Static Analysis Debugging with Symbolic Execution

Theodoros Kasampalis, Sandeep Dasgupta

11th August 2015

Outline

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Static Analysis

- Infer source code properties without execution
- Examples:
  - Pointer Analysis
  - Liveness Analysis
- Inferred properties true for any execution
- Applications:
  - Compilers
  - Security
  - Software Engineering

Outline

Debugging a Static Analysis Implementation

- Semantics bugs
  - no crash
  - erroneous results
- Effect visible in client code
  - Hard to trace back
- Static analysis specific tests
  - small regression tests

Debugging a Static Analysis Implementation

- Semantics bugs
  - no crash
  - erroneous results
- Effect visible in client code
  - Hard to trace back
- Static analysis specific tests
  - small regression tests

Debugging a Static Analysis Implementation

- Semantics bugs
  - no crash
  - erroneous results
- Effect visible in client code
  - Hard to trace back
- Static analysis specific tests
  - small regression tests

Debugging a Static Analysis Implementation

- Semantics bugs
  - no crash
  - erroneous results
- Effect visible in client code
  - Hard to trace back
- Static analysis specific tests
  - small regression tests

Debugging a Static Analysis Implementation

- Semantics bugs
  - no crash
  - erroneous results
- Effect visible in client code
  - Hard to trace back
- Static analysis specific tests
  - small regression tests

Debugging a Static Analysis Implementation

- Semantics bugs
  - no crash
  - erroneous results
- Effect visible in client code
  - Hard to trace back
- Static analysis specific tests
  - small regression tests

Debugging a Static Analysis Implementation

- Semantics bugs
  - no crash
  - erroneous results
- Effect visible in client code
  - Hard to trace back
- Static analysis specific tests
  - small regression tests

Outline

Related Work

- Compiler testing through miscompilation detection:
  - Regression test suites (LLVM test suite)
  - Randomly generated tests (Csmith, PLDI 11)
  - Equivalence Modulo Inputs (Orion, PLDI 14)
- Dynamic alias analysis error detection (NeonGoby, FSE 13)
- Symbolic execution (KLEE, OSDI 08)
- Concolic execution (zesti, ICSE 12 - SAGE, ICSE13)

Related Work

- Compiler testing through miscompilation detection:
  - Regression test suites (LLVM test suite)
  - Randomly generated tests (Csmith, PLDI 11)
  - Equivalence Modulo Inputs (Orion, PLDI 14)
- Dynamic alias analysis error detection (NeonGoby, FSE 13)
- Symbolic execution (KLEE, OSDI 08)
- Concolic execution (zesti, ICSE 12 - SAGE, ICSE13)

Related Work

- Compiler testing through miscompilation detection:
  - Regression test suites (LLVM test suite)
  - Randomly generated tests (Csmith, PLDI 11)
  - Equivalence Modulo Inputs (Orion, PLDI 14)
- Dynamic alias analysis error detection (NeonGoby, FSE 13)
- Symbolic execution (KLEE, OSDI 08)
- Concolic execution (zesti, ICSE 12 - SAGE, ICSE13)

Related Work

- Compiler testing through miscompilation detection:
  - Regression test suites (LLVM test suite)
  - Randomly generated tests (Csmith, PLDI 11)
  - Equivalence Modulo Inputs (Orion, PLDI 14)
- Dynamic alias analysis error detection (NeonGoby, FSE 13)
- Symbolic execution (KLEE, OSDI 08)
- Concolic execution (zesti, ICSE 12 - SAGE, ICSE13)

Related Work

- Compiler testing through miscompilation detection:
  - Regression test suites (LLVM test suite)
  - Randomly generated tests (Csmith, PLDI 11)
  - Equivalence Modulo Inputs (Orion, PLDI 14)
- Dynamic alias analysis error detection (NeonGoby, FSE 13)
- Symbolic execution (KLEE, OSDI 08)
- Concolic execution (zesti, ICSE 12 - SAGE, ICSE13)

Related Work

- Compiler testing through miscompilation detection:
  - Regression test suites (LLVM test suite)
  - Randomly generated tests (Csmith, PLDI 11)
  - Equivalence Modulo Inputs (Orion, PLDI 14)
- Dynamic alias analysis error detection (NeonGoby, FSE 13)
- Symbolic execution (KLEE, OSDI 08)
- Concolic execution (zesti, ICSE 12 - SAGE, ICSE13)

Related Work

- Compiler testing through miscompilation detection:
  - Regression test suites (LLVM test suite)
  - Randomly generated tests (Csmith, PLDI 11)
  - Equivalence Modulo Inputs (Orion, PLDI 14)
- Dynamic alias analysis error detection (NeonGoby, FSE 13)
- Symbolic execution (KLEE, OSDI 08)
- Concolic execution (zesti, ICSE 12 - SAGE, ICSE13)

Outline

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Our Idea

- Check inferred properties during symbolic execution
  - Apply analysis to an input program
  - Symbolically execute the input program
  - Check whether inferred properties hold
- Direct testing of static analysis code
- Static analysis inferences checked thoroughly
  - High path coverage of the input program
  - Big input program size

Outline

System Status Overview

- Checks incorporated within zesti
- Checks on all pointer dereferences
- Reachability analysis for inputs affecting pointer values [under implementation]
- Testing with LLVM test suite programs

System Status Overview

- Checks incorporated within zesti
- Checks on all pointer dereferences
- Reachability analysis for inputs affecting pointer values [under implementation]
- Testing with LLVM test suite programs

System Status Overview

- Checks incorporated within zesti
- Checks on all pointer dereferences
- Reachability analysis for inputs affecting pointer values [under implementation]
- Testing with LLVM test suite programs

System Status Overview

- Checks incorporated within zesti
- Checks on all pointer dereferences
- Reachability analysis for inputs affecting pointer values [under implementation]
- Testing with LLVM test suite programs

Outline

Symbolic Execution

- Symbolic execution using klee
- Migration from Klee to Zesti (a variant of klee)

Checker Logic

```
1. foreach load instructions
    1.1. base_address  = 'base address' of the load
    1.2. foreach 'pointer' in the same function scope as the load instruction
        1.2.1. result = mustAlais_OR_mayNOTAlias('base_address', 'pointer') // Querying the alias analysis.
              1.2.1.1. if result ==  must-alias, check if 'base_pointer' and 'pointer' points to the same
                     runtime memory object.
              1.2.1.2. if result ==  mayNot-alias, check if 'base_pointer' and 'pointer' do not points to the
                     same runtime memory object.
              1.2.1.3. Otherwise, continue.
```

Implicit klee_assumes

```
struct S {
  int x, y;
};
struct S data[] =
{
  { 1,2 },
  { 3,4 },
};
int main(int argc, char** argv) {

int x= 0 ;
struct S* z;

klee_make_symbolic(&x, sizeof(x), "X");
/*
** Without the following klee_assume, the dereference z->x gets resolved to many
** spurious memory objects.
** Generated in-bound constraints on the fly to prevent this.
*/
klee_assume(x >= 0 & x <= 100 );

z = &data[x++];
... = z->x ;

return 0;
}
```

Importance of choosing a variable as symbolic

```
1. int main() {
2.    int x=1 , y=2;
3.    int*  p = (int *)malloc(sizeof(int));

4.    klee_make_symbolic(&x, sizeof(x), "x");
5.    klee_make_symbolic(&y, sizeof(y), "y");
  /*
  ** If we skip to make y symbolic, then we may miss the
  ** opportunity of catching a potential pointer analysis
  ** bug. For ex. what if the pointer analysis infers that
  ** *p and the heap object at line 7 mayNOT alias.
  */

      if(0 != x*y) {
6.      p = (int *)malloc(4);
      } else {
        if(y == 0) {
7.        p = (int *)malloc(4);
        }
      }
8.    return *p;
}
```

Which variables to make symbolic

- Explicitly specifying which variablies to make symbolic is difficult.
  - Instrumented the code by inserting appropriate klee_make_symbolic.
  - Rechability Analysis to figure out candidates to be made symbolic.

Bugs Found

```
/* The bug shows up when there is a must alias check between
** x (at line 1) and the bitcast of x (at line 3).
*/
int main(int argc, char **argv) {
  int *A[5];
  for (int i = 0; i < 5; ++i) {
    A[i] = (int*) malloc((i+1)*sizeof(int));
  }

  int *x, a;
  char *y;

  for (int i = 0; i < 5; ++i) {
 1.    x = A[i];
 2.    a = *x;
 3.    y = (char *) x;
  }
  return *y;
}
}
```

Outline