Exploiting Undefined Behaviors for Efficient Symbolic Execution

Asankhaya Sharma
Department of Computer Science, NUS
asankhs@comp.nus.edu.sg

Motivation
Symbolic execution is a popular technique used for test generation, debugging and program analysis. We have developed a technique to reduce the runtime cost of symbolic execution with binaries.

Main Idea
- During compilation we use a static analysis to systematically introduce undefined behaviors (UB) in programs
- This triggers existing aggressive compiler optimizations based on undefined behaviors that reduce the size of generated binaries
- Reuse existing compiler optimizations for eliminating code that is not relevant for symbolic execution
- Based on a simple static analysis (CVA) that is applied as a pass during the compilation
- Does not require any change in the underlying symbolic execution engine to use the results from static analysis for dynamic path exploration
- Allows reduction in size of compiled binaries and prevents generation of irrelevant constraints

Key Benefits
- 14% reduction in size of binaries
- 30% reduction in number of constraints generated
- 48% reduction in time taken for symbolic execution

Overview of the Method
Change Value Analysis

Program
Program with CVA
Generated Binary

Program after CVA

Program with UB

Change Value Analysis
Statically determine program variables that depend on change in the value of the output using a three point lattice on status of program variables (Changed, Unchanged and Undefined)
1. Initially mark all variables as Undefined
2. Mark all output variables as Changed
3. Working backwards mark all those variables that depend on Changed variables as Changed
4. Continue till fixed point is reached
In the end replace all Undefined and Unchanged variables with a nondeterministic Undefined value

Three Point Lattice

Changed
Reachable code that affects the output

Unchanged
Reachable code that does not affect output

Undefined
Unreachable Code

Source Code
Change Value Analysis (GPL 3)  http://github.com/codelion/pa.llvm/tree/master/CVA
Pathgrind (GPL 3)  http://github.com/codelion/pathgrind

Experiments

Key Benefits

Benches from Software-artifact Infrastructure Repository (SIR)

Implemented as a compiler pass in LLVM Generated binaries are symbolically executed using Pathgrind

An Example

Program before CVA

```c
int foo (int x, int y, int z)
{
    int a;
    a = z;
    if (x - y > 0)
        a = x;
    else
        a = y;
    if (z > a)
        printf("z is max");
    return a;
}
```

Program after Compiler Optimizations

```c
int foo (int x, int y, int z)
{
    int a;
    if (x - y > 0)
        a = x;
    else
        a = y;
    if (z > a)
        printf("z is max");
    return a;
}
```

Program with UB

```c
int foo (int x, int y, int *)
{
    int a;
    a = *
    if (x - y > 0)
        a = x;
    else
        a = y;
    if (* > a)
        printf("z is max");
    return a;
}
```

Still possible to generate the same test cases using dynamic symbolic execution as the constraints on input that affect the output are preserved

Program with UB

Replace '*' with '*' which represents a nondeterministic value (e.g. Undefined in LLVM)

Undefined value triggers optimizations based on undefined behaviors which eliminates 3 lines from the program