Shadow Execution Frameworks for LLVM IR and JavaScript

Liang Gong & Cuong Nguyen*
* Name are in alphabetical order. Each author makes equal contributions to the project.

1. Motivation

Motivation
- Lesson learnt: code instrumentation/transformation and computation of extra information are foundation to dynamic analysis.
- Goal: minimize the efforts to build dynamic analysis by generalizing this approach with high scalability.

Contributions
- Design and implement an LLVM Shadow Execution System.
  - Based on instrumentation, concolic execution and shadow execution.
  - Perform instrumentation at the instruction level. The analysis is heavyweight but powerful.
  - Support multiple languages: C++, Java, OpenCL, Fortran, etc.
- Design and implement a JavaScript Dynamic Analysis Framework.
  - JavaScript is the most popular programming language for client-side web programming.
  - Facilitate analyzing any real-webpages.
  - Analysis module plug in and remove on the fly.
  - Powerful analysis interaction based on web console.
  - Detect real-world bugs using the framework.

2. Overview

Input: target application under analysis.
Output: instrumented/transformed codes with hooks.
Extensibility: override hooks to perform additional analysis.

3. Techniques and Challenges for LLVM IR

Shadow Execution
- Associate a shadow value with each concrete value.
- Carry useful information: error propagation, array bound checking, symbolic value, etc.
- Perform side-by-side with the concrete execution. Concrete execution assists shadow execution.

Core Interpreter (Resembles a compiler)
- Front end: instrument all 57 LLVM instructions + compute stack frame size for local/global variables + create fast lookup indices for variables.
- Back end: re-interpret all 57 LLVM instructions with shadow execution + model heap/stack and runtime environment.

Challenges
- Modeling dynamic allocation and pointer arithmetic in shadow execution.
  - Handle all tricky C pointer arithmetic (castings, pointer arithmetic, function pointers, etc.).
  - Array/struct and their combinations.
  - Handling uninterpreted calls.
  - With/without side effects + return arbitrary types (pointer, array, struct).
  - Model malloc.
- Achieving scalability: fast variable lookup and lazy construction of objects.

4. Evaluation for LLVM IR

Evaluation of the Core Interpreter
- Robustness: interpret 50 commonly used Linux core utilities (rm/ls/mkdir ..)
  - Check for each memory write that the interpretation and concrete execution writes the same value.
  - Check at each branch that two executions do not diverge.
- Performance:
  - 2x in average
  - 4x in the worst case

Evaluation of the Usability
- Implement an array-out-of-bound analysis (< 300 LoC).
  - Can trace back to the root causes of the problem.
  - Differentiate harmful and benign bugs.
  - Demo is available.
- Implement a NaN detection analysis (in progress).

5. Techniques and Challenges for JavaScript

Code Transformation:

```
%2 = load %b
%1 = load %a
%3 = add %1 %2
%a = alloca double
```

Jalangi Framework Runtime Code:

```
J$.W = function( ... ) { ... }
J$.R = function( ... ) { ... }
```

We add the following hooks:
- Binary Operation
- Unary Operation
- Variable Read/Write
- Field Get/Put
- Function/Method Call/Enter/Return
- Script Enter/Return
- Object/Function/Heap/Array Literal
- Condition/Switch

6. Evaluation for JavaScript

Check NaN Bug [< 100 Loc]

```
[object Object].now = end – start;  // NaN
```

Found interesting operations in the following websites’ homepages

Interesting Analysis Applications

- AttackMap
- Hack
- Hack
- Hack
- Hack
- Hack

Check NaN in [**query**] After diagnosing, confirm that is a bug.

```
[object Object].d <- undefined ಙ a <- undefined
```

Operating uninitialized variable

- Simply loading the Facebook homepage, the analysis detects hundreds of this kind of interesting operations.