

Static Analysis Debugging with Symbolic Execution

Theodoros Kasampalis, Sandeep Dasgupta

12th August 2015

Outline

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation
- 3 Related Work
- 4 Background
- 5 Our Idea
- 6 System Status Overview
- 7 Implementation
- 8 Questions?

Static Analysis

- Infer source code properties without execution

Static Analysis

- Infer source code properties without execution
- Examples:

Static Analysis

- Infer source code properties without execution
- Examples:
 - Pointer Analysis

Static Analysis

- Infer source code properties without execution
- Examples:
 - Pointer Analysis
 - Liveness Analysis

Static Analysis

- Infer source code properties without execution
- Examples:
 - Pointer Analysis
 - Liveness Analysis
- Applications:

Static Analysis

- Infer source code properties without execution
- Examples:
 - Pointer Analysis
 - Liveness Analysis
- Applications:
 - Compilers

Static Analysis

- Infer source code properties without execution
- Examples:
 - Pointer Analysis
 - Liveness Analysis
- Applications:
 - Compilers
 - Security

Static Analysis

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

Static Analysis

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

Outline

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation**
- 3 Related Work
- 4 Background
- 5 Our Idea
- 6 System Status Overview
- 7 Implementation
- 8 Questions?

Debugging a Static Analysis Implementation

- Semantic bugs

Debugging a Static Analysis Implementation

- Semantic bugs
 - no crash

Debugging a Static Analysis Implementation

- Semantic bugs
 - no crash
 - erroneous results

Debugging a Static Analysis Implementation

- Semantic bugs
 - no crash
 - erroneous results
- Effect visible in client code

Debugging a Static Analysis Implementation

- Semantic bugs
 - no crash
 - erroneous results
- Effect visible in client code
 - Hard to trace back

Debugging a Static Analysis Implementation

- Semantic bugs
 - no crash
 - erroneous results
- Effect visible in client code
 - Hard to trace back
 - Not reliable

Debugging a Static Analysis Implementation

- Semantic bugs
 - no crash
 - erroneous results
- Effect visible in client code
 - Hard to trace back
 - Not reliable
- Static analysis specific tests

Debugging a Static Analysis Implementation

- Semantic bugs
 - no crash
 - erroneous results
- Effect visible in client code
 - Hard to trace back
 - Not reliable
- Static analysis specific tests
 - small regression tests

Outline

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation
- 3 Related Work**
- 4 Background
- 5 Our Idea
- 6 System Status Overview
- 7 Implementation
- 8 Questions?

Related Work

- Compiler testing through miscompilation detection:

Related Work

- Compiler testing through miscompilation detection:
 - Test suites (LLVM test suite)

Related Work

- Compiler testing through miscompilation detection:
 - Test suites (LLVM test suite)
 - Randomly generated tests (Csmith, PLDI 11)

Related Work

- Compiler testing through miscompilation detection:
 - Test suites (LLVM test suite)
 - Randomly generated tests (Csmith, PLDI 11)
 - Equivalence Modulo Inputs (Orion, PLDI 14)

Related Work

- Compiler testing through miscompilation detection:
 - 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)

Related Work

- Compiler testing through miscompilation detection:
 - 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)

Related Work

- Compiler testing through miscompilation detection:
 - 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

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation
- 3 Related Work
- 4 Background**
- 5 Our Idea
- 6 System Status Overview
- 7 Implementation
- 8 Questions?

Array Out of Bounds Bug

```
1. int v[100];  
2. void f(int x) {  
3.     if (x > 99)  
4.         x = 99;  
5.     v[x] = 0;  
6. }  
  
7. int main(int argc, char **argv) {  
8.     int x = atoi(argv[1]);  
9.     f(x);  
  
10.    return 0;  
11. }
```

Symbolic Execution with KLEE

```
1. int v[100];  
2. void f(int x) {  
3.     if (x > 99)  
4.         x = 99;  
5.     v[x] = 0;  
6. }  
  
7. int main(int argc, char **argv) {  
8.     int x;  
9.     klee_make_symbolic(&x, sizeof(x), "X");  
10.    f(x);  
  
11.    return 0;  
12. }
```

Concolic Execution with zesti

```
1. int v[100];  
2. void f(int x) {  
3.     if (x > 99)  
4.         x = 99;  
5.     v[x] = 0;  
6. }  
  
7. int main(int argc, char **argv) {  
8.     int x = 50;  
9.     klee_make_symbolic(&x, sizeof(x), "X");  
10.    f(x);  
  
11.    return 0;  
12. }
```


Concolic Execution with zesti

```
1. int v[100];  
2. void f(int x) {  
3.     if (x > 99)  
4.         x = 99;  
5.     v[x] = 0;  
6. }  
  
7. int main(int argc, char **argv) {  
8.     int x = 100;  
9.     klee_make_symbolic(&x, sizeof(x), "X");  
10.    f(x);  
  
11.    return 0;  
12. }
```

Outline

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation
- 3 Related Work
- 4 Background
- 5 Our Idea**
- 6 System Status Overview
- 7 Implementation
- 8 Questions?

Our Idea

- Check inferred properties during symbolic execution

Our Idea

- Check inferred properties during symbolic execution
 - Apply analysis to an input program

Our Idea

- Check inferred properties during symbolic execution
 - Apply analysis to an input program
 - Symbolically execute the input program

Our Idea

- Check inferred properties during symbolic execution
 - Apply analysis to an input program
 - Symbolically execute the input program
 - Check whether inferred properties hold

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

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

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

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

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation
- 3 Related Work
- 4 Background
- 5 Our Idea
- 6 System Status Overview**
- 7 Implementation
- 8 Questions?

System Status Overview

- Implementation for checking an LLVM Alias Analysis (including tbaa, basicaa)

System Status Overview

- Implementation for checking an LLVM Alias Analysis (including tbaa, basicaa)
- Checks incorporated within zesti

System Status Overview

- Implementation for checking an LLVM Alias Analysis (including tbaa, basicaa)
- Checks incorporated within zesti
- Checks on all loads

System Status Overview

- Implementation for checking an LLVM Alias Analysis (including tbaa, basicaa)
- Checks incorporated within zesti
- Checks on all loads
- Pointer dereferences marked sensitive

System Status Overview

- Implementation for checking an LLVM Alias Analysis (including tbaa, basicaa)
- Checks incorporated within zesti
- Checks on all loads
- Pointer dereferences marked sensitive
- Reachability analysis for inputs affecting pointer values [under implementation]

System Status Overview

- Implementation for checking an LLVM Alias Analysis (including tbaa, basicaa)
- Checks incorporated within zesti
- Checks on all loads
- Pointer dereferences marked sensitive
- Reachability analysis for inputs affecting pointer values [under implementation]
- Testing with LLVM test suite programs

Outline

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation
- 3 Related Work
- 4 Background
- 5 Our Idea
- 6 System Status Overview
- 7 Implementation**
- 8 Questions?

Symbolic Execution

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

Debugger Logic for Pointer Analysis

- Following check is done after each pointer dereference (say loadI)

```
base_address = 'base address' of the loadI
pointerSet   = All the pointers in the same function scope as loadI
foreach( 'pointer' in pointerSet) {
    result = MustAlias_OR_MayNOTAlias('base_address', 'pointer') // Querying the alias analysis.
    if( result == must-alias) {
        if ('base_pointer' and 'pointer' DO NOT point to the same run-time memory object) {
            error
        }
    }
    if (result == mayNot-alias) {
        if ('base_pointer' and 'pointer' point to the same run-time memory object) {
            error
        }
    }
}
```

Implicitly adding klee_assumes

```
struct S {
    int member;
};
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 <= 1 );

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

    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 variables to make symbolic is difficult.
 - Instrumented the code by inserting appropriate `klee_make_symbolic`.
 - Reachability Analysis to figure out candidates to be made symbolic.

Bug Found in LLVM BasicAA

```
/* The bug shows up when there is a must alias check between
** x (at line 1) and the bitcast of x (at line 3).
** Our debugger is able to detect the error at i = 1, terminate and produce the input as N = 2147483647
*/
int main(int argc, char **argv)
{
    int *A[5];
    int N = 1;

    klee_make_symbolic(&N, sizeof(N), "N");

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


Outline

- 1 Static Analysis
- 2 Debugging a Static Analysis Implementation
- 3 Related Work
- 4 Background
- 5 Our Idea
- 6 System Status Overview
- 7 Implementation
- 8 Questions?**

