Project - Milestone-2

Haojie Zhou 's Submisssion: Part2 Seminal Input Features Detection in LLVM

Introduction

In our project, we aim to detect and analyze seminal input features in programs using LLVM passes. Seminal input features are those variables or elements in a program that are influenced by external input operations. Recognizing these features is vital for various use cases, including security analysis, debugging, and optimization.

Current Pathway

The current phase of the project involves the development and refinement of a static analysis tool capable of identifying and analyzing key control flow constructs within a given C program, as represented by its LLVM Intermediate Representation (IR).

Def-Use Analysis Pass: This pass analyzes the definition-use chains for each key point in the program, especially branching points or function pointers. If a variable directly influences a key point, it's marked. If the influence is indirect (e.g., computed using another variable influenced by input), that's noted as well. The main functions it includes are as follows:

1. **Control Flow Analysis**:
    - The tool accurately identifies both loops and switch statements within the LLVM IR.
    - For each loop encountered, the tool reports:
        - The header of the loop, even if unnamed.
        - The loop's depth within the nesting structure.
        - Exiting blocks of the loop, which are crucial for understanding loop termination conditions.
    - For switch statements, the tool enumerates cases and links them to their corresponding destination blocks.

2. **Def-Use Chain Analysis**:
    - The tool performs a preliminary analysis of the def-use chains for conditional branches within loops, aiding in the understanding of how different variables and their values influence the control flow.

3. **Verbose Output**:
    - A verbose log is generated that walks through each instruction in the IR, documenting the analysis process and findings in real-time. This serves as a detailed trace for developers to follow the tool's execution path.

Test Files

switch_test.c: A program that includes a switch function and a for loop.

```
#include <stdio.h>
int main() {
    int x;
    scanf("%d", &x);

    switch (x) {
        case 1:
            printf("One\n");
            break;
        case 2:
            printf("Two\n");
            break;
        default:
            printf("Other\n");
    }

    for (int i = 0; i < x; i++) {
        printf("%d ", i);
    }

    return 0;
}
```

Using `switch_test.c`, the tool demonstrates its capacity to dissect and interpret the control flow of a simple C program. The switch statement in `switch_test.c` is correctly identified, and the tool outputs the number of cases along with the case values. Although the destination blocks are unnamed, the tool successfully reports their presence, which is a step towards linking control flow elements back to the seminal input features.

The loop induced by the `for` statement in `switch_test.c` is also recognized, with its header, depth, and exiting blocks noted. The unnamed status of these blocks is typical of LLVM IR generated without optimization flags and debug information.

Test output:

- **Switch Statements**: The tool successfully identified the switch statement and its cases. It logged that the `scanf` function influences the switch variable `x`.
- **Loops**: The tool found the for-loop, determining that the loop's execution depends on the value of `x`, which is influenced by the `scanf` input.
- **Verbose Logging**: Each instruction was analyzed and logged, providing a comprehensive trace of the tool's execution and analysis process.

```
Analyzing function: main
Analyzing Instruction:   %1 = alloca i32, align 4
Analyzing Instruction:   %2 = alloca i32, align 4
Analyzing Instruction:   %3 = alloca i32, align 4
Analyzing Instruction:   store i32 0, ptr %1, align 4
```

```
Analyzing Instruction:   %4 = call i32 (ptr, ...) @scanf(ptr noundef
@.str, ptr noundef %2)
Analyzing Instruction:   %5 = load i32, ptr %2, align 4
Analyzing Instruction:   switch i32 %5, label %10 [
    i32 1, label %6
    i32 2, label %8
  ]
Switch statement found with 2 case(s).
Case value: 1, destination block: (unnamed)
Case value: 2, destination block: (unnamed)
Analyzing Instruction:   %7 = call i32 (ptr, ...) @printf(ptr noundef
@.str.1)
Analyzing Instruction:   br label %12
Analyzing Instruction:   %9 = call i32 (ptr, ...) @printf(ptr noundef
@.str.2)
Analyzing Instruction:   br label %12
Analyzing Instruction:   %11 = call i32 (ptr, ...) @printf(ptr noundef
@.str.3)
Analyzing Instruction:   br label %12
Analyzing Instruction:   store i32 0, ptr %3, align 4
Analyzing Instruction:   br label %13
Analyzing Instruction:   %14 = load i32, ptr %3, align 4
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   %15 = load i32, ptr %2, align 4
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   %16 = icmp slt i32 %14, %15
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
- Operand '  %16 = icmp slt i32 %14, %15' is a computed value.
  - Operand '  %14 = load i32, ptr %3, align 4' is a computed value.
    - Operand '  %3 = alloca i32, align 4' is a computed value.
      - Operand 'i32 1'
  - Operand '  %15 = load i32, ptr %2, align 4' is a computed value.
    - Operand '  %2 = alloca i32, align 4' is a computed value.
      - Operand 'i32 1'
Analyzing Instruction:   br i1 %16, label %17, label %23
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   %18 = load i32, ptr %3, align 4
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   %19 = call i32 (ptr, ...) @printf(ptr noundef
@.str.4, i32 noundef %18)
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   br label %20
```

```
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   %21 = load i32, ptr %3, align 4
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   %22 = add nsw i32 %21, 1
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   store i32 %22, ptr %3, align 4
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   br label %13, !llvm.loop !5
Loop found with header: (unnamed loop header)
Loop depth: 1
Loop exiting blocks: (unnamed)
Analyzing Instruction:   ret i32 0
```

Future Work

Semantics of I/O APIs: For complex scenarios (like fopen and getc), incorporate the semantics of the I/O operation. This would help understand how the size of a file or other such attributes influence the program's execution. Integration with Build Systems: The project can be further integrated with build systems to automate the process of applying these passes during the build phase.

Prerequisites

- LLVM >= 16.0 installed
- CMake installed
- C++ compiler (e.g., g++, clang)

Build the Pass

Clone the repo first if you haven't already:

```
git clone https://github.com/ncsu-csc512-project/part2-dev.git
```

Navigate to the root directory of the repo and run the following commands to build the pass:

```
export LLVM_DIR= # replace with your LLVM installation directory

mkdir build
cd build
cmake ..
make
```

Running the Pass

After building, you should have a `libDefUseAnalysisPass.so`and `libInputDetectionPass.so` file in your `build` directory. To run the pass in the `tests` file, use LLVM's `opt` tool as follows:

```
./run_tests.sh
```

Replace `complex_branch_test.bc` with the LLVM IR file you want to analyze.