MIMIC: Computing Models for Opaque Code

Stefan Heule, Manu Sridharan, Satish Chandra

Stanford University, Samsung Research America

September 4, 2015; FSE; Bergamo, Italy

Computing Models for Opaque Code

- Opaque code
  - Code is executable
  - Source not available, or hard to process
- Challenge: Program analysis in the presence of opaque code
- Model
  - Representation suitable for program analysis

Setting

- Opaque code in JavaScript
  - Standard library has native implementation
    - Arrays, Regex, Date, etc.
  - Code obfuscated before deployment

```
var arr = ['a','b','c','d'];
var x = arr.shift();
// x is 'a'
// arr is now ['b','c','d']
```

```
var _0x4240=["\x61","\x62","\x63","\x64",
"\x73\x68\x69\x66\x74"];
var arr=[_0x4240[0],_0x4240[1],
_0x4240[2],_0x4240[3]];
var x=arr[_0x4240[4]]();
```

Goals

- Problem statement: Given an (opaque) function f and some inputs I, automatically find a model that behaves like f

- Models should be executable (JavaScript code)
  - Agnostic to program analysis abstraction

Observing Opaque Code

- Opaque code is executable
  - Observe return values
  - Observe heap accesses on shared objects

```
['a','b','c','d'].shift();
```

- How can we get such detailed execution traces?
  - Ideally without having to change the JavaScript runtime

```
read field 'length' of arg0 //  4
read field 0 of arg0 // 'a'
has field 1 of arg0
read field 1 of arg0 // 'b'
write 'b' to field 0 of arg0
has field 2 of arg0
read field 2 of arg0 // 'c'
write 'c' to field 1 of arg0
has field 3 of arg0
read field 3 of arg0 // 'd'
write 'd' to field 2 of arg0
delete field 3 of arg0
write 3 to field 'length' of arg0
return 'a'
```

Execution Traces in JavaScript

- ECMAScript 6 will introduce proxies

- Proxies are objects of JavaScript with programmer-defined semantics
  - Intercept field reads, writes, enumerations of fields, etc.

```
var proxy = new Proxy(target, handler);
```

Example of Proxies in ECMAScript 6

```
var handler = {
  get: function(target, name) {
    return name in target? target[name] : 42;
  }
};
var p = new Proxy({}, handler);
p.a = 1;

console.log(p.a) // prints 1
console.log(p.b) // prints 42
```

- Strategy: proxy arguments to opaque code, record interactions

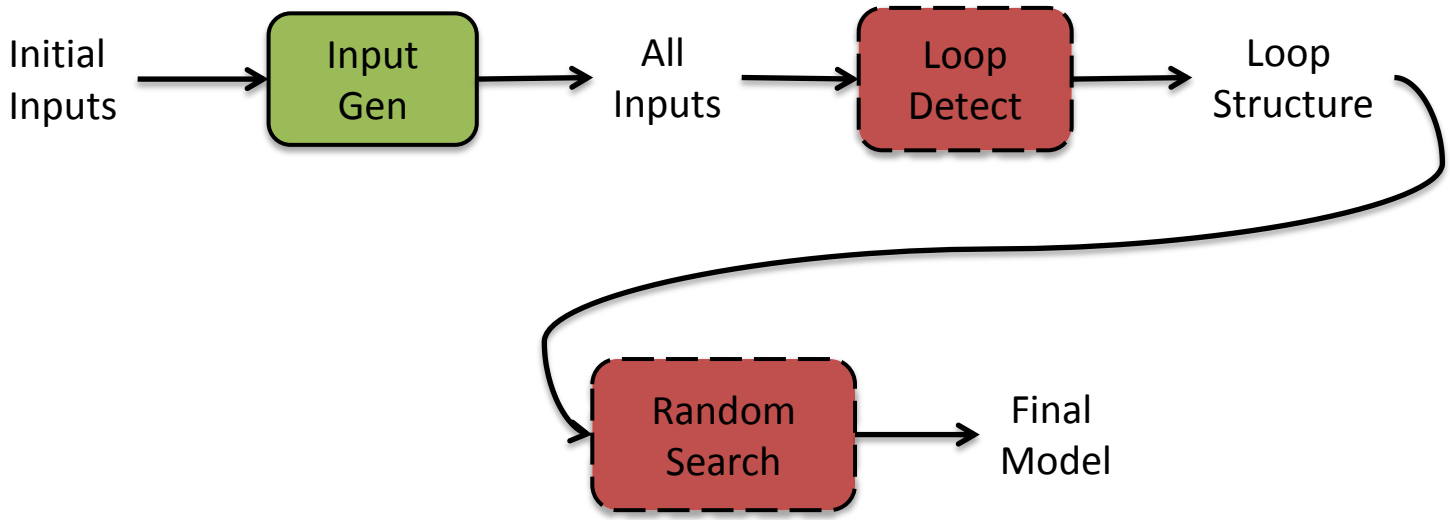
Traces Don't Tell the Full Story

- Traces contain partial information only
  - Where do values come from?
    - → Input generation
  - What is the program counter?
    - → Control flow reconstruction
  - What non-heap-manipulating computation is happening?
    - → Random search

```
read field 'length' of arg0 //  4
read field 0 of arg0 // 'a'
has field 1 of arg0
read field 1 of arg0 // 'b'
write 'b' to field 0 of arg0
has field 2 of arg0
read field 2 of arg0 // 'c'
write 'c' to field 1 of arg0
has field 3 of arg0
read field 3 of arg0 // 'd'
write 'd' to field 2 of arg0
delete field 3 of arg0
write 3 to field 'length' of arg0
return 'a'
```

Approach Overview

Given opaque function + initial inputs



Input Generation

Iterate Until Fixpoint or Enough Inputs

1. Start with initial inputs
2. Record traces for inputs
3. Extract locations from traces that are being read
4. Generate inputs that differ in those locations
5. Also, generate heuristically interesting inputs

['a','b','c','d']

[], ['b','b','c','d'], ['a','b','c'], ['b','foo','bar','def'], ...

Control Structure Reconstruction

- What statement did a trace event originate from?
  - Trivial for straight-line code
  - Less clear for loops

- Abstract trace to skeleton

  read; read; has; read; write; has; read; write; has; read; write; delete; write; return;

```
read field 'length' of arg0 //  4
read field 0 of arg0 // 'a'
has field 1 of arg0
read field 1 of arg0 // 'b'
write 'b' to field 0 of arg0
has field 2 of arg0
read field 2 of arg0 // 'c'
write 'c' to field 1 of arg0
has field 3 of arg0
read field 3 of arg0 // 'd'
write 'd' to field 2 of arg0
delete field 3 of arg0
write 3 to field 'length' of arg0
return 'a'
```

Control Structure Reconstruction (2)

- Problem can be viewed as learning a regular language

  read; read; has; read; write; has; read; write; has; read; write; delete; write; return;

  read; read; (has; (delete;| read; write;))* delete; write;

- From only positive examples
  - Theoretical result: impossible [Gold '67]

Control Structure Reconstruction (3)

- Limit ourselves to at most one loop

- There still might be multiple possible loop structures
  - Generate many proposals
  - Rank them based on how many traces they explain
    - Heuristic to break ties

read; read; (has; (delete;| read; write;))* delete; write;
read; read; (has; delete;| has; read; write;)* delete; write;
read; read; (has; (read; write;| delete; has; read; write;))* delete; write;
read; read; (has; (read; write;| delete;))*has; read; write; delete; write;

Control Structure Reconstruction (4)

- Probabilistically choose a loop proposal
  - Loop ranked i is chosen with probability

$$\alpha_{\text{loop}} \cdot \alpha \cdot (1-\alpha)^{i-1}$$

We use: $\alpha_{\text{loop}} = 0.9$ and $\alpha = 0.7$

- Multiple runs of procedure will eventually pick correct loop

Control Structure Reconstruction (5)

- Given the a loop proposal, we get an initial model

```
read;
read;
(
 has;
 (
  delete;
 |
  read;
  write;
 )
)*
delete;
write;
return;
```

→

```
var n0 = arg0.length
var n1 = arg0[0]
for (var i = 0; i < ?; i += 1) {
  var n2 = ? in arg0
  if (?) {
    delete arg0[?]
  } else {
    var n4 = arg0[?]
    arg0[?] = ?
  }
}
delete arg0[?]
arg0.length = ?
return ?
```

→

```
var n0 = arg0.length
var n1 = arg0[0]
for (var i = 0; i < 0; i += 1) {
  var n2 = 1 in arg0
  if (false) {
    delete arg0[0]
  } else {
    var n4 = arg0[1]
    arg0[0] = 'b'
  }
}
delete arg0[4]
arg0.length = 4
return 'a'
```

Randomized Search

- Then, apply random search (Markov Chain Monte-Carlo (MCMC) sampling inspired)
  - Randomly mutate the current program
  - Evaluate it with a fitness function
  - Accept "better" programs, and sometimes worse ones, too

Fitness Function

- Fitness function
  - Run model on all inputs
  - Compare all traces against real traces

- Score
  - Zero: if trace is matching perfectly
  - Partial score if only parts of trace are matching

Randomized Search (3)

- Program mutations
  - Select statement at random
  - Replace a random subexpression with a new random expression
    - For field read, replace either the field or receiver
    - For conditionals, replace condition
    - For loops, change loop bound
  - No need to remove/add statements
- Random expressions follow JavaScript grammar
  - Plus any local variable, constants seen in traces
  - Likelihood to generate expression of depth d decreases exponentially with d

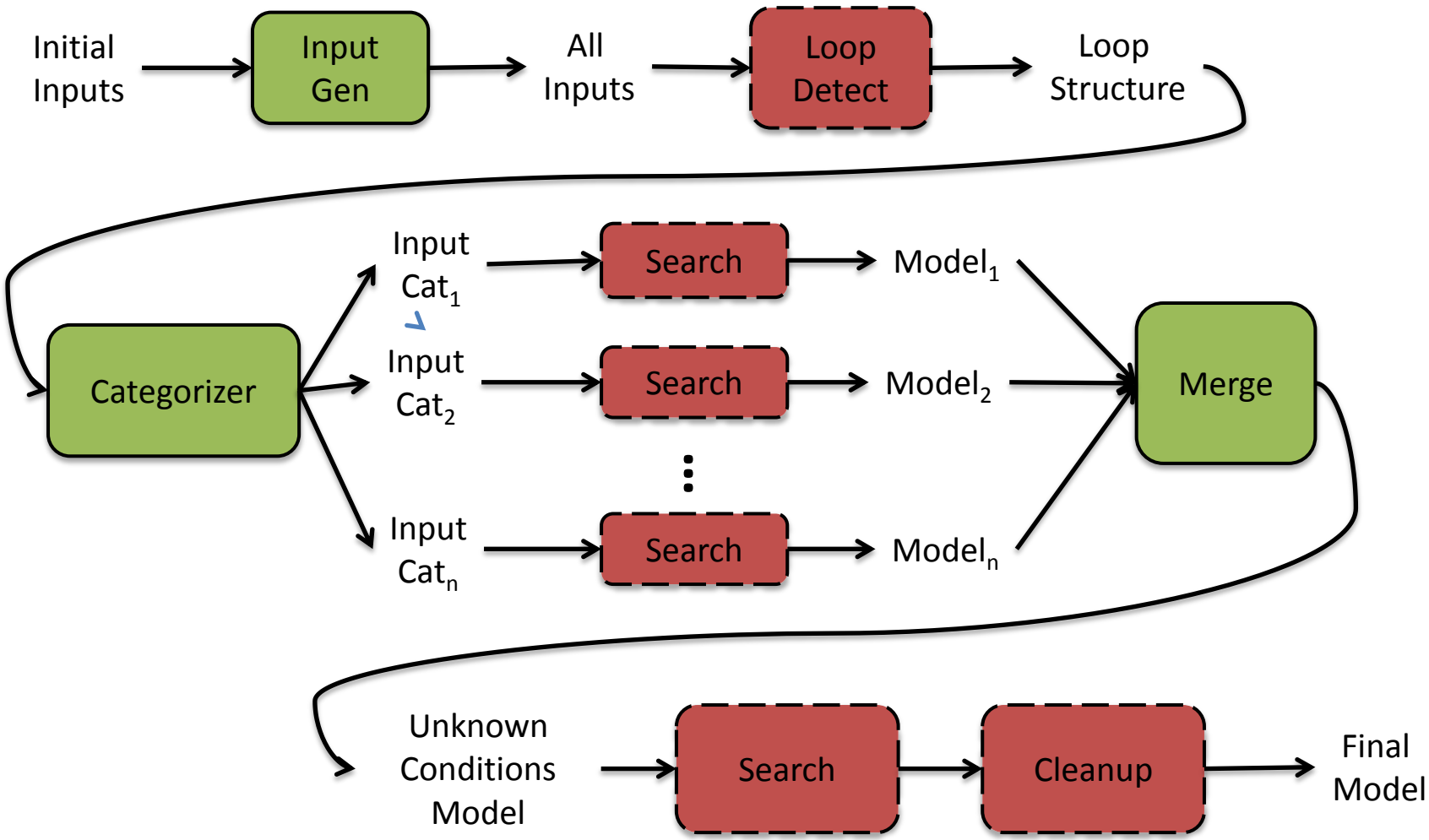
Random Search Result

- After some number of iteration, score goes to zero
  - This is a model
- What about the empty array??
  - Doesn't actually match the control flow structure

```
var n0 = arg0.length
var n1 = arg0[0]
for (var i = 0; i < (n0-1); i += 1) {
  var n2 = (i+1) in arg0
  if (n2) {
    var n3 = arg0[i+1]
    arg0[i] = n3
  } else {
    delete arg0[i]
  }
}
delete arg0[i]
arg0.length = i
return n1
```

Full Overview

Repeat until success:



Input Categories for shift

- For shift
  - Category for empty array
  - Category for non-empty arrays

```
var n0 = arg0.length
if (false) {
  /* model for non-empty arr */
} else {
  arg0.length = 0
}
```

```
var n0 = arg0.length
if (n0) {
  /* model for non-empty arr */
} else {
  arg0.length = 0
}
```

Implementation Notes

- Randomly generate models might not terminate
  - Stop execution if trace get too long

- Newly allocated objects
  - Don't show up in trace (only when returned)
  - Approach: Allocate at beginning of model, then randomly search for population code

```
if (false) { result[0] = 0; }
```

Optimizations

- Only use subset of inputs, not all inputs
  - Heuristic to choose 20 diverse inputs
    - How long is the trace? How does the initial model score?
  - At the end, validate with all inputs
    - If it fails, restart with failed inputs added
- Embarrassingly parallel search: exploit multiple cores
- Don't propose nonsensical programs
  - Type analysis

```
var n0 = arg0[arg0];
```

Evaluation

- JavaScript Array Standard Library

✓ every	✓ lastIndexOf	✓ reduce	✗ concat[1,2]	✗ sort[1]
✓ filter	✓ map	✓ reduceRight	✗ join[2]	✗ splice[1,2]
✓ forEach	✓ pop	✓ shift	✗ reverse[1]	✗ toString[3]
✓ indexOf	✓ push	✓ some	✗ slice[2]	✗ unshift[1]

- Problems
  1. Multiple loops
  2. Bugs in proxy implementation (not officially released)
  3. Missing program mutations

Evaluation (2)

- We contributed some of our models to WALA, a static analysis library for JavaScript
  - New models increase analysis precision
  - Also found a previous model to be wrong, and several to be incomplete (sparse arrays)

Evaluation (3)

Function	Time to synthesize (in seconds)	Loop rank
every	67.86 ± 22.41	1
filter	43.05 ± 16.13	1
forEach	4.59 ± 1.52	1
indexOf	42.94 ± 38.59	1
lastIndexOf	36.92 ± 19.22	2
map	15.64 ± 6.86	1
pop	2.35 ± 0.74	loop-free
push	291.94 ± 310.17	3
reduce	25.33 ± 12.51	1
reduceRight	126.41 ± 53.77	1
shift	117.54 ± 52.11	1
some	6.74 ± 3.16	1
max	56.61 ± 107.86	2
min	50.69 ± 121.95	2
sum	20.35 ± 39.83	2

Summary

- Opaque code problematic for analysis
- Automatically synthesize models
  - Using MCMC random search
  - Program traces to evaluate models

Source code and replication package

https://github.com/Samsung/mimic/

@stefan_heule

http://stefanheule.com/