Assignment 1– Symbolic Execution Playground (Part 1)

HSS
Fall 2022

In this part, you will be building a dynamic symbolic executor for C programs with LLVM and Z3.
Logistics:

- **LLVM Primer:** Please make sure that you have skimmed the LLVM Primer presentation (access it from the course webpage) to know the capabilities of LLVM.
- **Setup Repo:** I have created a `github` repo with all the necessary scripts to install LLVM, Z3 and starter code to write a pass. You can access it at: `https://github.com/HolisticSoftwareSecurity/hssllvmsetup`. The repo has examples of analysis (i.e., the passes that do not modify the IR) and instrumentation (i.e., the passes that modify the IR) passes.
- **Development Environment:** I use CLion (`https://www.jetbrains.com/clion/`) while working with LLVM and strongly suggest you to use it. You can get unlimited access using your `@purdue.edu` email.

In this part, you will implement a dynamic symbolic execution (DSE) engine that automatically generates inputs to efficiently explore different program paths. You will use an LLVM pass to encode C programs into our symbolic interpretation API that we have provided. The resulting tool will find assignments for input variables that crash an input C program.

Setup

The skeleton code for this part is located under `LLVMBasedDSE` folder of the following repo. We will frequently refer to this top level directory as `dse` when describing file locations for the lab. Clone the repository to a folder:

```
$ cd
$ git clone https://github.com/HolisticSoftwareSecurity/DynamicSymbolicExecution.git
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 14 (delta 11), reused 11 (delta 8), pack-reused 0
Unpacking objects: 100% (14/14), done.
```

Then, run the following commands setup the lab:

```
$ cd ~/DynamicSymbolicExecution/LLVMBasedDSE
$ mkdir build && cd build
$ cmake ..
$ make
$ export LD_LIBRARY_PATH=~/DynamicSymbolicExecution/LLVMBasedDSE/build/DSE:$LD_LIBRARY_PATH
```

The export `LD_LIBRARY_PATH` command should be run on each terminal session you begin. You should now see `dse` and `libInstrumentPass.so` in the current directory `LLVMBasedDSE/build/DSE`.

`dse` is a tool that performs dynamic symbolic execution on an input program using Z3. You can run the dse program with the following commands:

```
$ cd ~/DynamicSymbolicExecution/LLVMBasedDSE/DSE
$ cd test
$ clang -emit-llvm -S -fno-discard-value-names -c simple0.c
$ opt -load ../../build/DSE/libInstrumentPass.so -Instrument -S simple0.ll -o simple0.instrumented.ll
$ clang -o simple0 -L../../build/DSE -lruntime simple0.instrumented.ll
$ ../../build/DSE/dse ./simple0 N # where N is the number of iterations
$ timeout 10 ../../build/DSE/dse ./simple0  # run for 10 seconds
```

Initially, you will see `formula.smt2 not found` since you have not implemented the instrumentation part yet.

Lab Instructions

Dynamic symbolic execution (DSE) uses techniques from both randomized testing and symbolic execution to search all of a program's execution paths for bugs. DSE tracks both runtime values and symbolic constraints, and uses the former to simplify solving the latter during a backtracking search on program computation trees.

We have provided the backbone for a symbolic interpreter using Z3 [1]. You will need to encode a C program into this symbolic interpreter API as well as write the code that drives the dynamic symbolic execution. We provide several details on how to do this in the following sections.

This lab assumes that input programs only have integer variables (no pointers or other types of variables) and do not have functions (no CallInstr).

Understanding Z3. Z3 is a theorem prover developed at Microsoft. It's a large and complex tool, so this will serve as a cursory guide for its capabilities and what it can do. Consider a simple, generic system of equations such as the following, where X and Y are integers:

```
X < Y
X > 2
```

Although this example is trivial, think about how you might solve this using any programming language of your choice. You may resort to using loops to check numbers or finding a library to handle matrix multiplication. This is because most of these programming languages are imperatively-directed, meaning there's a sequence of commands needed to solve the problem.

On the other hand, Z3 has a declarative interface, which in this case means all you need to give it is the list of constraints (in this case, `X < Y` and `X > 2`). Plug the following into the online Z3 solver [2] to see the results:

```
(declare-const x Int)
(declare-const y Int)
(assert (< x y))
(assert (> x 2))
(check-sat)
(get-model)
```

[1] https://github.com/Z3Prover/z3

[2] https://rise4fun.com/Z3/tutorial/guide

Z3 may not give you all possible results that match the constraints but importantly, it verifies satisfiability, which is the key factor that will be leveraged for this DSE engine.

If you're curious about Z3 and want more information, you can check out the following resources:

- `https://github.com/Z3Prover/z3/wiki/Documentation`
- `https://github.com/Z3Prover/z3/blob/master/examples/c%2B%2B/example.cpp`
- `https://theory.stanford.edu/~nikolaj/programmingz3.html`

Part 1: LLVM Instrumentation

The first component of this dynamic symbolic execution implementation is instrumentation of the input program, which is done in `src/Instrument.cpp`. This follows the familiar format and pattern seen in prior labs, except now this LLVM pass will inject various functions defined in `src/Runtime.cpp`, accompanied with the appropriate metadata from each valid LLVM Instruction. This will enable DSE to interact with Z3 at runtime. Specifically, these are the functions that will require instrumentation (from `include/Instrument.h`):

```
static const char *DSEInitFunctionName = "__DSE_Init__";
static const char *DSEAllocaFunctionName = "__DSE_Alloca__";
static const char *DSEStoreFunctionName = "__DSE_Store__";
static const char *DSELoadFunctionName = "__DSE_Load__";
static const char *DSEConstFunctionName = "__DSE_Const__";
static const char *DSERegisterFunctionName = "__DSE_Register__";
static const char *DSEICmpFunctionName = "__DSE_ICmp__";
static const char *DSEBranchFunctionName = "__DSE_Branch__";
static const char *DSEBinOpFunctionName = "__DSE_BinOp__";
```

Symbolic Inputs. The skeleton code provides an auxiliary function called `DSE_Input` for a user to specify symbolic inputs. In target programs, you should first include the header file `include/Runtime.h` to use the function. In the following example code, the dynamic symbolic execution engine will treat variable `x` and `y` to have symbolic inputs and `z` to have a concrete value 0:

```
#include \../include/Runtime.h"
int main() {
    int x, y, z;
    DSE_Input(x);
    DSE_Input(y);
    z = 0;
    ...
}
```

Note that `DSE_Input` is a macro and will be expanded with a unique ID. See `include/Runtime.h` and `src/SymbolicInterpreter.cpp` for details.

Initially, the DSE engine will assign random numbers to the input variables. After each iteration of DSE, new inputs are generated and stored in file input.txt in the form of comma-separated values (CSV). The file will have a mapping from IDs to their integer values. The following is an example of the symbolic mapping X0 : 1, X1 : 10:

```
X0,1
X1,10
```

If there exists an `input.txt` file, target programs instrumented with the methodologies below will use the integer values for inputs rather than random numbers.

Instrumentation for DSE Initialization. You will first instrument the input program to invoke a function `__DSE__Init__` at the beginning of main. The skeleton code provides the definition of `__DSE__Init__` in `src/SymbolicInterpreter.cpp`. The function initializes inputs if `input.txt` exists and registers a callback function `__DSE_Exit__` which will be invoked when the target program is terminated normally. The skeleton code also provides the definition of `__DSE_Exit__` that stores a list of covered branches (in `branch.txt`), path formula (in `formula.smt2`), and logs (in `log.txt`). In short, your instrumentation module should transform the code in the left to the right:

```
define dso_local i32 @main() #0 {
entry:
    %retval = alloca i32, align 4
    ...
```

```
define dso_local i32 @main() #0 {
entry:
call void __DSE_Init__();
    %retval = alloca i32, align 4
    ...
```

Instrumentation for IR instructions. You will next instrument the remaining IR instructions. In general, each operand in an instruction should be instrumented if it changes anything in the symbolic memory state. Constants are instrumented with the `__DSE_Const__` function and registers are instrumented with the `__DSE_Register__` function (see the next section for details). Additionally, the instrumented function calls for the Alloca instructions must appear after the instruction, whereas the instrumented function calls for all other instructions must appear before the instruction. `__DSE_ICmp__` and `__DSE_BinOp__` take the ID of the register in the left hand side as their first argument and its LLVM opcode (`llvm::CmpInst::Predicate` and `llvm::Instruction:: BinaryOps`, respectively) as the second argument. We provide some example instrumentations (the function calls are simplified for readability):

```
...
  %x = alloca i32, align 4
...
```

```
...
  %x = alloca i32, align 4
__DSE_Alloca__(i32 1, i32* %x)
...
```

```
...
  store i32 0, i32* %retval, align 4
...
```

```
...
__DSE_Const__(i32 0)
__DSE_Store__(i32* %retval)
  store i32 0, i32* %retval, align 4
...
```

```
...
  %cmp = icmp eq i32 %1, 1024
...
```

```
...
 __DSE_Register__(i32 5)
 __DSE_Const__(i32 1024)
 __DSE_ICmp__(i32 6, i32 32)
  %cmp = icmp eq i32 %1, 1024
...
```

Part 2: Runtime Symbolic Interpretation

The second component of this lab involves writing the runtime symbolic interpretation functions in `src/Runtime.cpp`. In previous labs, the instrumentation functions have been provided, but this time you will be doing it yourself. When each of these functions get invoked at runtime, it will change the symbolic memory state and path conditions. Here is where you'll be using the Z3 API to add constraints for the symbolic interpreter class.

Symbolic Interpretation for LLVM Instructions. You will define symbolic manipulation functions for each LLVM instruction and instrument the input program to invoke these functions at runtime. Following the real execution of the program, the DSE engine manipulates a symbolic memory state. The `SymbolicInterpreter` class in `include/SymbolicInterpreter.h` maintains symbolic memory which is defined as a map from symbolic addresses to symbolic expressions. It also maintains a stack of symbolic expressions.

An instance of the `Address` class represents a symbolic memory address. A symbolic address is either a memory address or a register, following the definition of LLVM IR. The Type field denotes the type of address. For memory addresses (allocated via AllocaInstruction of LLVM), we will use their physical addresses as symbolic addresses. For registers, we will assign unique register IDs via `getRegisterID()` in `Instrument.h`. For symbolic expression, you will reuse Z3's expressions, which are instances of the `z3::expr` class.

Symbolic manipulation of concrete execution is performed using two auxiliary functions `__DSE_Const__`, `__DSE_Register__`, each of which encodes concrete constants and registers to their symbolic counter parts. The functions are defined in `src/SymbolicInterpreter.cpp`. Function `__DSE_Const__` takes a constant integer of LLVM IR, makes a symbolic expression for the number, and pushes the symbolic expression to a stack (the field Stack in class `SymbolicInterpreter`). Function `__DSE_Register__` takes an ID of an LLVM register, and pushes its symbolic counterpart to the stack. Each element of the stack is either a constant or a register. The symbolic expressions in the stack will be used for the succeeding instrumented function.

You will define the symbolic manipulation functions for LLVM instructions using the auxiliary functions. Consider the following LLVM code equivalent to a simple C program `int x = 1; int y = x;` (types are omitted for simplicity):

Instrumented	Concrete Memory	Stack	Symbolic Memory
`%x = alloca`			
`__DSE_Alloca__(0,%x)`			
`%y = alloca`	`%x : 0x1000`		
`__DSE_Alloca__(1,%y)`	`%y : 0x1004`	`[]`	`Reg(0) : 0x1000`
`__DSE_Const__(1)`			
`__DSE_Store__(%x)`		`[]`	`Reg(1) : 0x1004`
`store 1, %x`		`[Const(1)]`	
`__DSE_Load__(2,%x)`	`0x1000 : 1`	`[]`	`0x1000 : 1`
`%a = load %x`			
`__DSE_Register__(2)`	`%a : 1`	`[]`	`Reg(2) : 1`
`__DSE_Store__(%y)`		`[Reg(2)]`	
`store %a, %y`	`0x1004 : 1`	`[]`	`0x1004 : 1`

- `__DSE_Alloca__` takes the ID of the register in the left hand side and the address of a newly allocated physical memory block. In the above example, the ID of \%x is 0 and the physical memory address is `0x1000`. The symbolic memory after line 2 will have entry `Reg(0) : 0x1000`.
- `__DSE_Store__` assumes that there exists a symbolic expression of its value operand (constant or register) on top of the stack. It takes a physical memory address as a parameter and stores the symbolic expression at the address.
- `__DSE_Load__` takes the ID of the register in the left hand side and the address of the physical memory block of which value will be loaded to the register.

The behavior of other symbolic manipulation functions are defined in a similar way. `__DSE_ICmp__` and `__DSE_BinOp__` take the ID of the register in the left hand side and its LLVM opcode (`llvm::CmpInst::Predicate` and `llvm::Instruction::BinaryOps`, respectively). The skeleton code provides the implementation of `__DSE_Branch__` in `SymbolicInterpreter.cpp` for a reference.

Working with Z3 Expressions. Instructions like `llvm::Inst::CmpInst` and `llvm::BinaryOperator` manipulate symbols and need to be represented equivalently in the constraints. You will be working with Z3 Expressions to represent these manipulations. The Z3 API uses a feature of C++ called **operator-overloading** to allow you to use C++ arithmetic and comparison operators with objects of type `z3::expr`. We show some examples below to represent arithmetic and comparison expressions on `z3::expr` objects. These examples assume that E1 and E2 are two objects of type `z3::expr`, and their result is stored in another object E of type `z3::expr`.

Operation	**Representation**
Addition	E = (E1 + E2)
Less-Than	E = (E1 <E2)

Part 3: Backtracking Strategy

Recall from the lecture how conditions were handled in order for the DSE analysis to explore more paths of the input test program. Modify the `searchStrategy()` function in `src/Strategy.cpp` to perform this backtracking behavior. It should alter the current path formula that will be given to Z3 so that it will derive a new input.

Path Formula and Search Strategy. After each execution of an instrumented program, a path formula will be encoded and stored in `formula.smt2`. All the IDs of executed branch instructions will be stored in `branch.txt` in order of execution which may be useful to generate next inputs. Given the current satisfiable path formula, function `searchStrategy` will propose a formula to derive new inputs that can lead to exploring more paths. The main function in `DSE.cpp` will iteratively generate new inputs until a crashing input is found or a timeout occurs.

Format of Input Programs

Input programs in this lab are assumed to have only sub-features of the C language as follows:

- All values are integers (i.e., no floating points, pointers, structures, enums, arrays, etc). You can ignore other types of values.
- Assume that user inputs are only introduced via the `DSE_Input` function and other call instructions to other functions do not exist.

Example Input and Output

Your DSE engine should run on a given instrumented program. For example,

```
$ cd DSE/test
$ make
$ ../../build/DSE/dse ./simple0 5
```

It will find a crashing input after 1 iteration and the input will be stored in `input.txt`:

```
Floating point exception
Crashing input found (1 iters)
$ cat input.txt
X0,1024
```

Items to Submit

Submit files `Instrument.cpp`, `Runtime.cpp`, and `Strategy.cpp`