Lab: binary-level software security

2012 summer schools on cryptography and principles of software security

Gang Tan, Lehigh University

In this lab session, we are going to use the LLVM compiler for inserting software masks before memory store instructions to enforce that writes to memory are within the data region. For simplicity, we ignore the control-flow aspect.

1 Getting familiar with LLVM

LLVM is a compiler infrastructure with reusable compiler and optimization components. Its compiler front-end, `clang`, compiles source code into code in an intermediate language, called LLVM bitcode. LLVM optimizations such as constant propagation are implemented as bitcode-to-bitcode transformations. The LLVM backend then translates bitcode to native code. We have installed a copy of LLVM's source code in the VM image under the `llvm` directory.

Our plan is to insert an extra pass into LLVM: it performs bitcode-to-bitcode transformation so that a masking instruction is inserted before every memory write.

If you are new to LLVM, Chris Lattner's overview article is a great start:
`http://www.aosabook.org/en/llvm.html`

1.1 LLVM's command-line tools

Go to page `http://clang.llvm.org/get_started.html` and go through the commands under the section "Clang Compiler Driver". We put a `hello.c` under directory `llvm/examples`.

- To compile it to native code, run

```
clang hello.c -o hello
```

- To compile it to LLVM bitcode, run

```
clang -c -emit-llvm hello.c -o hello.bc
```

- To view the bitcode, run LLVM's bitcode disasembler:

```
llvm-dis hello.bc -o -
```

- To run the LLVM optimizer on the bitcode, run

```
opt -O3 < hello.bc > hello_opt.bc
```

1.2 LLVM programming basics

Go over the document "LLVM Programmer's Manual" at `http://llvm.org/docs/ProgrammersManual.html`. Reading sections 1-3 and 5 should be sufficient for this lab session.

1.3 Learn how to add a pass to LLVM

Go over the document "Writing an LLVM Pass" at `http://llvm.org/docs/WritingAnLLVMPass.html`. Just read sections 1 and 2 for now. The source code of the `Hello` pass is at the directory

```
llvm/llvm-3.1.src/lib/Transforms/Hello/Hello.cpp
```

2 Adding pass: dumping memory operations

Add a new pass in `Hello.cpp`. The new pass should dump information about all memory load and store instructions in a bitcode file. We have already added skeleton code in `Hello.cpp`. Search for the `DumpMemOp` class and make it complete.

The `DumpMemOp` class has a method `runOnBasicBlock`, which runs on every basic block of a bitcode file. Given a basic block, it iterates through all its instructions. So you should add code for matching load and store instructions. Look for the discussion about `dyn_cast` in "LLVM Programmer's Manual" to learn how to match a particular kind of instructions.

After you finish the coding, go to the following directory and type `make` to build the pass

```
llvm/build/lib/Transforms/Hello
```

Run the following command to run the newly coded pass on a bitcode file called `memwrite.bc`

```
opt -load ../build/Release+Asserts/lib/LLVMHello.so -dumpmemop < memwrite.bc > /dev/null
```

3 Adding pass: instrumenting memory writes

Complete the code in the following file, which should implement a pass that adds a masking instruction before every store instruction.

```
llvm/llvm-3.1.src/lib/Transforms/Scalar/InsMemWrite.cpp
```

If the memory address to write in a store instructions is in variable `%a`, then the software mask should be implemented by a bit-wise logical-and instruction. Specifically, perform a logical-and between `$a` and constant `0x20000000` and store the result back to `$a`. here, we assume the region ID is `0x2000`.

Note that it won't be enough to insert just a logical-and instruction. LLVM bitcode is a typed intermediate language. The memory address in a store instruction is of a pointer type, while the logical-and instruction takes integers. So it's necessary to first insert an instruction that converts the address pointer to an integer, followed by the logical-and instruction, and then followed by an instruction for converting the result integer to a pointer.

After you finish the coding, you can run the following command directly to invoke the pass for instrumenting memory writes:

```
opt -inswrite < memwrite.bc > memwrite_new.bc
```

4 An optimization

Implement the optimization described as follows. It inserts less masking instructions before memory writes. The idea is that if a variable's definition dominates multiple uses and each use is possibly for a memory write, then we can just insert one masking instruction after the definition. This is more efficient than inserting one mask before each use.

The following program gives a simple example. The definition of pointer `p` at "`int *p = &a`" dominates the following two memory writes through `p`. So it is safe if we insert a masking instruction immediately after the definition.

```
int a;
int *p = &a;
*p=10;
*p=a+20;
```

LLVM bitcode uses static-single-assignment (SSA) form, which makes implementing the above optimization quite easy. You should use the def-use chain to facilitate your implementation.