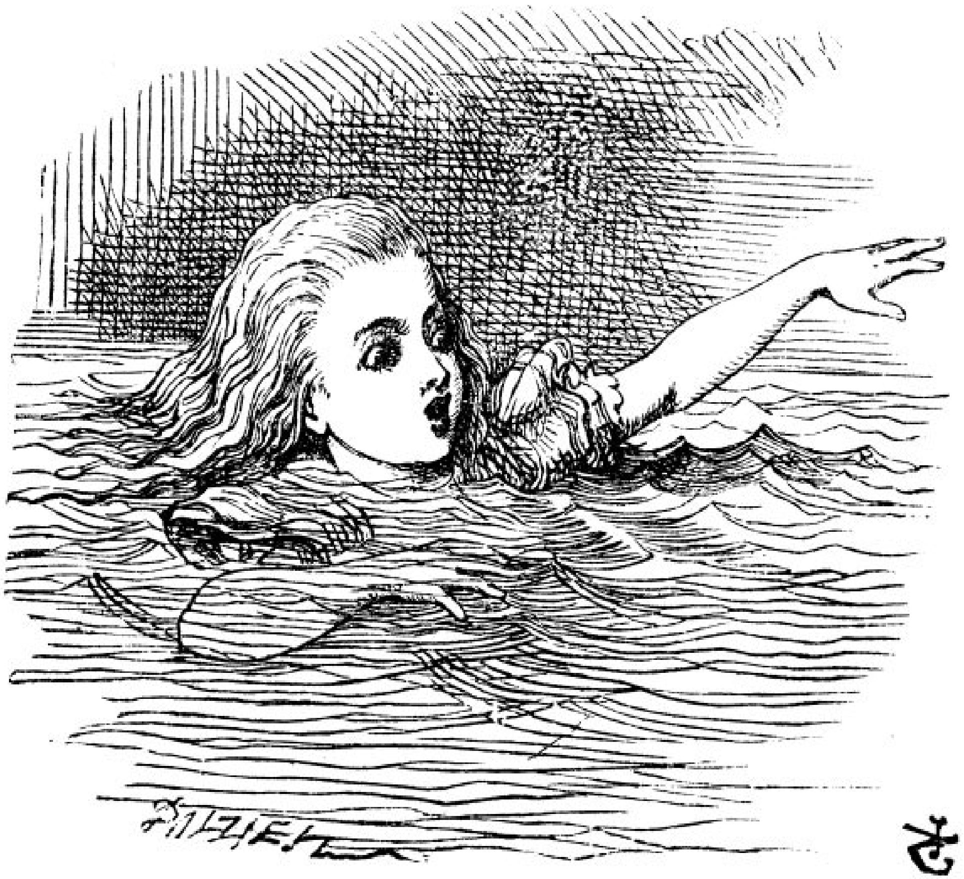
- Not much we can do about

Too much included state

- Limit symbolic state

Too much executed code

- Divide and conquer



Road map

Motivation

Definition and tools

State explosion

Scaling up

Divide and conquer

Binary analysis

The end

Interesting input sizes

<10 symbolic bytes

- Address, offset or pointer

20-80 symbolic bytes

- Shellcode, ROP chain

>200 symbolic bytes

- Shellcode plus data, long ROP chains
- Complete data structures

Heuristics to the rescue

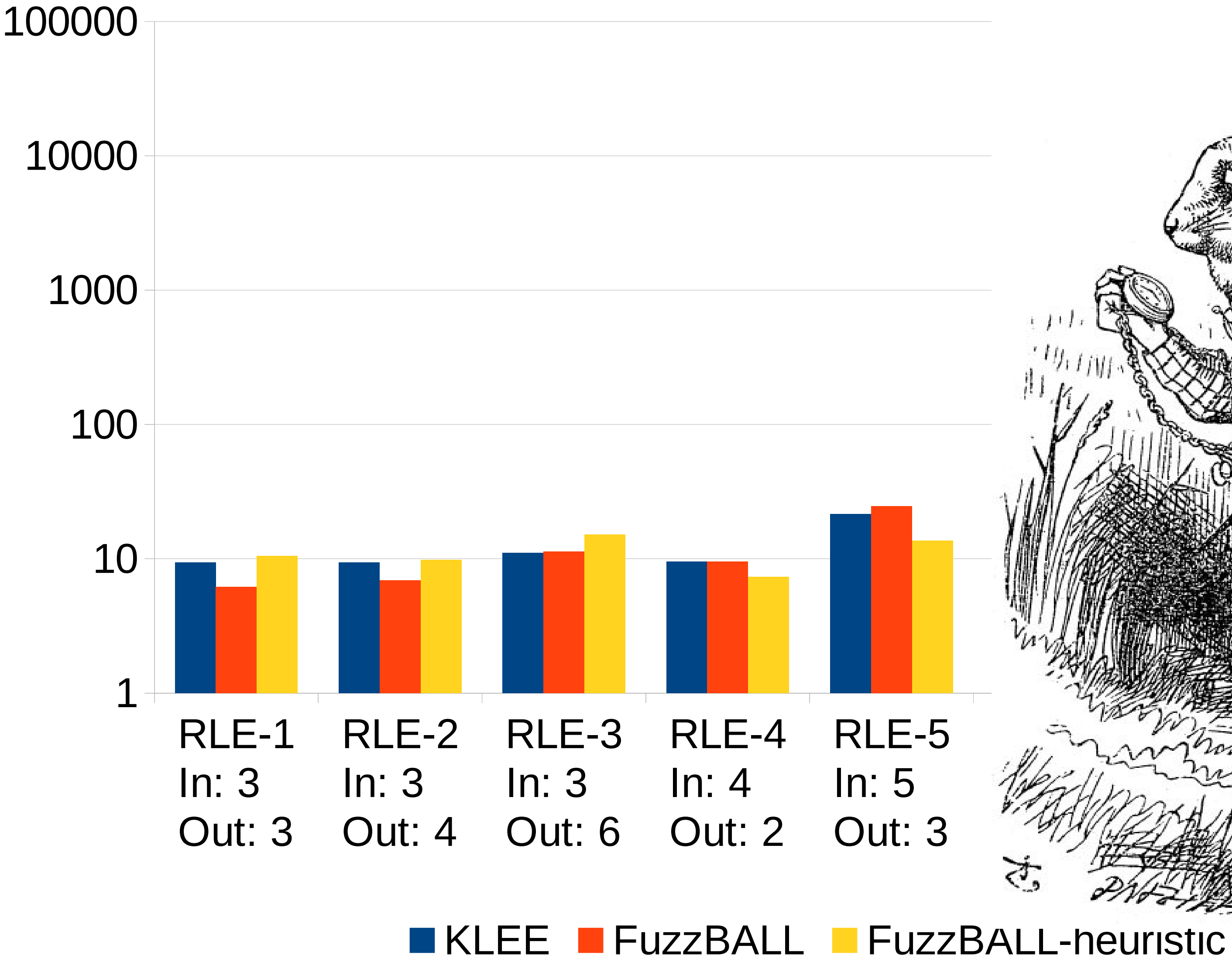
Assume properties for transformations

- Surjectivity: there exists a pre-image
- Sequentiality: output is never revoked
- Streaming: bounded transformation state

Encoded heuristics

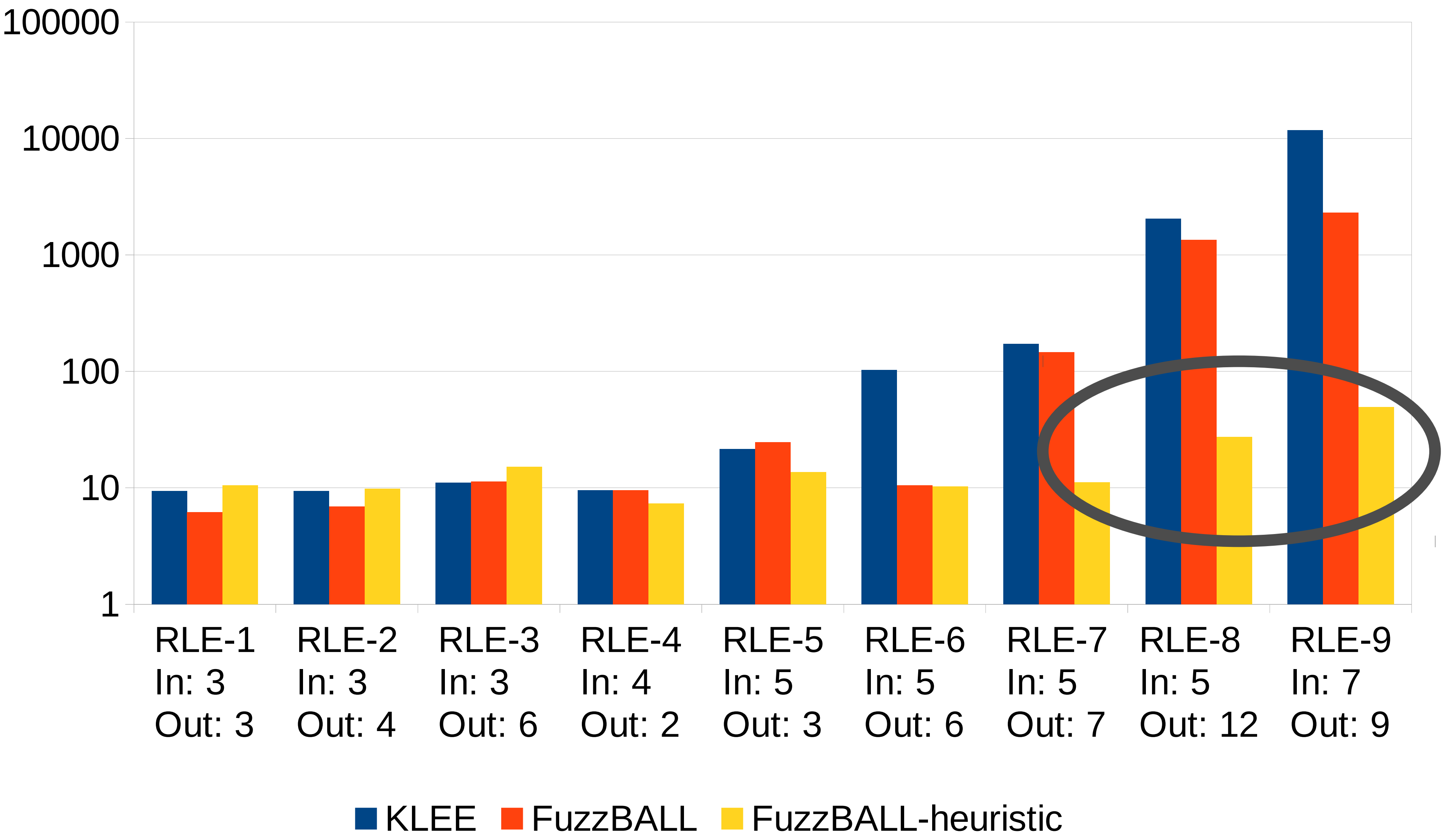
- Prune early, prune often if target unreachable
- Be greedy, prioritize paths that maximize output
- Optimize array accesses

RLE encoding: heuristics*



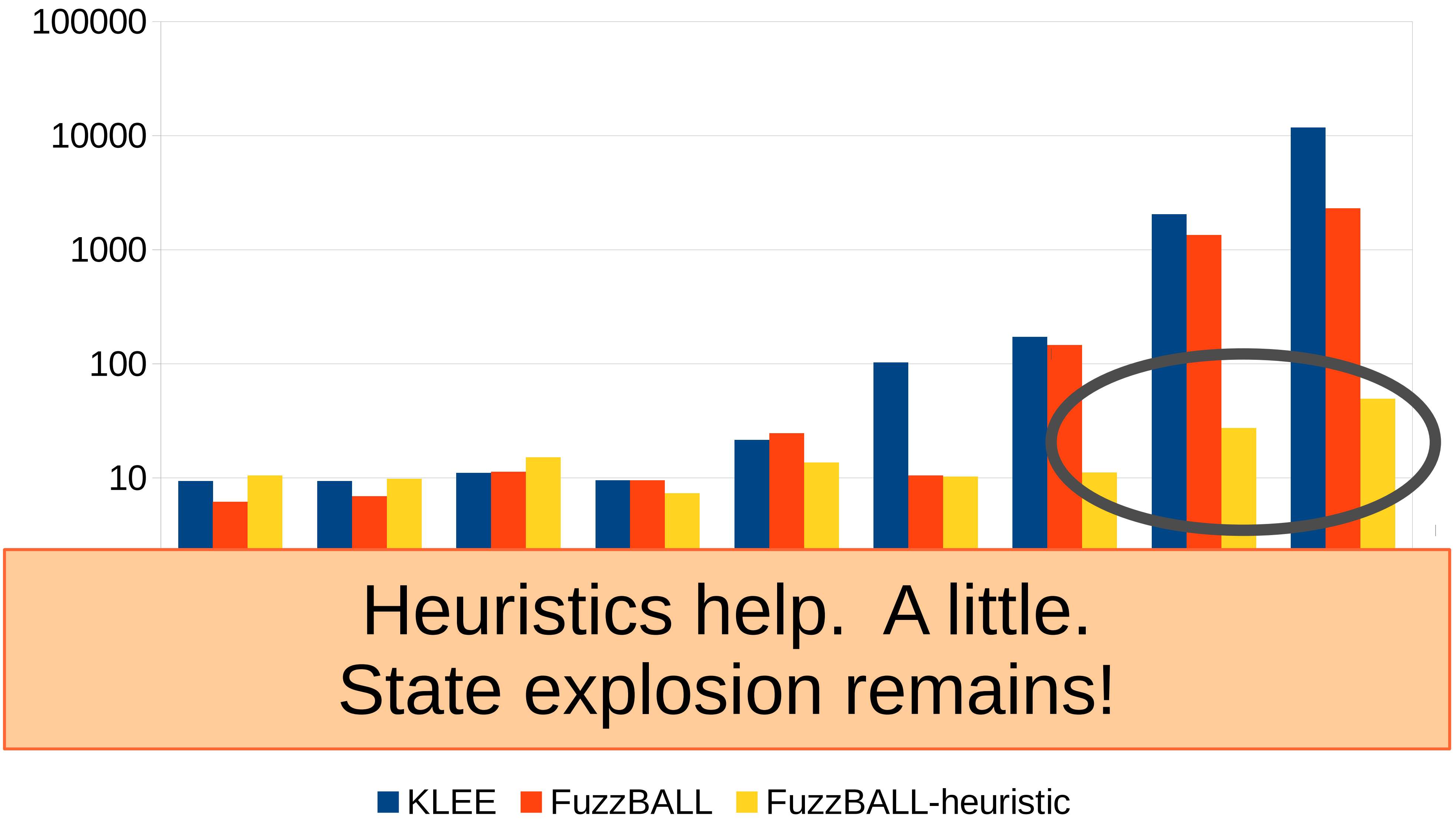
* Detailed results from TR Berkeley/EECS-2013-125

RLE encoding: heuristics*



* Detailed results from TR Berkeley/EECS-2013-125

RLE encoding: heuristics*



Heuristics help. A little.
State explosion remains!

* Detailed results from TR Berkeley/EECS-2013-125

Road map

Motivation

Definition and tools

State explosion

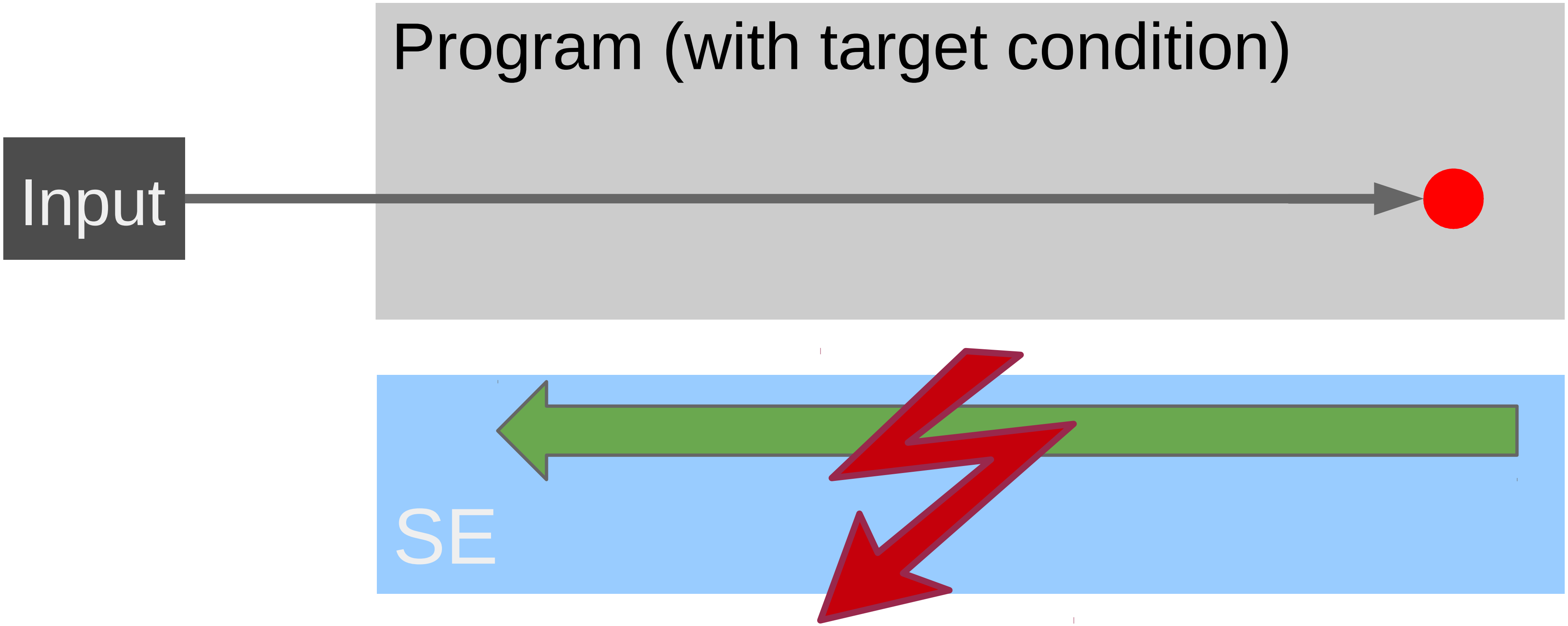
Scaling up

Divide and conquer

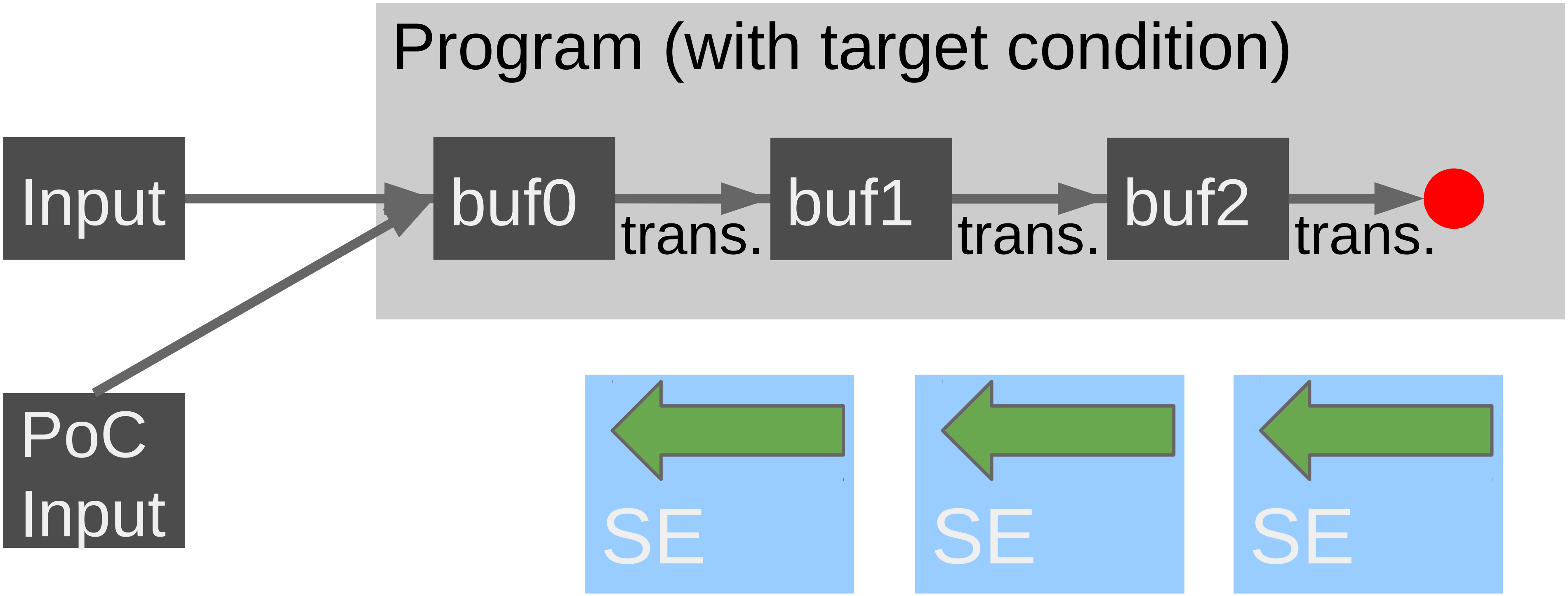
Binary analysis

The end

Divide and conquer



Divide and conquer



Does Symbolic Exec. scale?

Hex and Run Length Decoding

- Two transformations, e.g.,
  FB41014280 → \xfbA\x01B\x80
  \xfbA\x01B\x80 → AAAAAAB
- We know all buffer locations

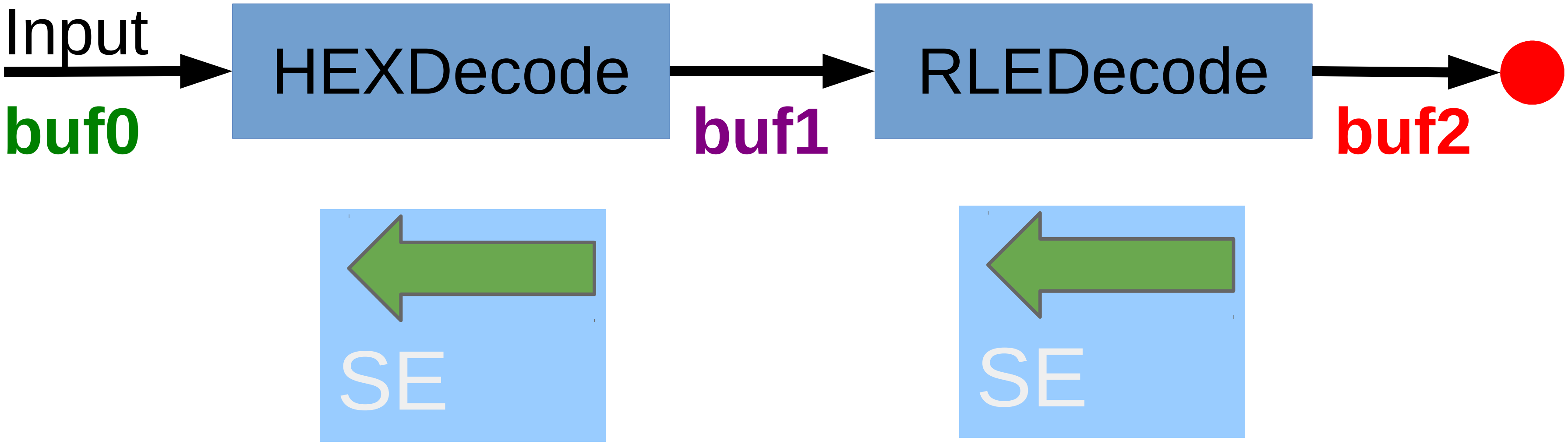
Evaluate performance of

- KLEE/FuzzBALL
- FuzzBALL with heuristics
- FuzzBALL with two iterations

Example #2: HEX & RLE

Demo!

```
ASCIIHexDecode(buf0, len0, buf1, 4096);
if (RunLengthDecode(buf1, len1, buf2, 4096) != -1) {
    if (strncmp(argv[3], (char*)buf2, strlen(argv[3])) == 0) {
        printf("Correctly recovered str\n");
    }
}
```



Input

buf0

HEXDecode

buf1

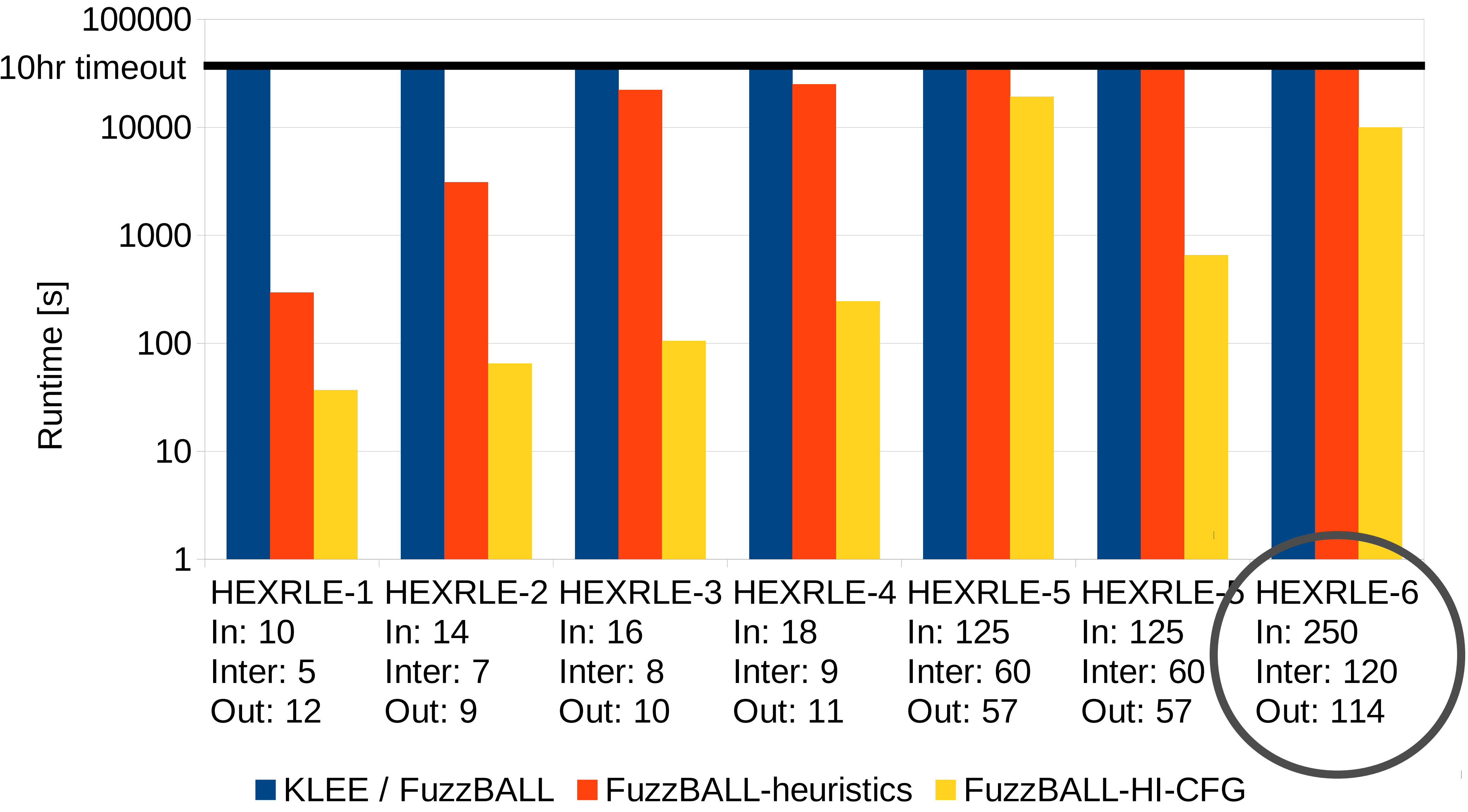
RLEDecode

buf2

SE

SE

HEXRLE encoding: iterations



One problem solved...

Divide and conquer mitigates scaling issues

We now have two new problems:

- Finding transformation boundaries
- Finding buffers locations

Road map

Motivation

Definition and tools

State explosion

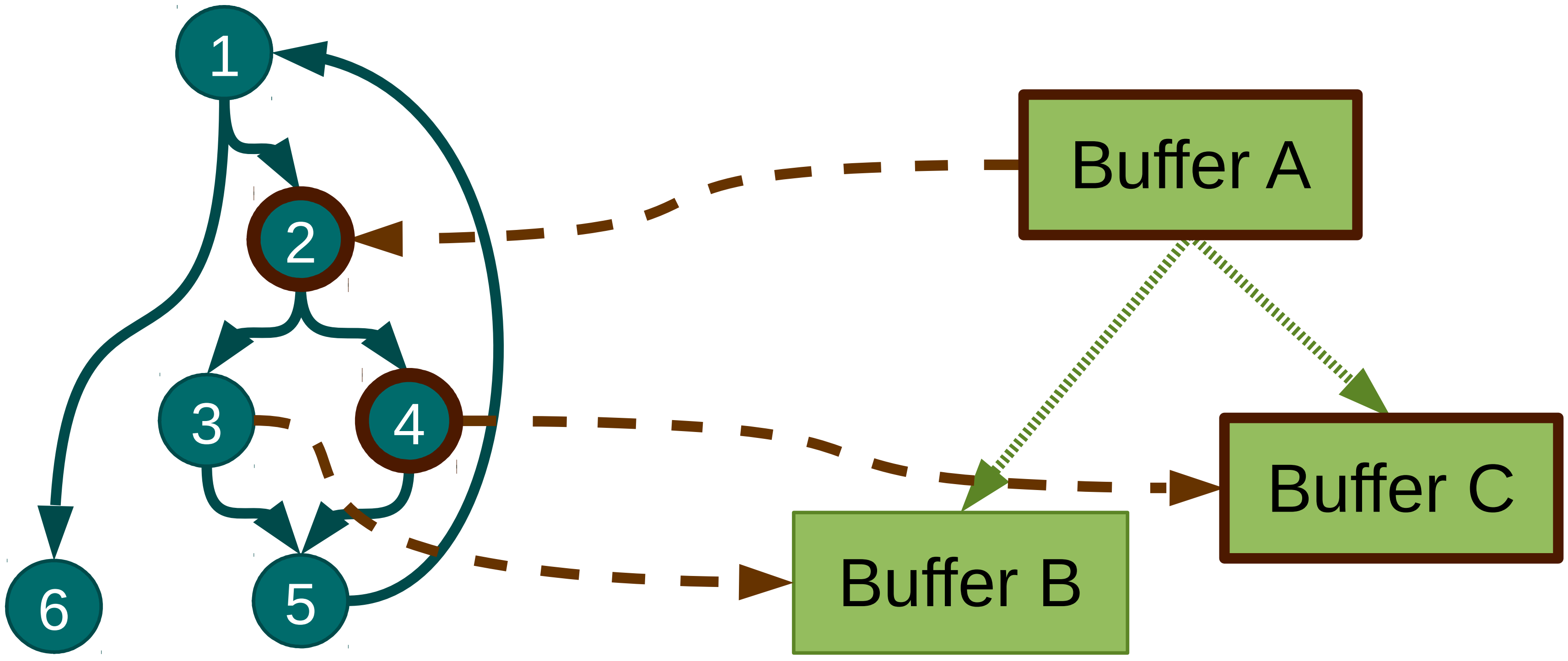
Scaling up

Divide and conquer

Binary analysis

The end

Hybrid Info. and Control-Flow Graph



Control-Flow Graph

Information-Flow Graph

Trace-based binary analysis

Trace allows to recover both (live) control-flow and information-flow using concrete input

1. Start with concrete input
2. Collect instruction-level trace
3. Process trace offline to discover buffers

Grouping memory accesses

"***Related***" accesses target same buffer

- Temporal relation
- Spatial relation

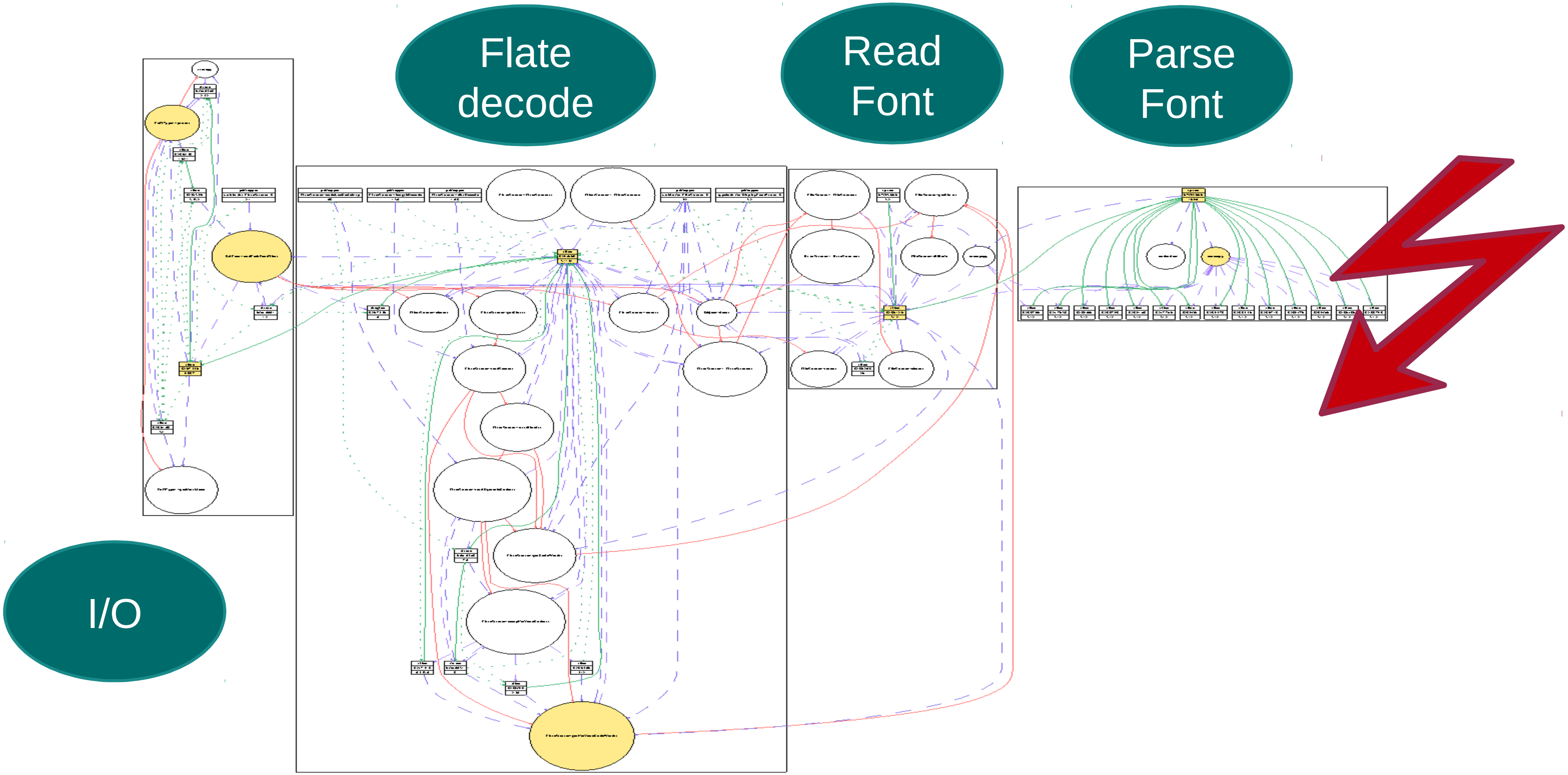
Assume a buffer hierarchy

- Layers of buffers

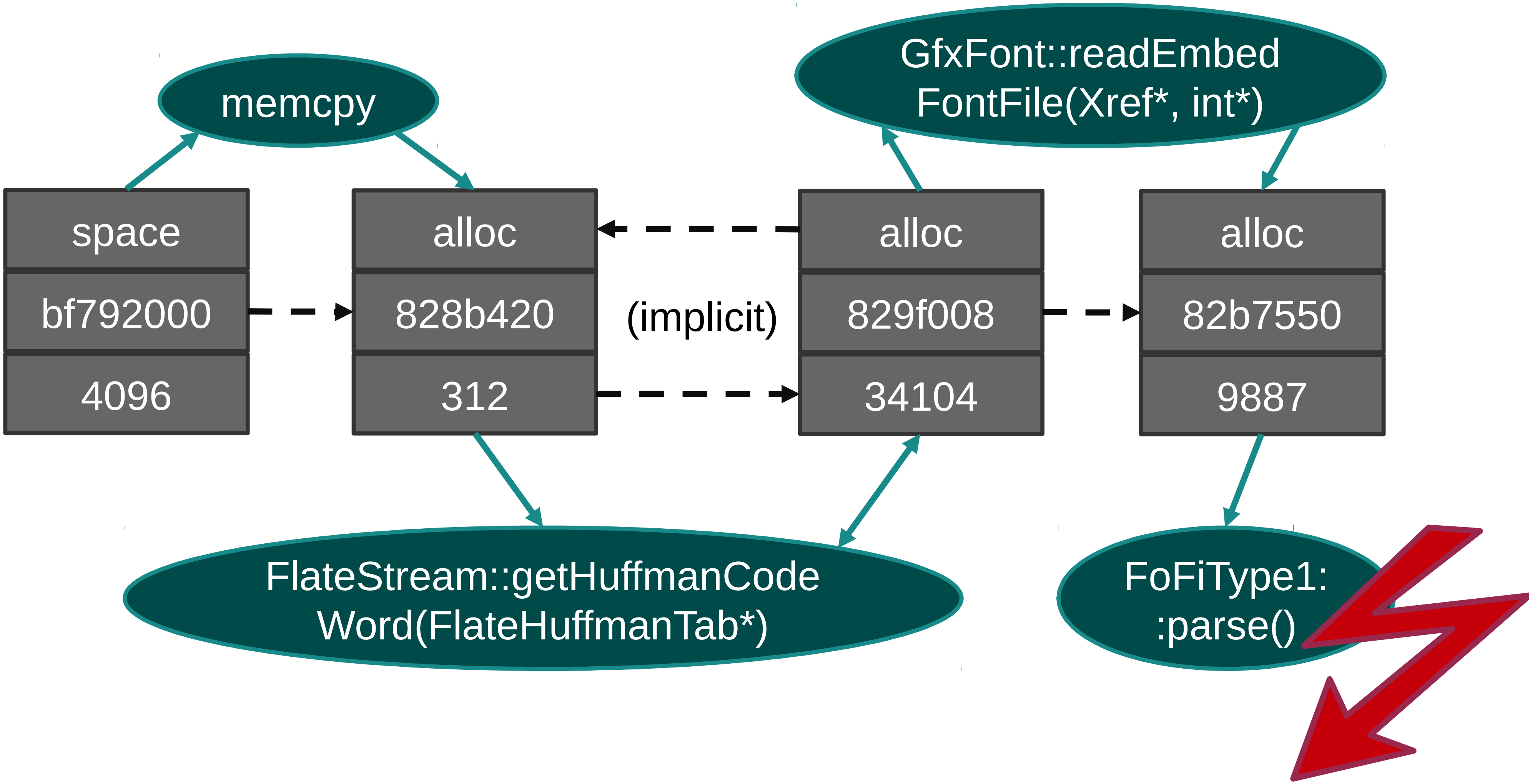
Find "***natural***" boundaries between transformations

Example #3: CVE-2010-3704

Type 1 font parsing bug in Poppler PDF-viewer

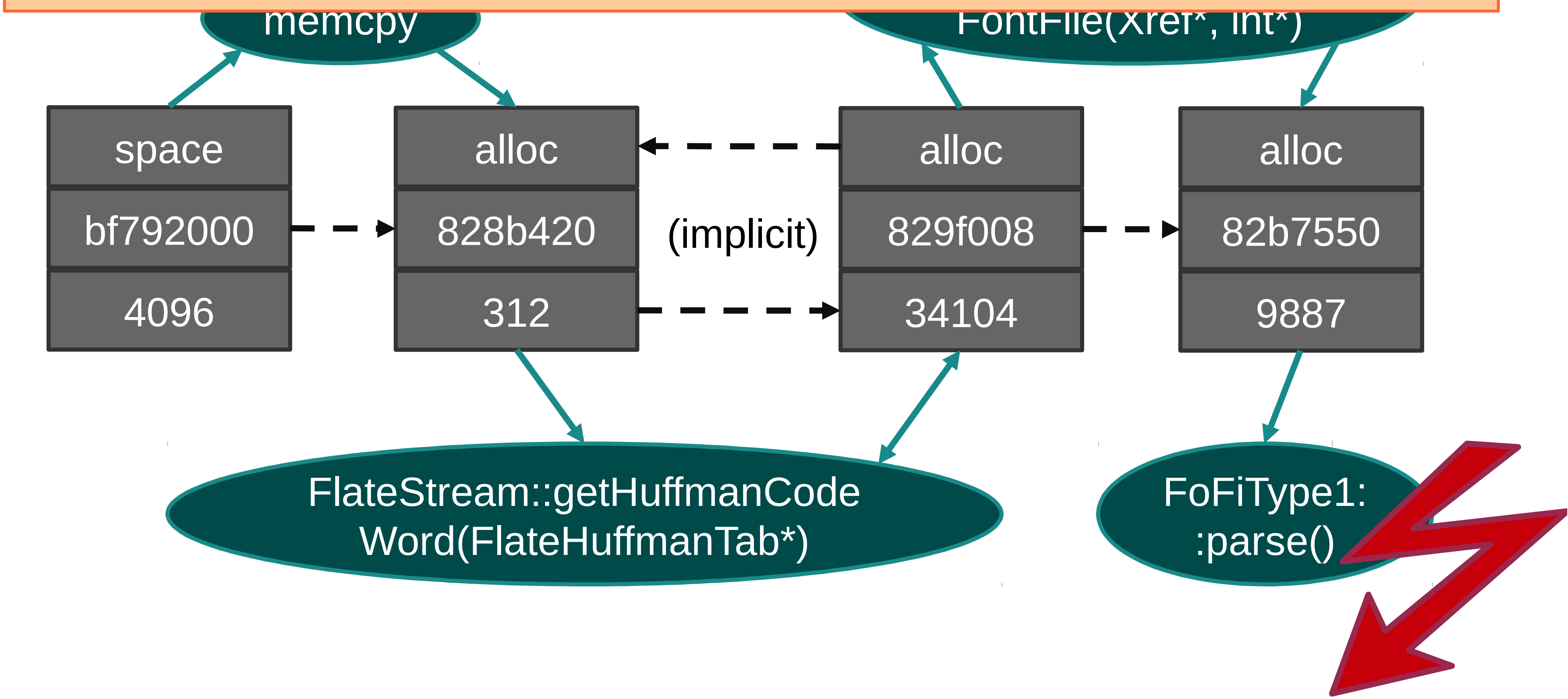


Example #3: Poppler buffers



Example #3: Poppler buffers

“Automatically” produce input
that triggers vulnerability



memcpy

FontFile(Xref*, int*)

space
bf792000
4096

alloc
828b420
312

(implicit)

alloc
829f008
34104

alloc
82b7550
9887

FlateStream::getHuffmanCode
Word(FlateHuffmanTab*)

FoFiType1:
:parse()

Road map

Motivation

Definition and tools

State explosion

Scaling up

Divide and conquer

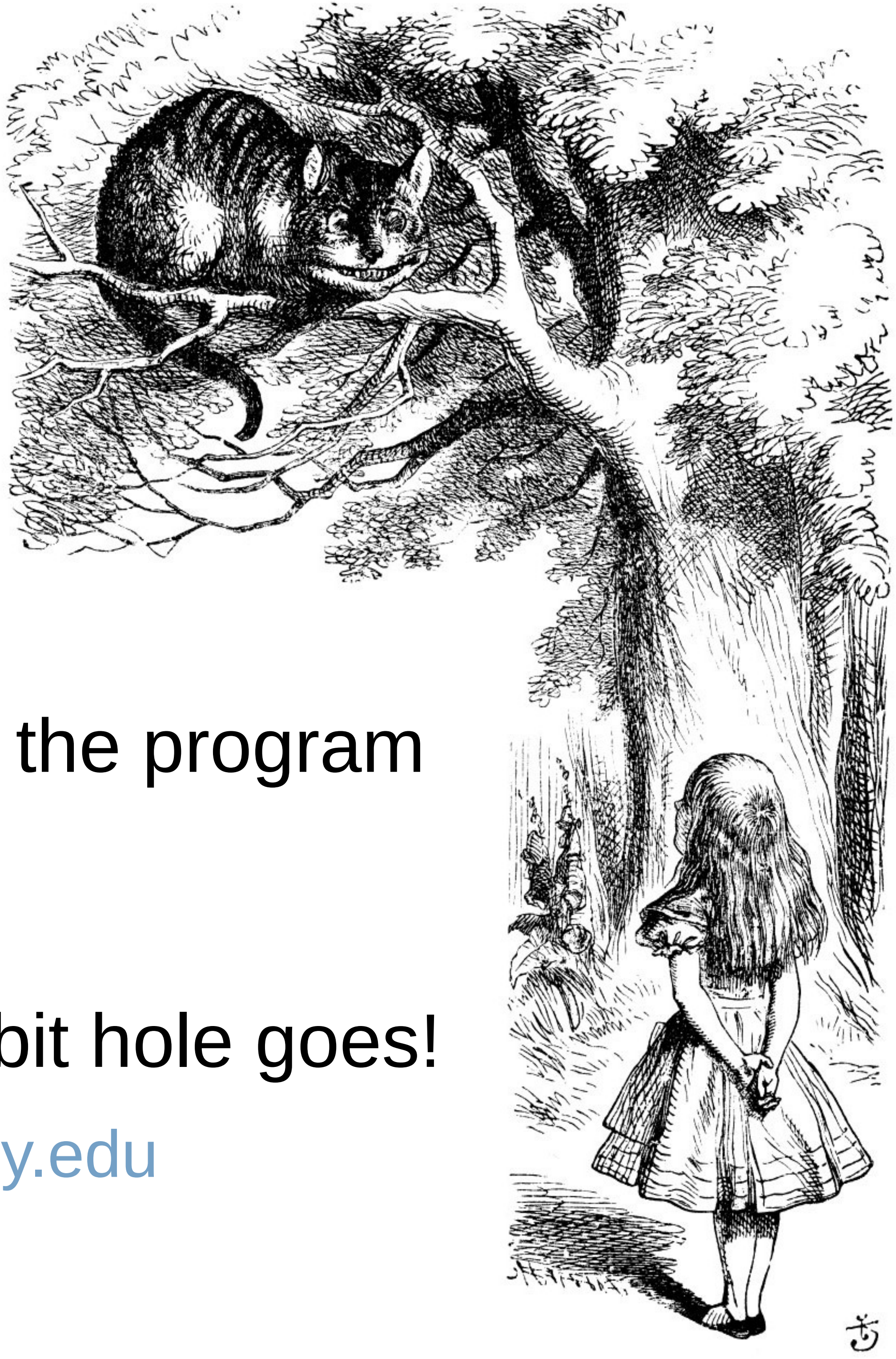
Binary analysis

The end

The end

Symbolic Execution is

- No panacea
- A great tool for PoCs



Trigger conditions deep in the program

- Construct PoC input

Explore how deep the rabbit hole goes!

- http://bitblaze.cs.berkeley.edu
- http://nebelwelt.net