Fall 2024 CSC512
Compilers Final Project

Elizabeth Lin
North Carolina State University
etlin@ncsu.edu

1 Motivation and Objectives

The goal of the project is to identify inputs that determine behaviors in a C program. Inputs could be from the user or from files on the system.

2 Challenges

To determine what values control a program's behavior, we first have to identify key points. Key points include values that control the execution of loops and function pointers. Identifying key points manually is not too complicated, we can skim over the source code of the program and look for keywords that are related to loops, for example: *for, while*. However, programatically identifying key points and where they originate from is more complicated.

To do so programatically, we need to first identify branching points in a program. Then we need to trace a variable from the branching point back to its origin.

3 Code Artifacts

The submission repository for the code is at https://github.com/elizabethtl/csc512-course-proj, this includes the pass and test files. The dev repo is at https://github.com/elizabethtl/csc512-course-proj-dev, in addition to the submission repo, this includes the LLVM source files.

4 Solutions

Our course project focuses on using LLVM passes to identify key points in a program. LLVM passes allow you to make changes to the compiler and output more information about the code.

Before we start writing our LLVM pass, we have to download the LLVM source code. The LLVM source code can be found at the github repository: https://github.com/LLVM/LLVM-project. We download the source code from the Github repository and build LLVM.

With LLVM built on our machine, we can start writing our LLVM pass. We used the skeleton pass at https://github.com/sampsyo/LLVM-pass-skeleton to get started with our pass. Our main directory contains a `CmakeLists.txt` file that specifies how we compile the pass. Our pass is located in a subdirectory `part1`. The pass is a C++ file `pass.cpp`. The `part1` subdirectory also contains another `CMakeLists.txt` file that specifies the source files of the pass. To build the pass, we create a folder `build` in our main directory. Listing 1 shows the commands we ran in our main directory to build the pass.

```
mkdir build      // create build dir
cd build
cmake ..         // use CMake file in root dir
make
cd ..            // change back to root dir
```

Listing 1. Build LLVM pass

```
clang -O0 -g -fno-discard-value-names -fpass-plugin=`echo build/part1/part1pass.*` example.c
```

Listing 2. Compile with LLVM pass

```
PreservedAnalyses run(Module &M, ModuleAnalysisManager &AM) {
  for (auto &F : M) {
    for (auto &BB : F) {
      for (auto &I : BB) {
        if (BranchInst *br = dyn_cast<BranchInst>(&I)) {
          if (br->isConditional()) {
            Value *condition = br->getCondition();
            traceVariableOrigin(condition);
  }}}}}
  return PreservedAnalyses::all();
};
```

Listing 3. Main function in LLVM pass

To run the pass, we compile C programs with `clang` using the `-fpass-plugin` flag. Listing 2 includes the command we use to compile and run our pass. The `-g` flag, the `-O0` flag, and `-fno-discard-value-names` ensure that the debug information is included during our compilation. The debug information is used in our passes to identify metadata such as variable names and line numbers.

4.1 Structure of our LLVM pass

Our LLVM pass starts with a main function `run`. In this function, we identify :

- functions within a module
- basic blocks within a function
- instructions within a basic block

Figure 1 shows the flow of our LLVM pass.

For each instruction, we check if it is a loop or branch instruction. If it is, we continue to the next part of the pass to

PL'18, January 01–03, 2018, New York, NY, USA
2018.

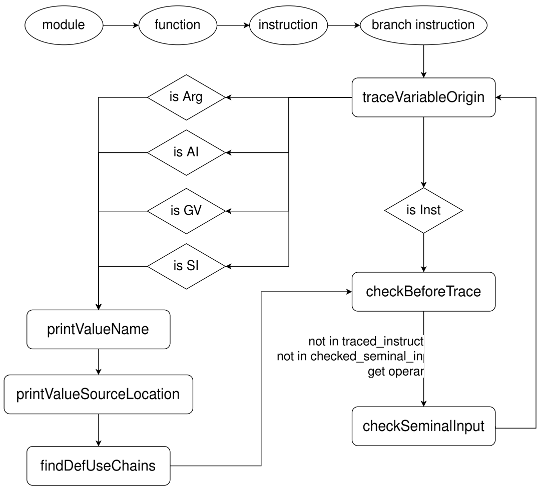

Figure 1. LLVM pass code flow

```
void printValueName(Value *V) {
    errs() << "\thas name: " << V->hasName() << ", value name: "
      << V->getName() << "\n";
}
```

Listing 4. function that prints the variable name

```
void printValueSourceLocation(Value *V) {
    User* lastUser = nullptr;
    for (auto *user : V->users()) {
        lastUser = user;
    }
    if (auto *instr = dyn_cast<llvm::Instruction>(lastUser)) {
        if (DebugLoc debugLoc = instr->getDebugLoc()) {
            unsigned line = debugLoc.getLine();
            unsigned col = debugLoc.getCol();
            StringRef file = debugLoc->getScope()->getFilename();
            StringRef dir = debugLoc->getScope()->getDirectory();
            errs() << "\tLocation: " << dir << "/" << file << ":"
      << line << ":" << col << "\n";
        }
    }
}
```

Listing 5. function that prints the location a variable is first found

analyze the branch instruction. The code for the `run` function is shown in listing 3.

To trace the variable used in the branch instructions (loops), we define a `traceVariableOrigin` function, shown in listing 6. This function looks at the values passed in, if the value is: 1) an argument 2) an `alloca` instruction 3) a global variable or 4) a `store` instruction, we likely found the origin of the variable; thus, it prints the variable name and location, and analyzes the def-use chain of the variable (lines 2-33). If it is an instruction, we get the operands of the instruction and pass the operands into the `traceVariableOrigin` function again (lines 34-41). Listing 4 shows our implementation of printing the variable name and listing 5 shows the code that prints the location of the value.

```
1  void traceVariableOrigin(Value *V) {
2      // If it's an argument, print and return
3      if (isa<Argument>(V)) {
4          errs() << "\tVariable originates as a function argument: "
        << *V << "\n";
5          printValueName(V);
6          printValueSourceLocation(V);
7          findDefUseChains(V);
8          return;
9      }
10     // If it's an alloca instruction, it's a local variable
11     if (AllocaInst *AI = dyn_cast<AllocaInst>(V)) {
12         errs() << "\tVariable originates from an alloca: " << *AI
      << "\n";
13         printValueName(V);
14         printValueSourceLocation(V);
15         findDefUseChains(V);
16         return;
17     }
18     // If it's a global variable
19     if (GlobalVariable *GV = dyn_cast<GlobalVariable>(V)) {
20         errs() << "\tVariable originates from a global variable: "
        << *GV << "\n";
21         printValueName(V);
22         printValueSourceLocation(V);
23         findDefUseChains(V);
24         return;
25     }
26     // If it's a store instruction
27     if (StoreInst *SI = dyn_cast<StoreInst>(V)) {
28         errs() << "\tVariable defined by store instruction: " << *
      SI << "\n";
29         printValueName(V);
30         printValueSourceLocation(V);
31         findDefUseChains(V);
32         return;
33     }
34     // If it's defined by an instruction, trace back its operands
35     if (Instruction *Inst = dyn_cast<Instruction>(V)) {
36         if (isInstructionInVector(traced_instructions, Inst)) {
37             return;
38         }
39         errs() << "\tTracing variable defined by instruction: " <<
        *Inst << "\n";
40         checkBeforeTrace(Inst);
41     }
42 }
```

Listing 6. Function traceVariableOrigin

```
void findDefUseChains(llvm::Value *Val) {
    errs() << "\tfindDefUseChains()\n";
    for (auto *User : Val->users()) {
        if (auto *instr = llvm::dyn_cast<llvm::Instruction>(User))
        {
            checkBeforeTrace(instr);
        }
    }
}
```

Listing 7. function findDefUseChains

When we trace a variable from the loop back to the origin, we may have missed instructions that assign user input to the variable. Thus, we analyze the def-use chain and pass the def-uses back into the `traceVariableOrigin` function.

Due to the calling of `traceVariableOrigin` many times through def-use chains, the function may be called multiple times on the same instruction. To avoid tracing the same instruction multiple times, we've coded the tracing feature into multiple functions, `traceVariableOrigin` and other functions that check if an instruction has already been traced before

```
void checkBeforeTrace(Instruction *Inst) {
    if (isInstructionInVector(traced_instructions, Inst) && isInstructionInVector(checked_seminal_inputs, Inst) ) {
        return;
    } else if (!isInstructionInVector(traced_instructions, Inst)) {
        traced_instructions.push_back(Inst);
    } else if (!isInstructionInVector(checked_seminal_inputs, Inst)) {
        checked_seminal_inputs.push_back(Inst);
    }
    checkSeminalInput(Inst);
    for (Use &U : Inst->operands()) {
        Value *Operand = U.get();
        if (seenOperands.find(Operand) != seenOperands.end()) {
            continue; // Skip duplicate operand
        }
        seenOperands.insert(Operand);
        checkSeminalInput(Operand);
        traceVariableOrigin(Operand); // Recursively trace the operand
    }
}
```

Listing 8. Function to check if instruction has been traced

```
void checkSeminalInput(Value *V) {
    if (auto *callInst = dyn_cast<CallInst>(V)) {
        if (Function *calledFunc = callInst->getCalledFunction()) {
            errs() << "\t  called function: " << calledFunc->getName() << "\n";
            if (calledFunc->getName().contains("scanf")) {
                errs() << "\t  --- SEMINAL INPUT ---\n";
                errs() << "\t   Value originates from scanf: " << *callInst << " --\n";
            }
            if (calledFunc->getName().contains("getc")) {
                errs() << "\t  --- SEMINAL INPUT ---\n";
                errs() << "\t   Value from a call to getc\n";
                Value *arg = callInst->getArgOperand(0);    // Get the first argument
                errs() << "\t  Argument passed to getc: " << *arg << "\n";
            }
            if (calledFunc->getName().contains("fopen")) {
                errs() << "\t  --- SEMINAL INPUT ---\n";
                errs() << "\t   Value from a call to fopen\n";
                Value *arg = callInst->getArgOperand(0);    // Get the first argument
                errs() << "\t  Argument passed to fopen: " << *arg << "\n";
            // other I/O APIs included in the code in GitHub repo
}}}}
```

Listing 9. Function to check if input is seminal input

we trace it. We use two vectors, `traced_instructions` and `checked_seminal_inputs`, to keep track of traced instructions and operands. Function `checkBeforeTrace` in listing 8 checks if the instruction has been in the two vectors. If it is not in the two vectors, we check for seminal input through function `checkSeminalInput`. Then we put the operands through `traceVariableOrigin` and trace and find variable origins.

The `checkSeminalInput` function checks if there is a function call to user input such as: `scanf`,`getc`,`fopen`, etc. If any is found, it prints out the function called and any additional information. An implementation can be found in listing 9

```
1  Tracing variable defined by instruction:   %1 = load i8, i8* %c, align 1, !dbg !94
2  Variable originates from an alloca:   %c = alloca i8, align 1
3  has name: 1, value name: c
4  Location: /home/elizabeth/Documents/csc512/project/csc512-course-proj/test2.c:10:10
5  findDefUseChains()
6  Tracing variable defined by instruction:   %conv = trunc i32 %call1 to i8, !dbg !92
7    called function: getc
8    --- SEMINAL INPUT ---
9     Value from a call to getc
10   Argument passed to getc:   %0 = load %struct._IO_FILE*, %struct._IO_FILE** %fp, align 8, !dbg !90
11 Tracing variable defined by instruction:   %call1 = call i32 @getc(%struct._IO_FILE* noundef %0), !dbg !92
```

Listing 10. LLVM pass output

5 Lessons and Experiences

This project required a lot of knowledge about how LLVM works and how the APIs from LLVM can be used. It also required you to have knowledge about how compilers work, including understanding intermediate representation and instructions and registers. Most of the above was taught in class, but writing a LLVM pass gives you hands on experience.

The LLVM documentation however, was not very helpful. The website for the documentation was a little outdated and made it hard for me to find the functions I need. Fortunately, large language models helped. Chatgpt actually helped me understand basic concepts about LLVM passes. There also were not too many code examples of LLVM passes I could learn from. Chatgpt was able to come up with short examples of working code for LLVM passes. This made learning how to code passes a lot easier for me.

6 Results

To test the LLVM pass, I have five testing C programs. Two of the programs are short code snippets: `test1.c` and `test2.c`. The other three are C programs that are interactive and each have more than 200 lines: `test3-snake-game.c`, `test4-library-management-system.c` and `test5-cafeteria-system.c`.

To run the test programs, simple replace `example.c` in listing 2 with the test file. The output of my pass documents the instructions it traced through. I like to see the full trace of the pass, thus I opted to keep most of the debug information. A snippet of the sample output can be found in listing 10. I output all instructions the pass traces through (line 1, 6, 11). Variable names and locations are also outputted (line 3). If the variable is a seminal input, a **SEMINAL INPUT** text is printed (line 8), it is followed by the function call or input value and more information (line 9).

Test files 3-5 are more interactive, thus I will briefly discuss the seminal inputs found by my LLVM pass. The outputs from the test files are included in the GitHub repository, in the subdirectory `test-outputs`.

In `test3-snake-game.c`, my pass found that the variable `direction`, value inputted at line 272 in the C program, is the seminal input of the program. The `direction` variable controls which direction the snake moves and is indeed a key input.

In `test4-library-management-system.c`, my pass found multiple inputs. One of the seminal inputs is the variable `choice` at line 56. The `choice` variable determines the option of the library system the user selects. Another seminal input identified is `librecord` at line 105. The `librecord` variable reads from a file `librecord.txt`. This file acts like a database and stores book information of the library system.

In `test5-cafeteria-system.c`, my pass also found multiple inputs. One of the seminal inputs is the variable `file` at line 357. The `file` variable reads from a file `menu.txt`, which has information about food and menu items in the cafeteria system. Interestingly, the `file` variable is related to two user inputs, my pass identified a `fopen` and a `scanf` function that updates the variable. If you check the code from lines 357-364, you'll find that there is indeed a `fopen` and a `fscanf`. Another one of the inputs identified by the pass is the `price` variable at line 432. Here the variable is set to the value specified in the file it is reading from.

7 Remaining Issues and Possible Challenges

I have tried to cover most of the user inputs from my test programs. But I could have missed some functions where user input is also read into the program. However, it would not be too complicated to add missed seminal input functions into the pass. We simple need to add the function names to `checkSeminalInput` to identify more input functions.