Software Testing Project

Phase 1

Introduction

Goals

- Designing a simple random testing tool
- Gaining exposure to LLVM in general and the LLVM IR which is the intermediate representation
- used by LLVM
- Using LLVM to perform a sample testing

Constrains

- One or more int inputs. (the tester will generate values for these inputs)
- The inputs are between -100 and +100
- The inputs are defined at the beginning of the program in the form of a_i
- Can contain other int variables
- Can contain if statements

Design and Implementation

Steps

1. Start from the first basic block of the main function and generate a path using *generate_path()* function
2. Start from the first basic block in the generated path and perform the following (*traverse_path):*
   i. Analyze all the instruction in the current basic block (*analyze_instruction*)
   ii. Add all the variables to *all_variables* vector
   iii. Add input variables to *input_variables* vector
   iv. Create a vector for each input variable that shows the values the variable can have (the initial range is [-100, 100])
   v. Analyze *branch* and *cmp* instructions and determine what their result should be so the BB successor is the next BB in the path
   vi. Modify the range of values the input variables based on the result *branch* and *cmp* instructions should have
   vii. If there is a successor, move to the next BB
3. For each input variable, choose a random value from the range of values they can have and print it
4. Print the BB that the program has visited in the generated path

Generate Path

```
void generatePath(BasicBlock* BB)
```

1. If the current BB doesn't have any successors, stop
2. If the current BB has 1 successor, add it to the list and run the function with the new BB
3. If the current BB has 2 successors, choose one of them randomly, add it to the list and run the function with the new BB

```
void generatePath(BasicBlock* BB) {

  const Instruction *TInst = BB->getTerminator();
  unsigned NSucc = TInst->getNumSuccessors();
  if (NSucc == 1) {
      BasicBlock *Succ = TInst->getSuccessor(0);
      blocks.push_back(Succ);
      generatePath(Succ);
  }
  else if (NSucc>1) {
      unsigned rnd = std::rand() / (RAND_MAX/NSucc);
      BasicBlock *Succ = TInst->getSuccessor(rnd);
      blocks.push_back(Succ);
      generatePath(Succ);
  }
}
```

Analyze Instructions

```
void analyze_instruction(std::string str)
```

i. Add all new the variables to *all_variables* vector

ii. Add new input variables to *input_variables* vector

iii. Create a vector for each new input variable that shows the values the variable can have (the initial range is [-100, 100])

iv. Analyze *branch* and *cmp* instructions and determine what their result should be so the BB successor is the next BB in the path

v. Modify the range of values the input variables based on the result *branch* and *cmp* instructions should have

```
void analyze_instruction(Instruction* inst) {

    std::string str;
    llvm::raw_string_ostream(str) << (*inst);

    if (isa<AllocaInst> (inst)){
        ...
    }

    else if (isa<LoadInst> (inst)) {
        ...
    }

    else if (isa<StoreInst> (inst)) {
        ...
    }

    else if (isa<CmpInst> (inst)) {
        ...
    }

    else if (isa<BranchInst> (inst)) {
        ...
    }

    else if (isa<BinaryOperator> (inst)) {
        ...
    }
}
```

Output

After traversing the generated path, the following determine the output of the program:

1. For each input variable, choose a random value from the range of values they can have and print it
2. Print the BB that the program has visited in the generated path

```
// print arguments with random valid values
for (int i=0; i<input_names.size(); i++) {
    int rnd_index = (rand()%input_limits[i].size());
    llvm::outs() << input_names[i] << " = " <<
std::to_string(input_limits[i][rnd_index]) << "\n";
}

// print the path
llvm::outs() << "Sequence of Basic Blocks:\n";
for (auto &BB : blocks)
    llvm::outs() << getSimpleNodeLabel(BB) << "\n";
```

Steps to build and run the code

1. Generating human-readable LLVMIR files from test files:

```
clang -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
clang++ -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
clang-10 -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
clang++-10 -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
```

2. Generate CFG by LLVM:

```
opt -dot-cfg test1.ll
```

3. Generate a PDF from CFG:

```
./allfigs2pdf
```

Steps to build and run the code (contd.)

4. Compiling the RandomPath LLVM pass:

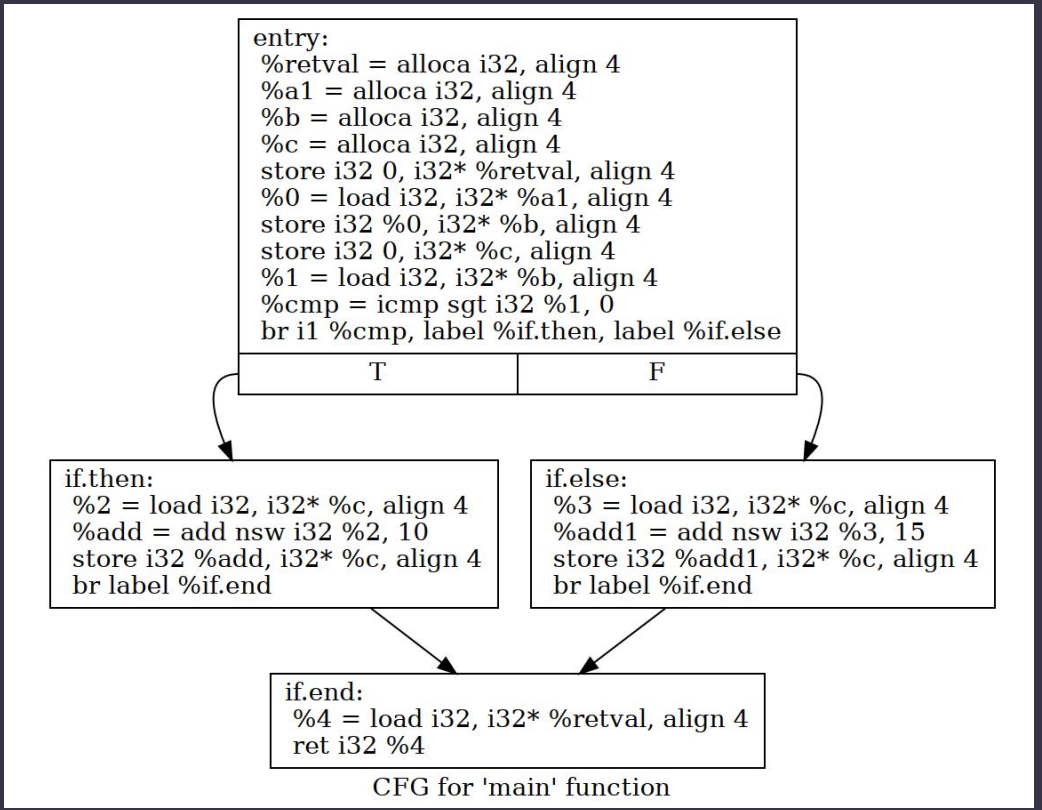
```
clang++-10 -o RandomPath RandomPath.cpp `llvm-config
--cxxflags` `llvm-config --ldflags` `llvmconfig --libs`
-lpthread -lncurses -ldl
```

5. Running LLVM pass on the LLVM IR files:

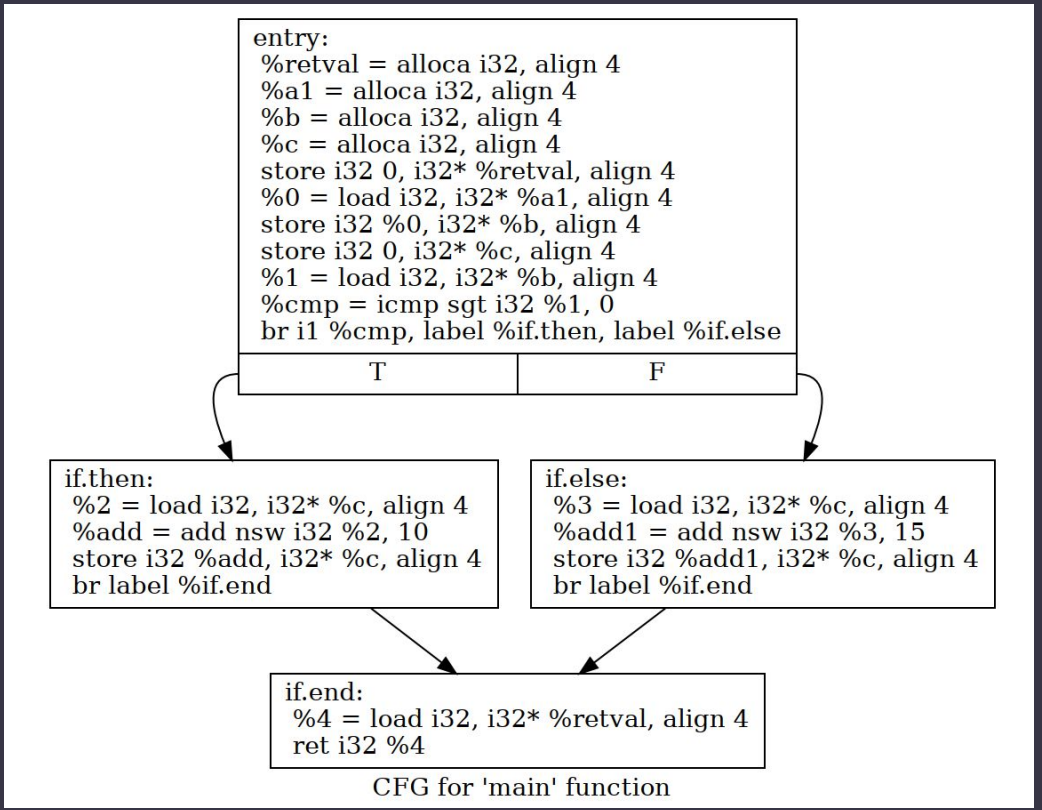
```
./RandomPath test1.ll
./RandomPath test2.ll
./RandomPath test3.ll
```

Examples (1)

```
int main() {
    int a1;
    int b = a1;
    int c = 0;

    if (b > 0)
      c += 10;
    else
      c += 15;
}
```



Examples (1)



CFG for 'main' function

Examples (1)

Iteration 1:

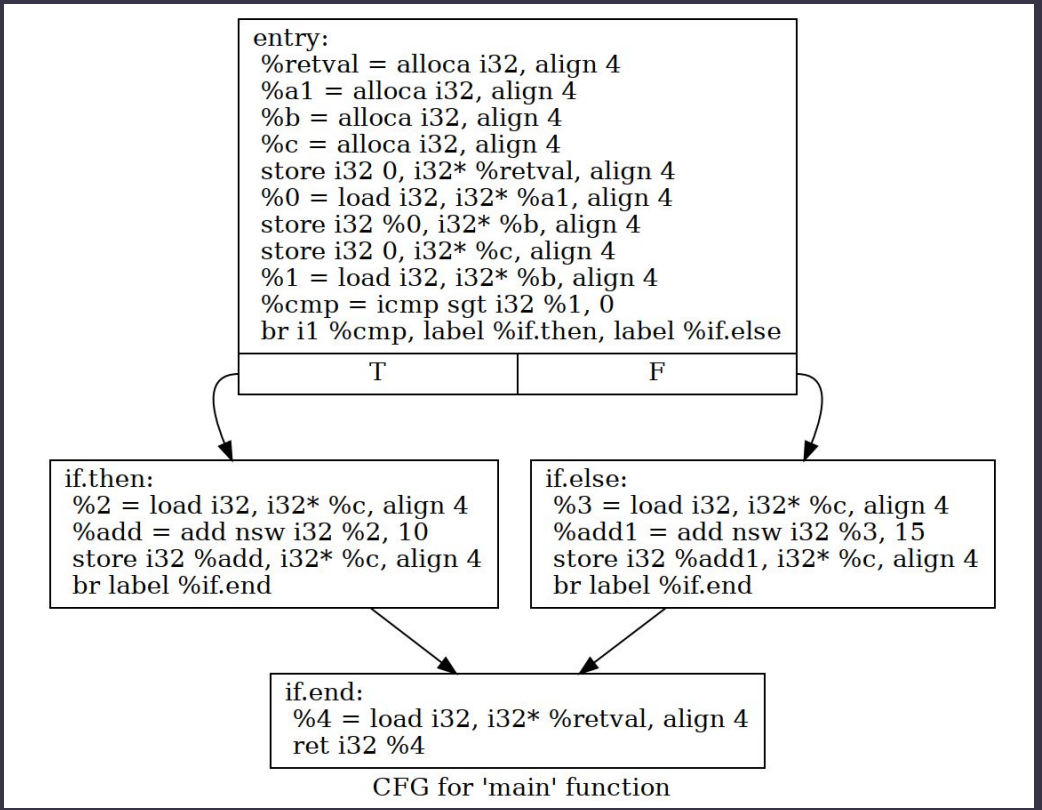
a1 = -1

Sequence of Basic Blocks:

entry

if.else

if.end



CFG for 'main' function

Examples (1)

Iteration 1:

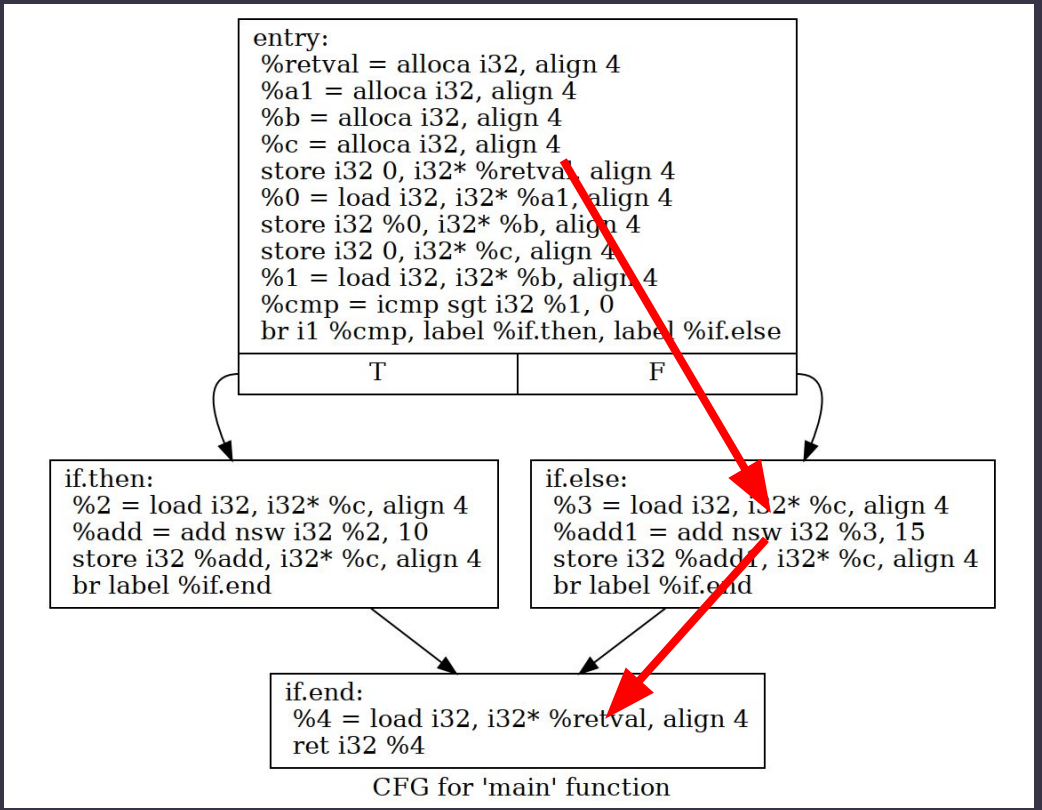
a1 = -1

Sequence of Basic Blocks:

entry

if.else

if.end



CFG for 'main' function

Examples (1)

Iteration 1:

a1 = -1

Sequence of Basic Blocks:

entry

if.else

if.end

Iteration 2:

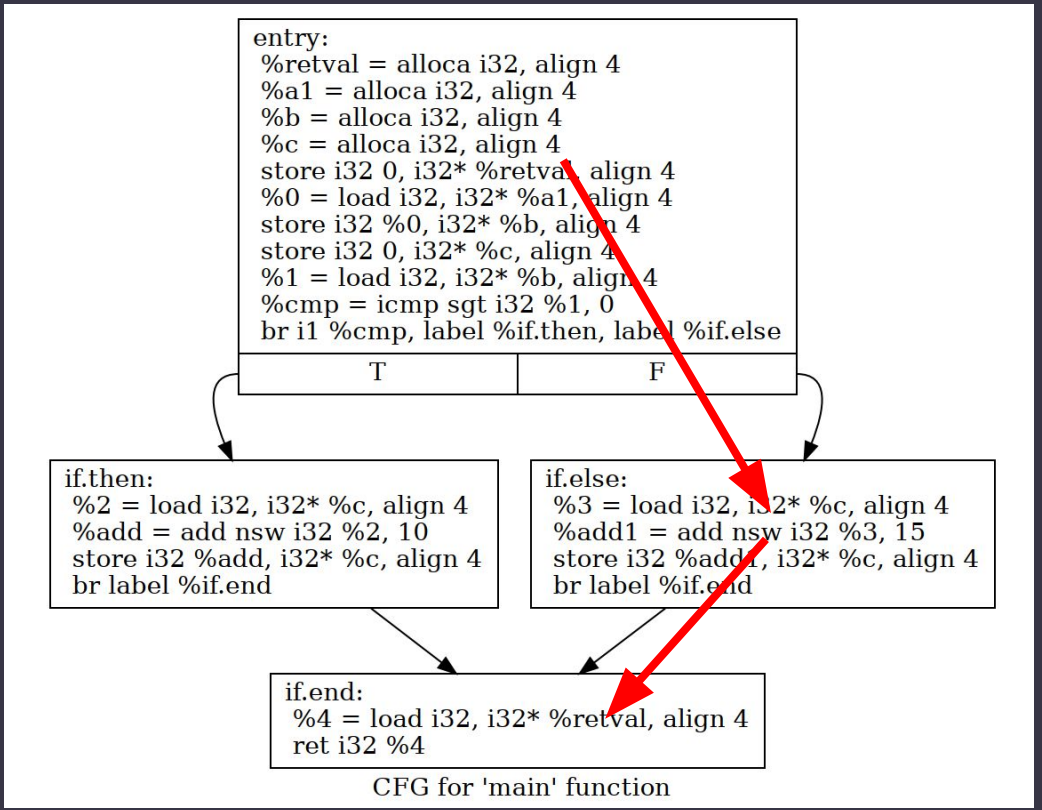
a1 = 16

Sequence of Basic Blocks:

entry

if.then

if.end



CFG for 'main' function

Examples (1)

Iteration 1:

a1 = -1

Sequence of Basic Blocks:

entry

if.else

if.end

Iteration 2:

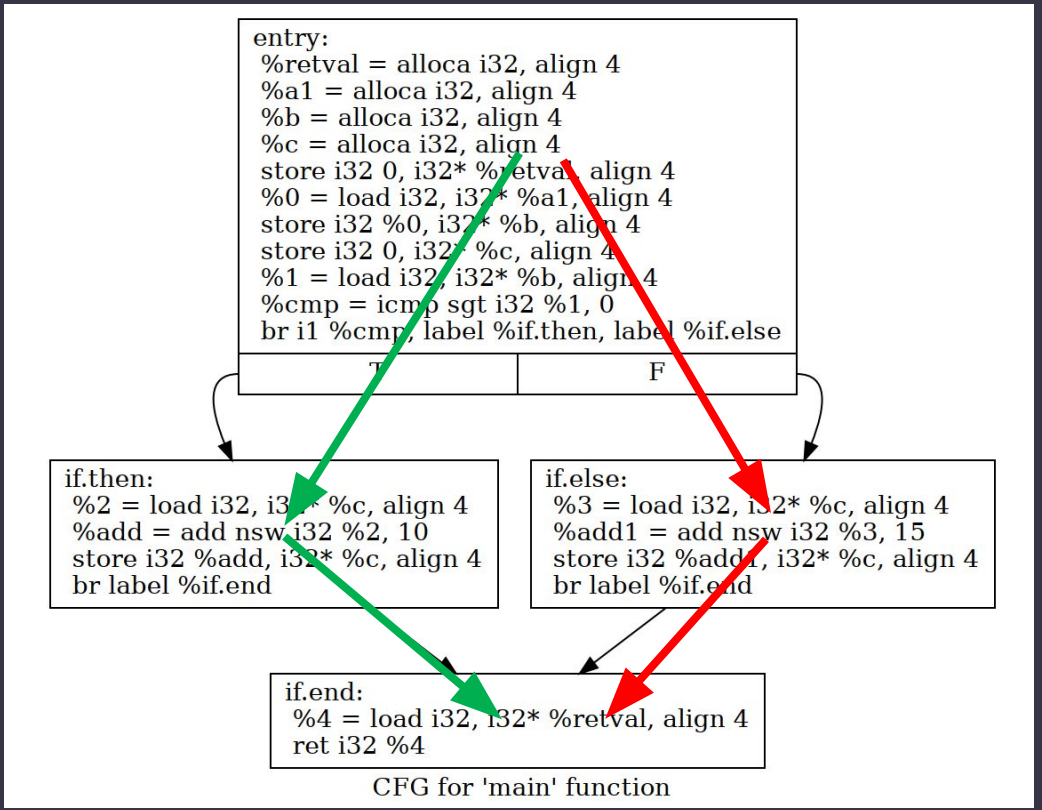
a1 = 16

Sequence of Basic Blocks:

entry

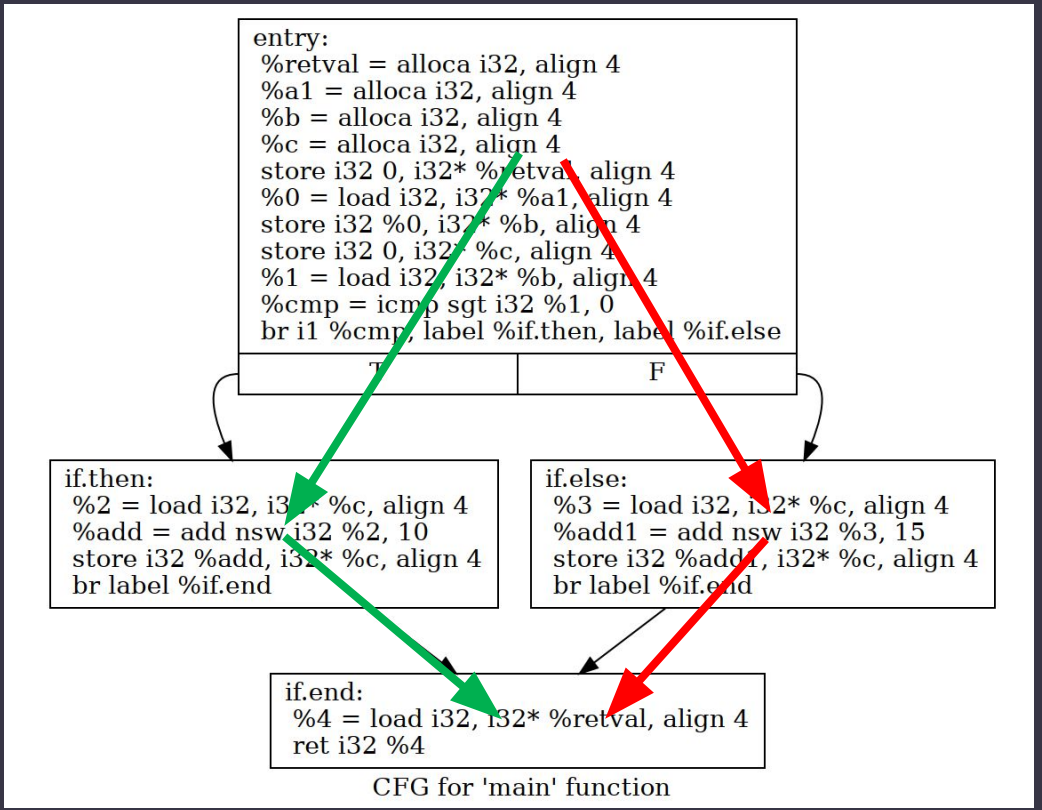
if.then

if.end



CFG for 'main' function

Examples (1)



CFG for 'main' function

Examples (1)

Iteration 3:

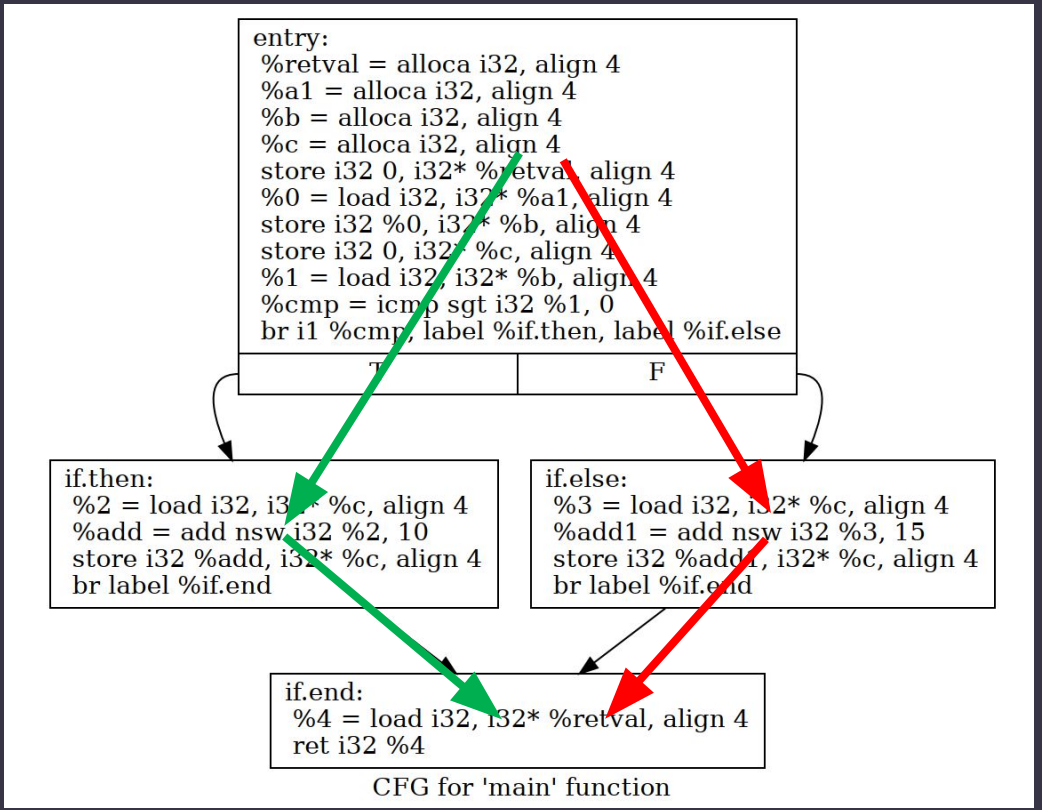
a1 = 50

Sequence of Basic Blocks:

entry

if.then

if.end



CFG for 'main' function

Examples (1)

Iteration 3:

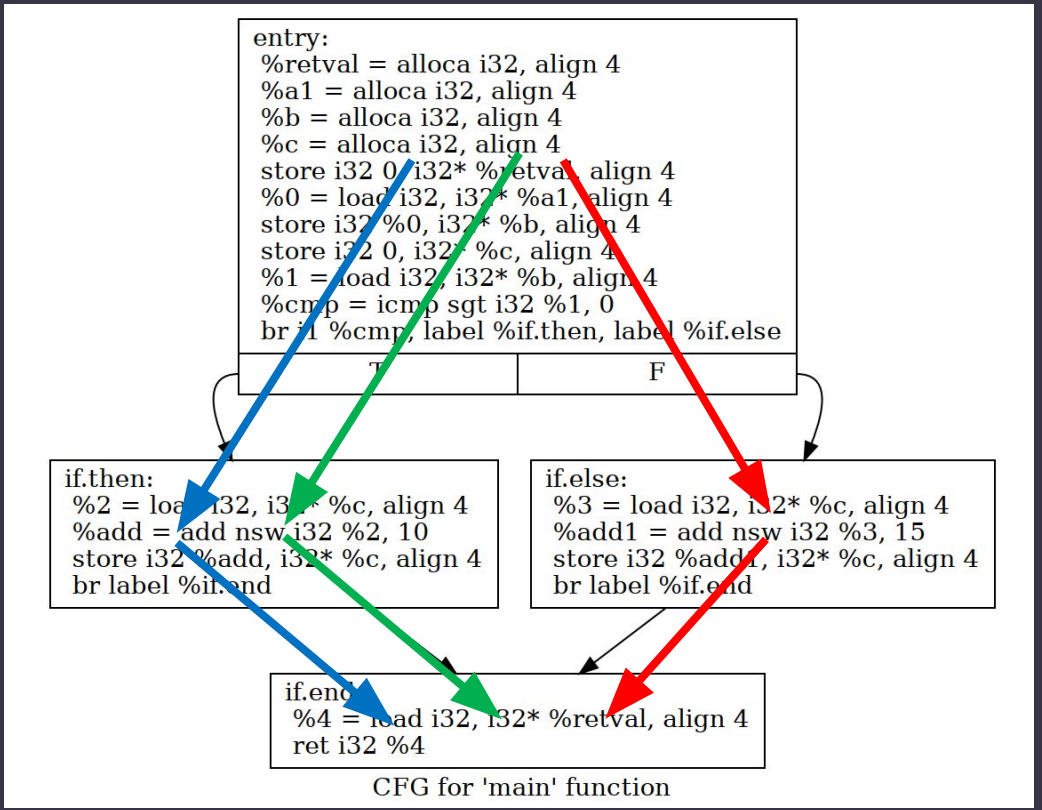
a1 = 50

Sequence of Basic Blocks:

entry

if.then

if.end



CFG for 'main' function

Examples (1)

Iteration 3:

a1 = 50

Sequence of Basic Blocks:

entry

if.then

if.end

Iteration 4:

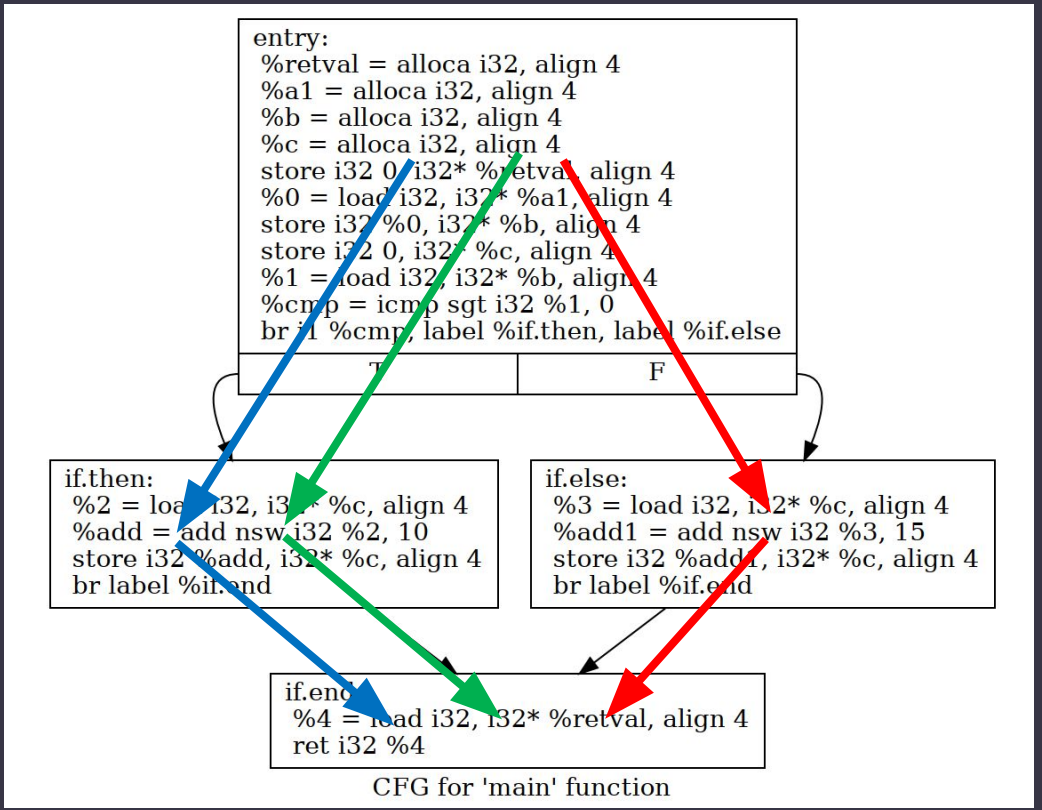
a1 = -63

Sequence of Basic Blocks:

entry

if.else

if.end



CFG for 'main' function

Examples (1)

Iteration 3:

a1 = 50

Sequence of Basic Blocks:

entry

if.then

if.end

Iteration 4:

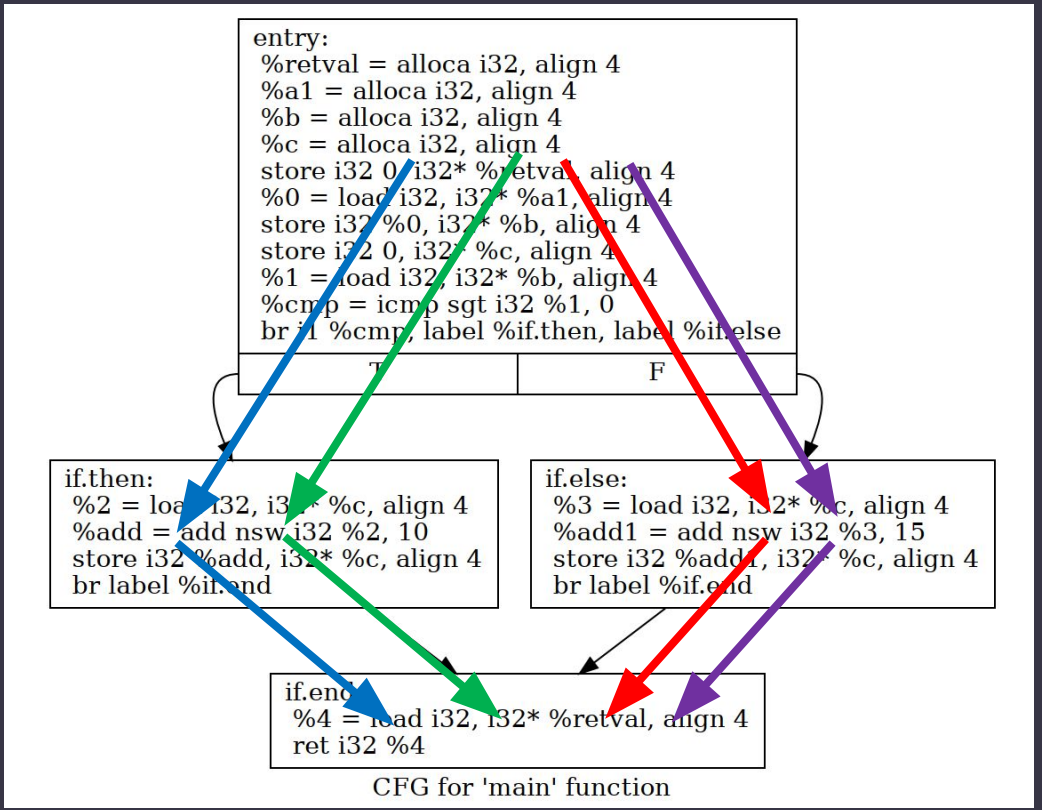
a1 = -63

Sequence of Basic Blocks:

entry

if.else

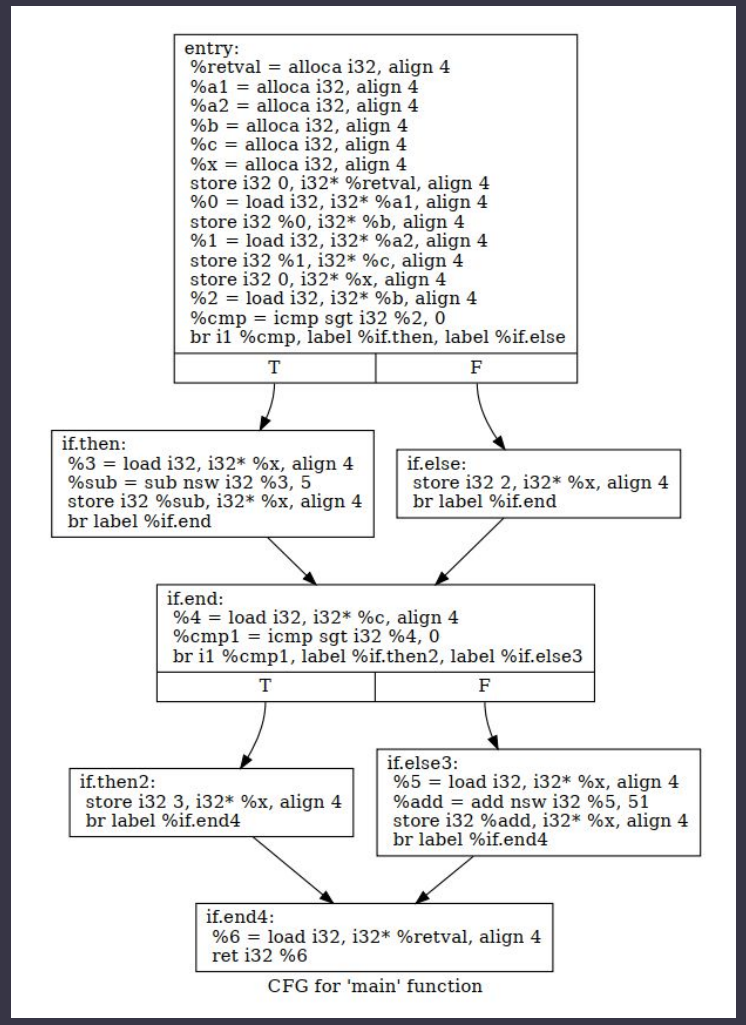
if.end



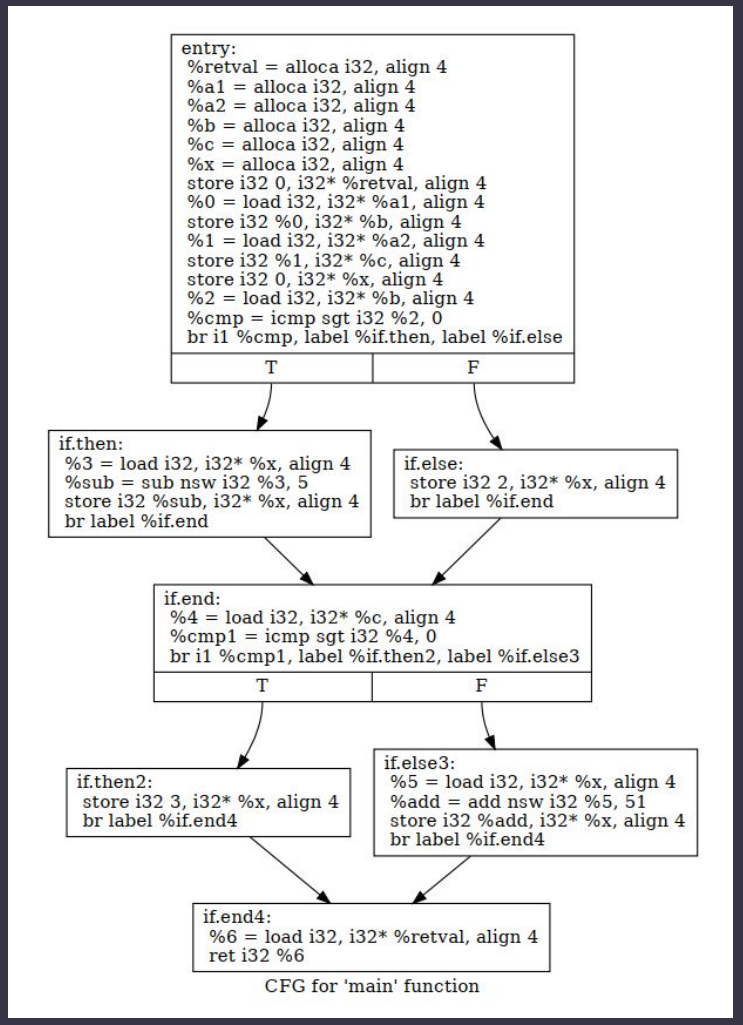
CFG for 'main' function

Examples (2)

```
int main() {

    int a1;
    int a2;
    int b = a1;
    int c = a2;
    int x = 0;

    if (b > 0){
      x -= 5;
    } else {
      x = 2;
    }

    if (c > 0){
      x = 3;
    } else {
      x += 51;
    }
}
```



Examples (2)


entry:
%retval = alloca i32, align 4
%a1 = alloca i32, align 4
%a2 = alloca i32, align 4
%b = alloca i32, align 4
%c = alloca i32, align 4
%x = alloca i32, align 4
store i32 0, i32* %retval, align 4
%0 = load i32, i32* %a1, align 4
store i32 %0, i32* %b, align 4
%1 = load i32, i32* %a2, align 4
store i32 %1, i32* %c, align 4
store i32 0, i32* %x, align 4
%2 = load i32, i32* %b, align 4
%cmp = icmp sgt i32 %2, 0
br i1 %cmp, label %if.then, label %if.else
T
F
if.then:
%3 = load i32, i32* %x, align 4
%sub = sub nsw i32 %3, 5
store i32 %sub, i32* %x, align 4
br label %if.end
if.else:
store i32 2, i32* %x, align 4
br label %if.end
if.end:
%4 = load i32, i32* %c, align 4
%cmp1 = icmp sgt i32 %4, 0
br i1 %cmp1, label %if.then2, label %if.else3
T
F
if.then2:
store i32 3, i32* %x, align 4
br label %if.end4
if.else3:
%5 = load i32, i32* %x, align 4
%add = add nsw i32 %5, 51
store i32 %add, i32* %x, align 4
br label %if.end4
if.end4:
%6 = load i32, i32* %retval, align 4
ret i32 %6
CFG for 'main' function

Examples (2)

Iteration 1:

a1 = -49

a2 = -88

Sequence of Basic Blocks:

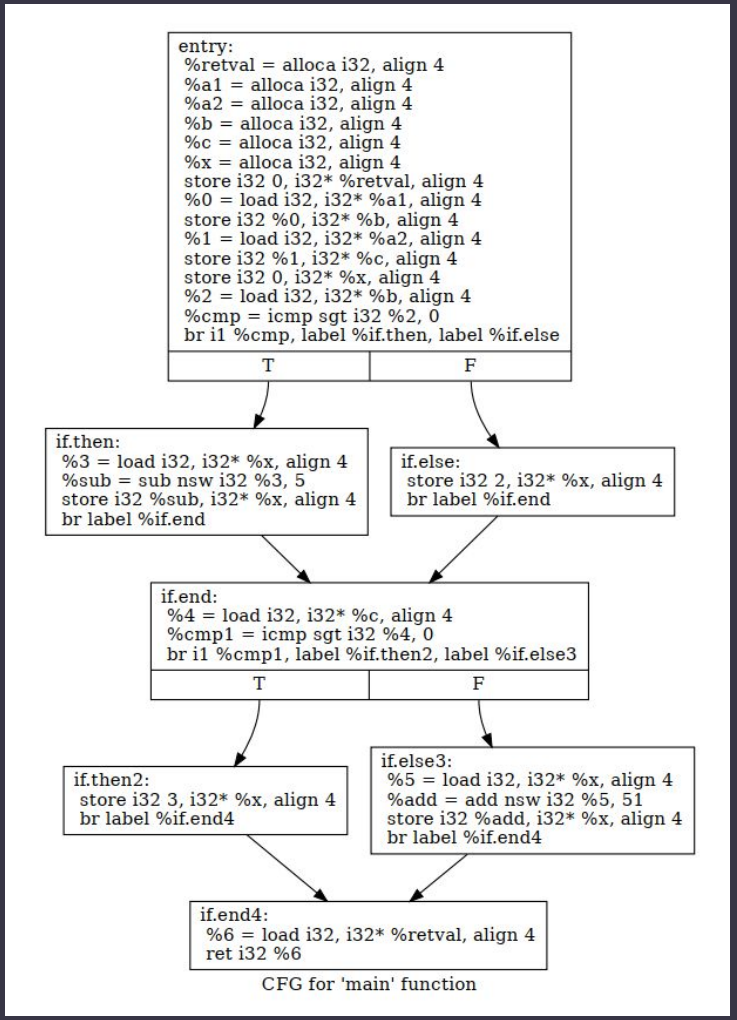
entry

if.else

if.end

if.else3

if.end4



Examples (2)

Iteration 1:

a1 = -49

a2 = -88

Sequence of Basic Blocks:

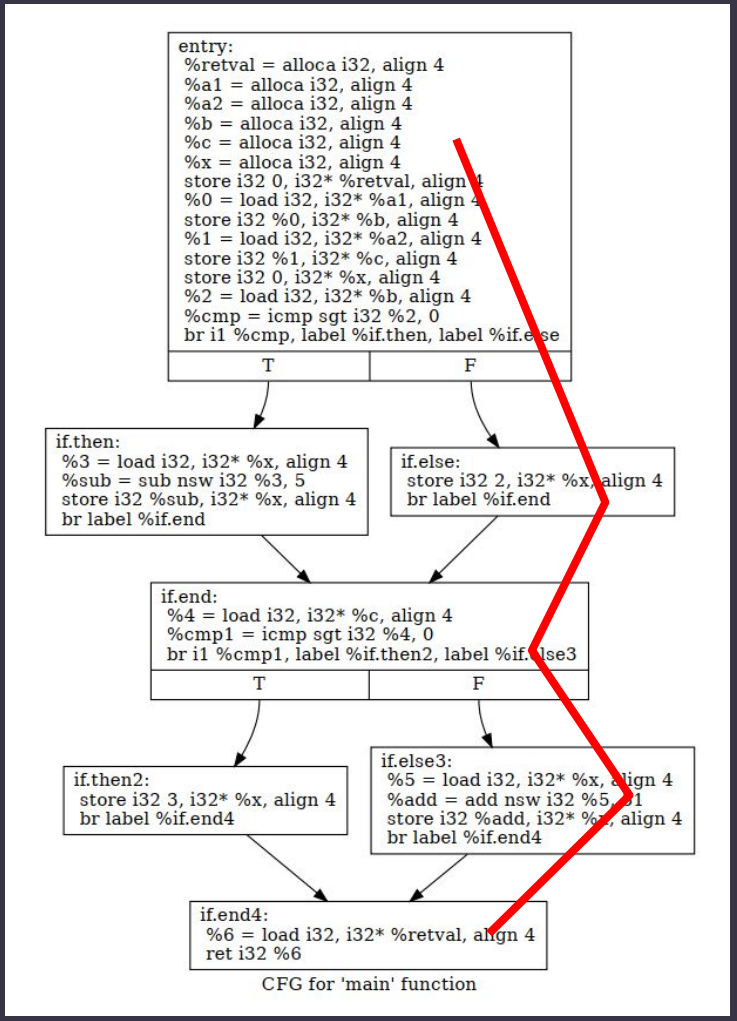
entry

if.else

if.end

if.else3

if.end4



Examples (2)

Iteration 1:

a1 = -49

a2 = -88

Sequence of Basic Blocks:

entry

if.else

if.end

if.else3

if.end4

Iteration 2:

a1 = 89

a2 = -40

Sequence of Basic Blocks:

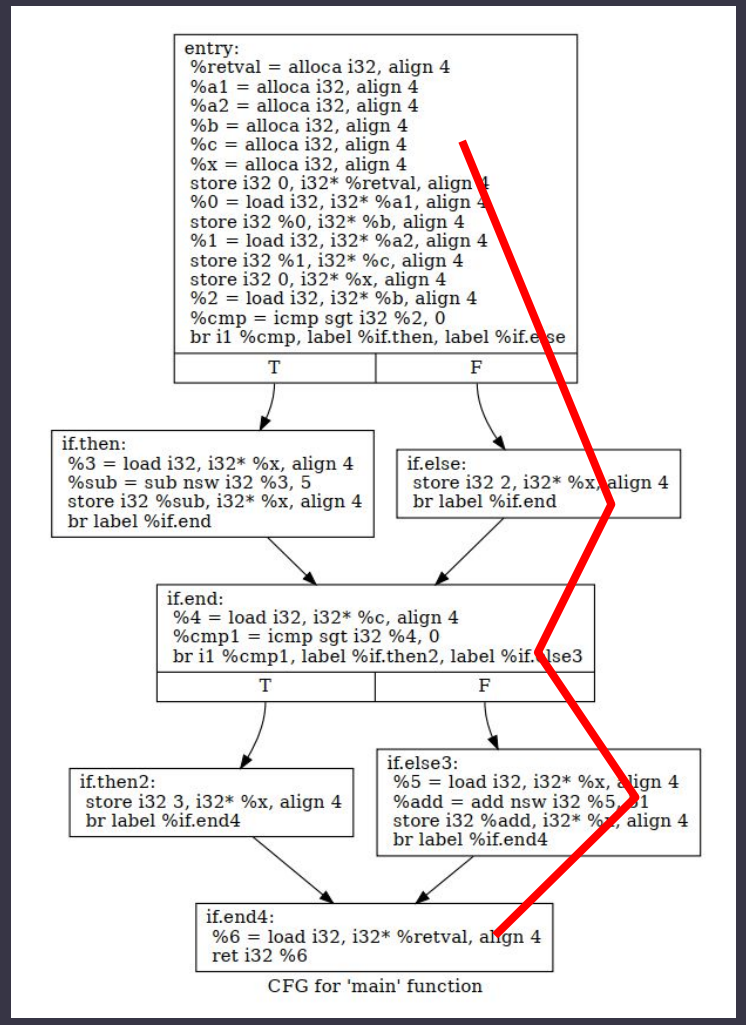
entry

if.then

if.end

if.else3

if.end4



CFG for 'main' function

Examples (2)

Iteration 1:

a1 = -49

a2 = -88

Sequence of Basic Blocks:

entry

if.else

if.end

if.else3

if.end4

Iteration 2:

a1 = 89

a2 = -40

Sequence of Basic Blocks:

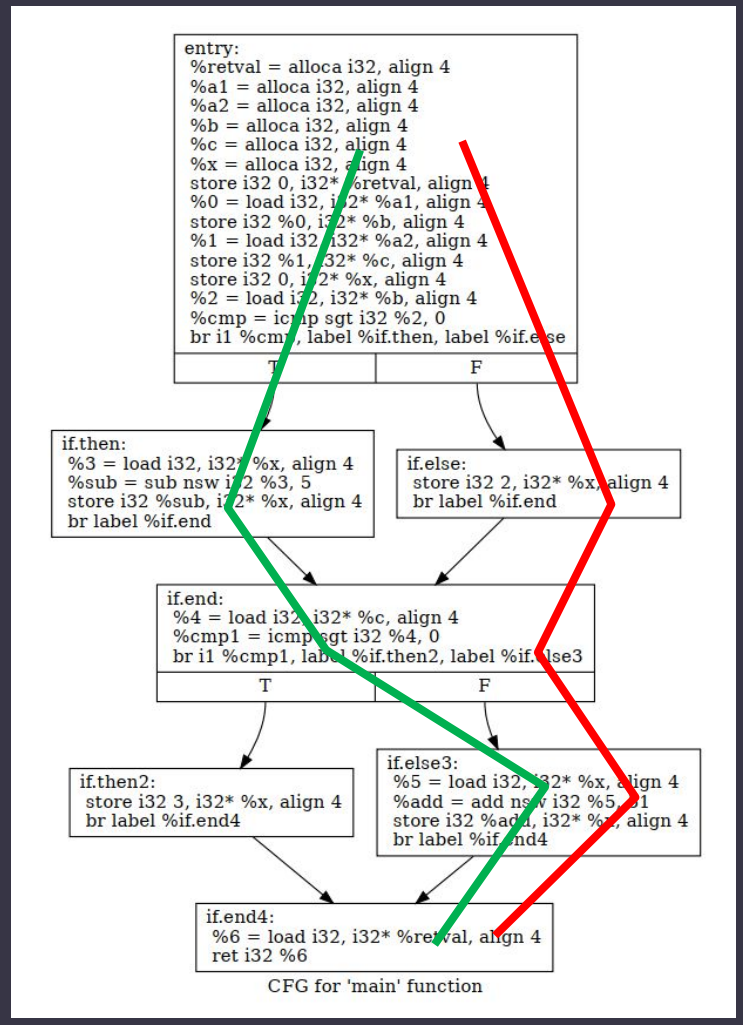
entry

if.then

if.end

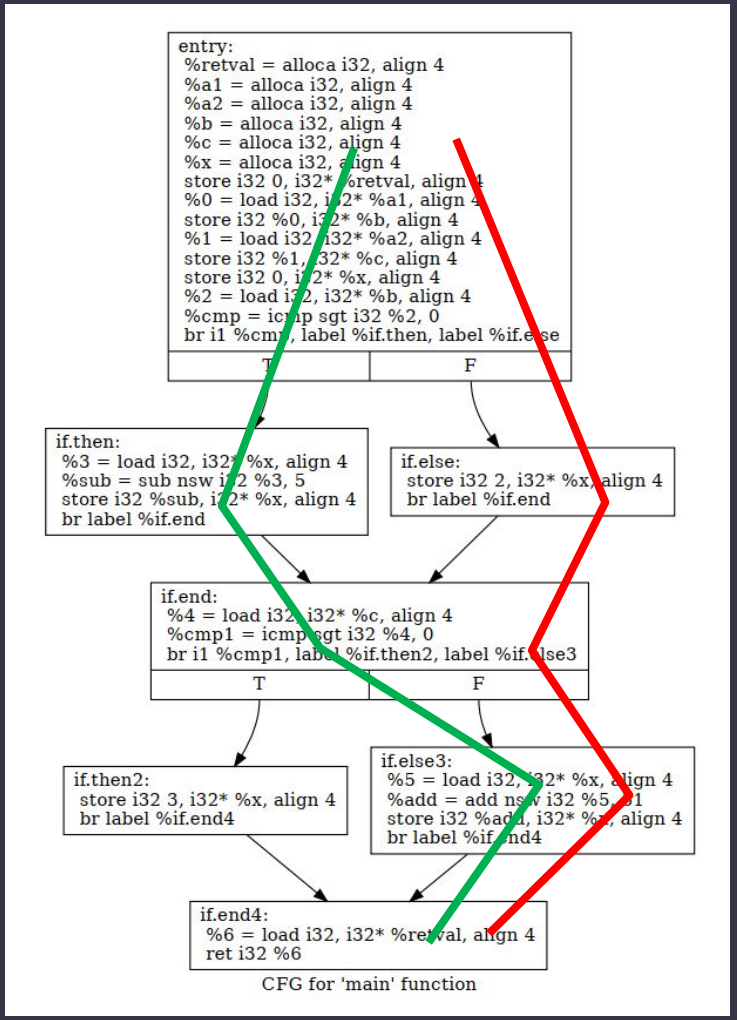
if.else3

if.end4



CFG for 'main' function

Examples (2)


entry:
%retval = alloca i32, align 4
%a1 = alloca i32, align 4
%a2 = alloca i32, align 4
%b = alloca i32, align 4
%c = alloca i32, align 4
%x = alloca i32, align 4
store i32 0, i32* %retval, align 4
%0 = load i32, i32* %a1, align 4
store i32 %0, i32* %b, align 4
%1 = load i32, i32* %a2, align 4
store i32 %1, i32* %c, align 4
store i32 0, i32* %x, align 4
%2 = load i32, i32* %b, align 4
%cmp = icmp sgt i32 %2, 0
br i1 %cmp, label %if.then, label %if.else
T
F
if.then:
%3 = load i32, i32* %x, align 4
%sub = sub nsw i32 %3, 5
store i32 %sub, i32* %x, align 4
br label %if.end
if.else:
store i32 2, i32* %x, align 4
br label %if.end
if.end:
%4 = load i32, i32* %c, align 4
%cmp1 = icmp sgt i32 %4, 0
br i1 %cmp1, label %if.then2, label %if.else3
T
F
if.then2:
store i32 3, i32* %x, align 4
br label %if.end4
if.else3:
%5 = load i32, i32* %x, align 4
%add = add nsw i32 %5, 1
store i32 %add, i32* %x, align 4
br label %if.end4
if.end4:
%6 = load i32, i32* %retval, align 4
ret i32 %6
CFG for 'main' function

Examples (2)

Iteration 3:

a1 = 40

a2 = 29

Sequence of Basic Blocks:

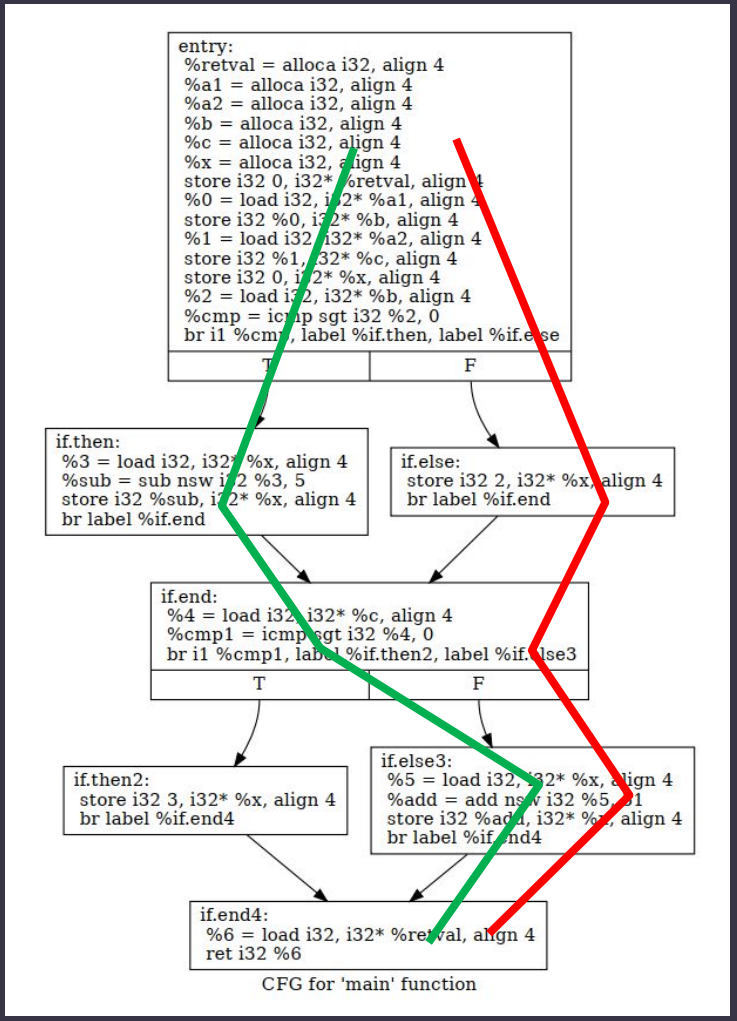
entry

if.then

if.end

if.then2

if.end4



Examples (2)

Iteration 3:

a1 = 40

a2 = 29

Sequence of Basic Blocks:

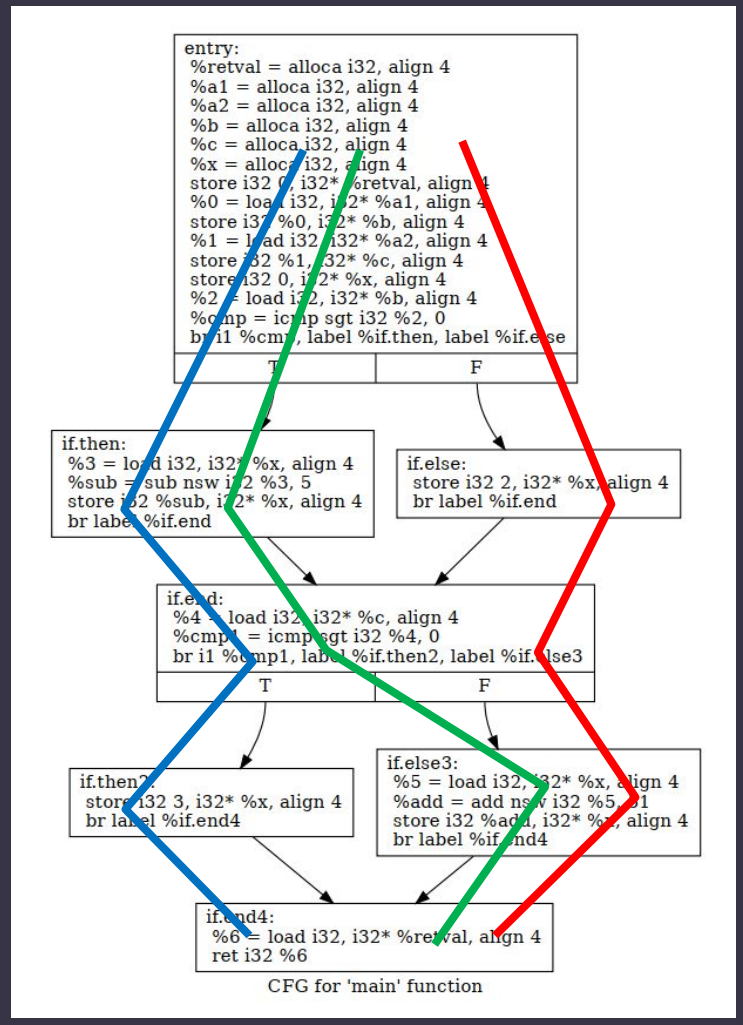
entry

if.then

if.end

if.then2

if.end4



CFG for 'main' function

Examples (2)

Iteration 3:

a1 = 40

a2 = 29

Sequence of Basic Blocks:

entry

if.then

if.end

if.then2

if.end4

Iteration 4:

a1 = -67

a2 = 82

Sequence of Basic Blocks:

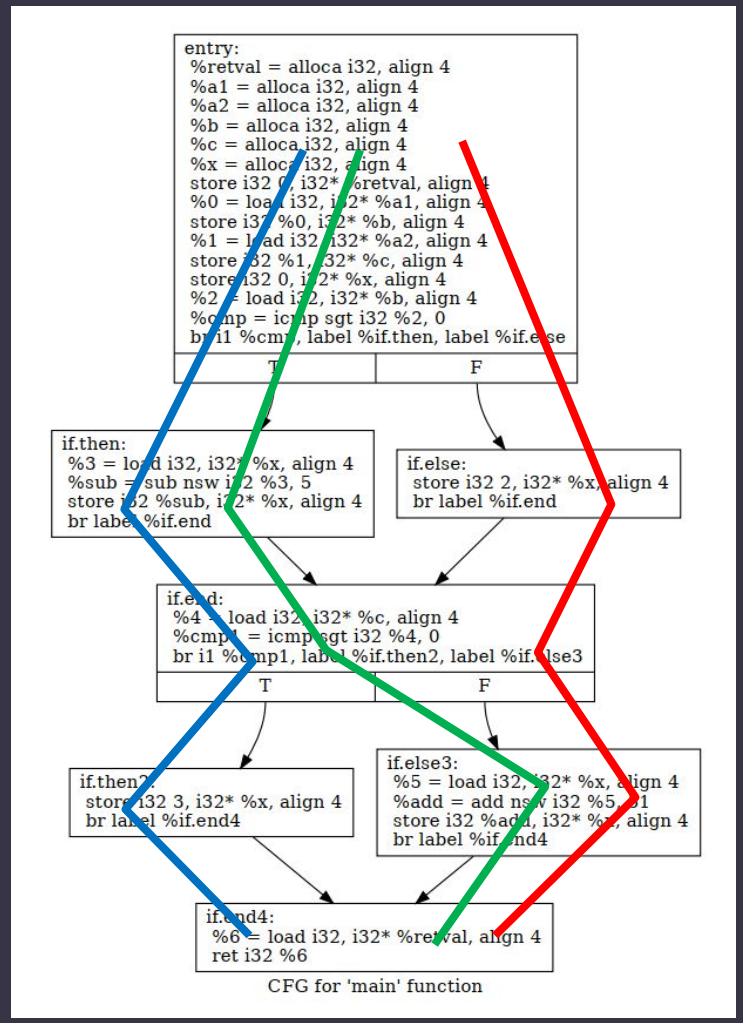
entry

if.else

if.end

if.then2

if.end4



CFG for 'main' function

Examples (2)

Iteration 3:

a1 = 40

a2 = 29

Sequence of Basic Blocks:

entry

if.then

if.end

if.then2

if.end4

Iteration 4:

a1 = -67

a2 = 82

Sequence of Basic Blocks:

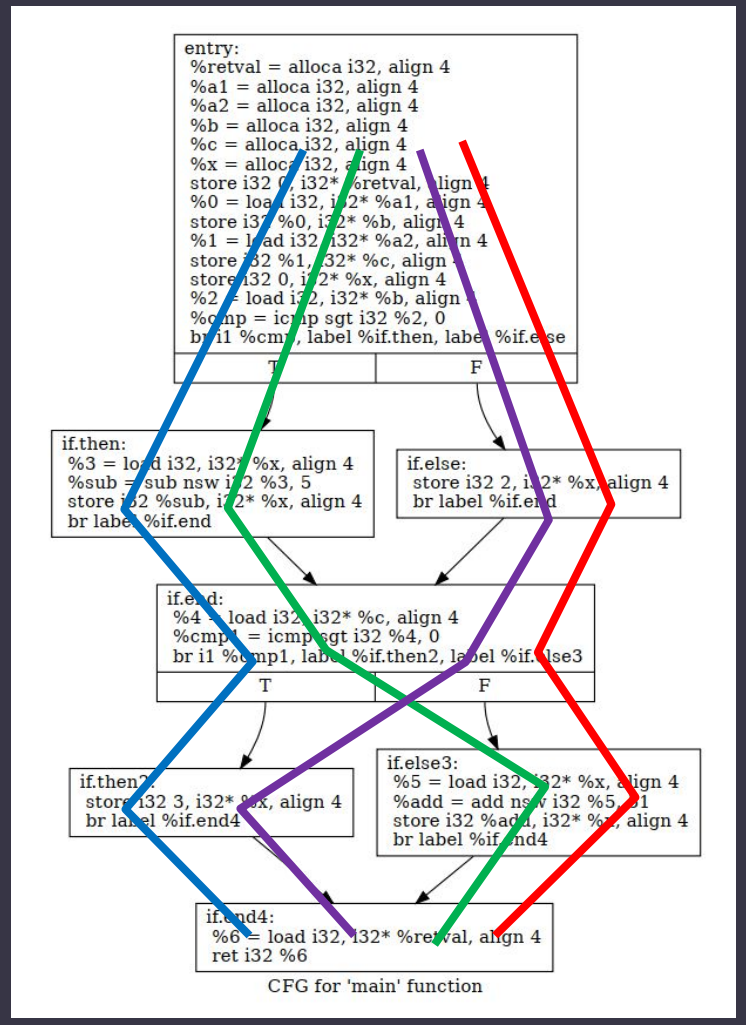
entry

if.else

if.end

if.then2

if.end4



CFG for 'main' function

Phase 2

Introduction

Goals

- Designing a simple fuzz testing tool
- Gaining exposure to LLVM in general and the LLVM IR which is the intermediate representation
- used by LLVM
- Using LLVM to perform a sample testing

Constrains

- One or more int inputs. (the tester will generate values for these inputs)
- The inputs are between -100 and +100
- The inputs are defined at the beginning of the program in the form of ai
- Can contain other int variables
- Can contain if statements
- Can contain binary operations (add, sub, mul, div)
- Can contain loops (while, for, …)

Design and Implementation

1. **Finding all the basic blocks in the .ll file:**

```
for (auto &F: *M)
        for (auto &BB: F)
         allBlocks.insert(std::pair<std::string,BasicBlock* >(F.getName(),&BB));
```

2. **Iterations:**

Running fuzz testing function for 5 iterations, or until the coverage reaches 100%.

The program calls "*generatePath*" function, then the program prints the following:

i. The name of the input variables and their generated values.

ii. The names of the blocks that have been visited in the path. The blocks are stored in the "*path_of_blocks*" variable.

iii. The block coverage: (100**visitedBlocks*/*AllBlocks*)%. "*visitedBlockes*" are set in "*generatePath*" function.

Then the program calls *"fuzzing"* function to generate new values for the input variable for the next iteration.

In the first iteration, the values of the input variables are generated randomly.

Iterations

```
for (int i=0; i<5 && (visitedBlocks.size()<allBlocks.size()); i++) {
        for (auto &F: *M)
            if (strncmp(F.getName().str().c_str(),"main",4) == 0){
                path_of_blocks.clear();
                visitedBlocksInPath.clear();
                BasicBlock* BB = dyn_cast<BasicBlock>(F.begin());
                generatePath(BB);
            }

        // output
        for (int i=0; i<input_variables.size(); i++)
            llvm::outs() << input_variables[i].first << " = " << input_variables[i].second <<"\n";

        llvm::outs() << "Sequence of Basic Blocks:\n";

        for (auto &BB : path_of_blocks)
            llvm::outs() << getSimpleNodeLabel(BB) << "\n";

        llvm::outs() << "block coverage: " << visitedBlocks.size()*100/allBlocks.size() << "%\n\n";

        fuzzing(); // calls the fuzzing function to change the input variable
}
```

Generate Path

3. **Generate Path:**

   i. The current basic block is added to “visitedBlocks” and “visitedBlocksInPath”

   ii. Then every instruction within the basic block gets analyzed. If the basic block has two successors, the successor gets determined during the analysis

   iii. If the basic block has a successor, “GeneratePath” is called for the determined successor (the tester moves to the next basic block)

Generate Path (contd.)

```
void generatePath(BasicBlock* BB) {

    path_of_blocks.push_back(BB);

    visitedBlocks.insert(std::pair<std::string,BasicBlock* >(BB->getParent()->getName(),BB));
    visitedBlocksInPath.insert(std::pair<std::string,BasicBlock* >(BB->getParent()->getName(),BB));

    // Analyze the block
    global_succ_name = "";
    for (auto &I: *BB) {
        std::string str;
        llvm::raw_string_ostream(str) << I;
        analyze_instruction(&I);
    }
    const Instruction *TInst = BB->getTerminator();
    unsigned NSucc = TInst->getNumSuccessors();
    if (NSucc == 0)
        return;

    else if (NSucc == 1){
        BasicBlock *Succ = TInst->getSuccessor(0);
        generatePath(Succ);
    }

    else {

        // Find succ number
        int succ_number = 0;
        if (global_succ_name == getSimpleNodeLabel(TInst->getSuccessor(1)))
            succ_number = 1;

        BasicBlock *Succ = TInst->getSuccessor(succ_number);

        generatePath(Succ);
    }
}
```

Analyze Instructions

4. **Analyze Instructions:**

   i. allocation: a new variable gets created, initialized and added to the variable sets. If the variable is an input variable, the initialized value is a random number between -100 and 100

   ii. load: creates a new variable and sets its value to another variable's value

   iii. store: stores a constant int or the value of a variable into another variable

   iv. cmp: compares two variables, one variable and a constant int, or two constant ints with the given operand. most of the work is done in "comparison_analysis" function

   v. branch: determines the basic block successor based on the comparison variable, and stores the successor's name in a global variable

   vi. binary operator: performs binary operations between two variables, one variable and one constant, or two constants, and stores the result in a new variable

Analyze Instructions (contd.)

```
void analyze_instruction(Instruction* inst) {

    std::string str;
    llvm::raw_string_ostream(str) << (*inst);

    if (isa<AllocaInst> (inst)){
        ...
    }

    else if (isa<LoadInst> (inst)) {
        ...
    }

    else if (isa<StoreInst> (inst)) {
        ...
    }

    else if (isa<CmpInst> (inst)) {
        ...
    }

    else if (isa<BranchInst> (inst)) {
        ...
    }

    else if (isa<BinaryOperator> (inst)) {
        ...
    }
}
```

Fuzzing

5. **Fuzzing:**

After generating the path and running the instruction analysis, it's time to change the the values of the variables using fuzzing. In this project we assume that for generating new values we have 3 different choices:

a) Negate: in this situation the fuzzing algorithm will change the sign of previous number.

```
if (operation == "negate") {
    new_fuzz_value = (-prev_value);
}
```

b) Change Digit: In this operation we have two situations. If the number has 1 digit, the fuzz algorithm changes the previous number to another single digit number. If the number has 2 digits, the algorithm changes one of the digits.

```
else if (operation == "change_digit") {
    if (std::abs(prev_value) < 10){
        ...
    }
    else {
        ...
    }
}
```

Fuzzing (contd.)

c. Add or Remove Digit: In this operation one of the digits is added to or removed from the number. In this operation we also have two situations. If the number has only one digit, the new number will derive from adding a new number to right or left. And if the number has two digits, the new number will derive from removing one of the digits. For example:

   i. if a1 = 1: the next value can be 12 or 21

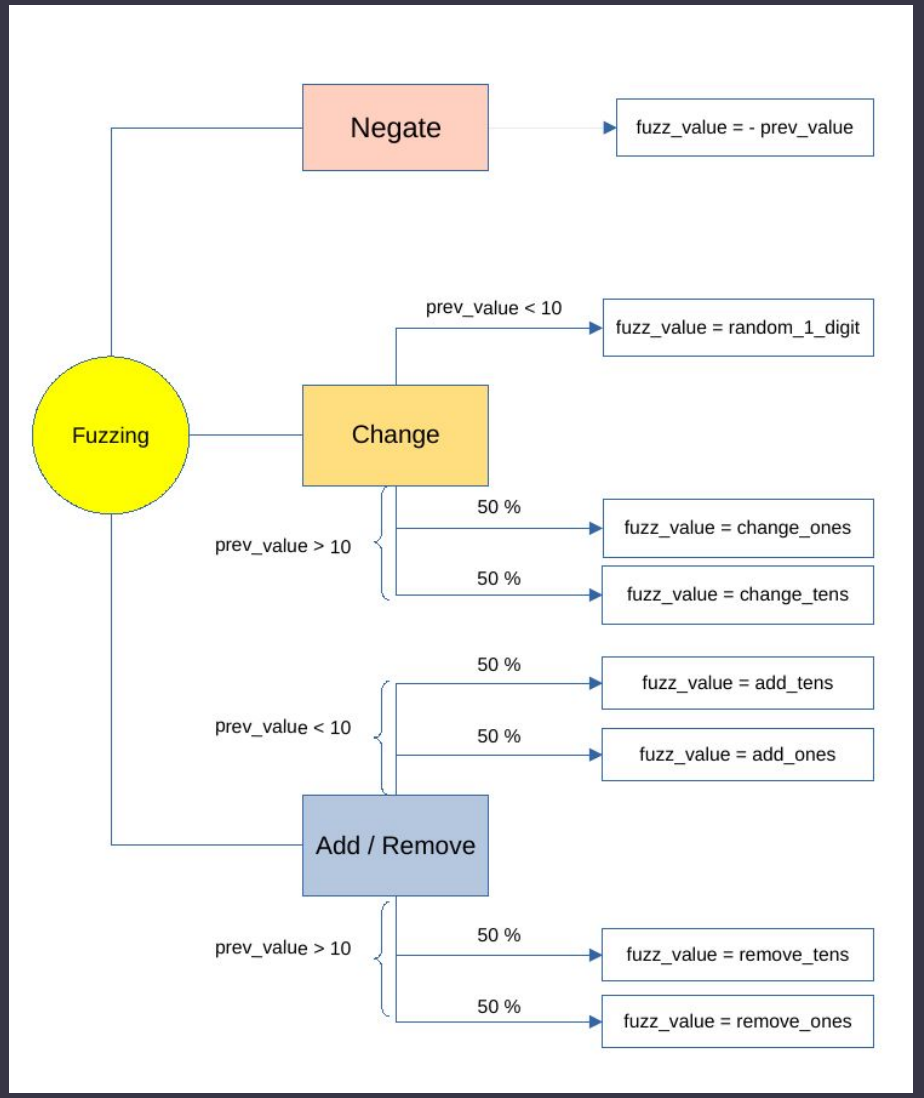
   ii. if a1 = 45: the next value can be 4 or 5

the choice between new values is stochastic.

```
else if (operation == "add_remove_digit") {
    if (std::abs(prev_value) < 10) { // single digit number
        ...
    }
    else { // two digit number
        ...
    }
}
```

At the end, the generated vector will be compared to previous generated vectors. If at least one input value is different in the new vector, fuzzing algorithms accepts the new values, otherwise, it will generate values.

Fuzzing (contd.)



Steps to build and run the code

1. Generating human-readable LLVMIR files from test files:

```
clang -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
clang++ -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
clang-10 -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
clang++-10 -emit-llvm -fno-discard-value-names -S -o test1.ll test1.c
```

2. Generate CFG by LLVM:

```
opt -dot-cfg test1.ll
```

3. Generate a PDF from CFG:

```
./allfigs2pdf
```

Steps to build and run the code (contd.)

4. Compiling the FuzzTesting LLVM pass:

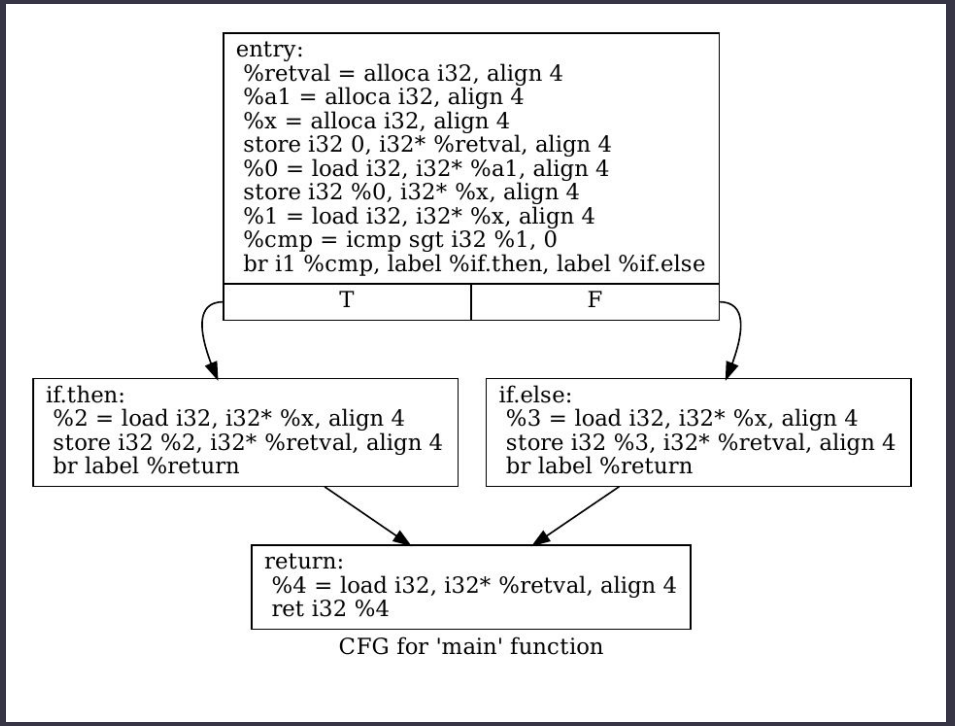
```
clang++-10 -o FuzzTesting FuzzTesting.cpp `llvm-config
--cxxflags` `llvm-config --ldflags` `llvmconfig --libs`
-lpthread -lncurses -ldl
```

5. Running LLVM pass on the LLVM IR files:

```
./FuzzTesting test1.ll
./FuzzTesting test2.ll
./FuzzTesting test3.ll
```

Examples (1)

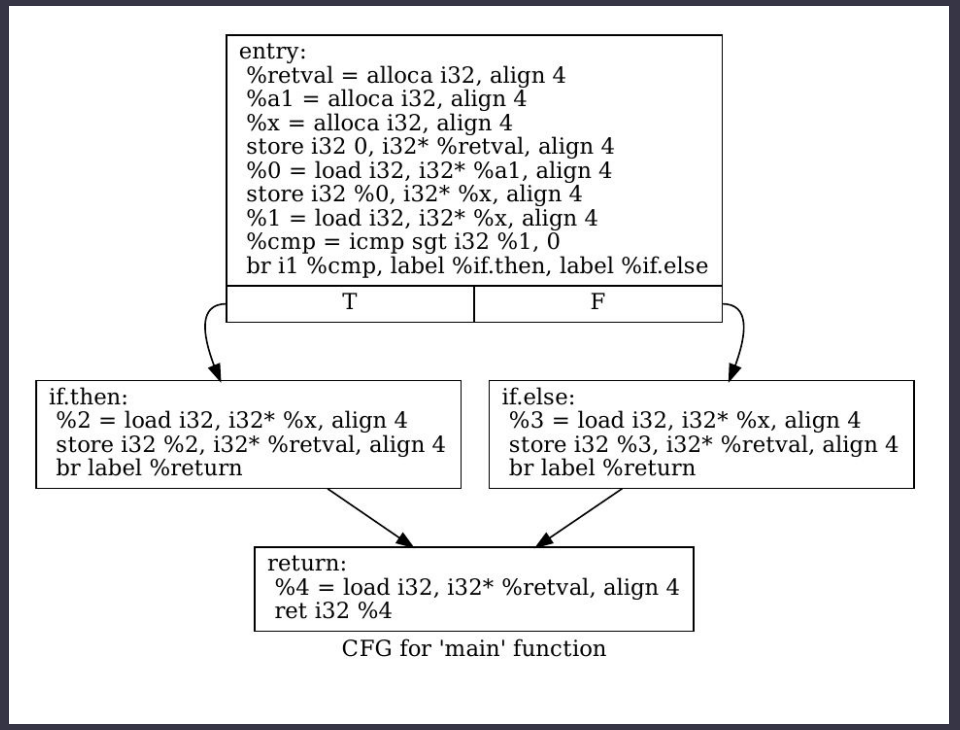
```
int main() {
    int a1;
    int x = a1;
    if (x > 0) {
        return x;
    }
    else {
        return x;
    }
}
```



CFG for 'main' function

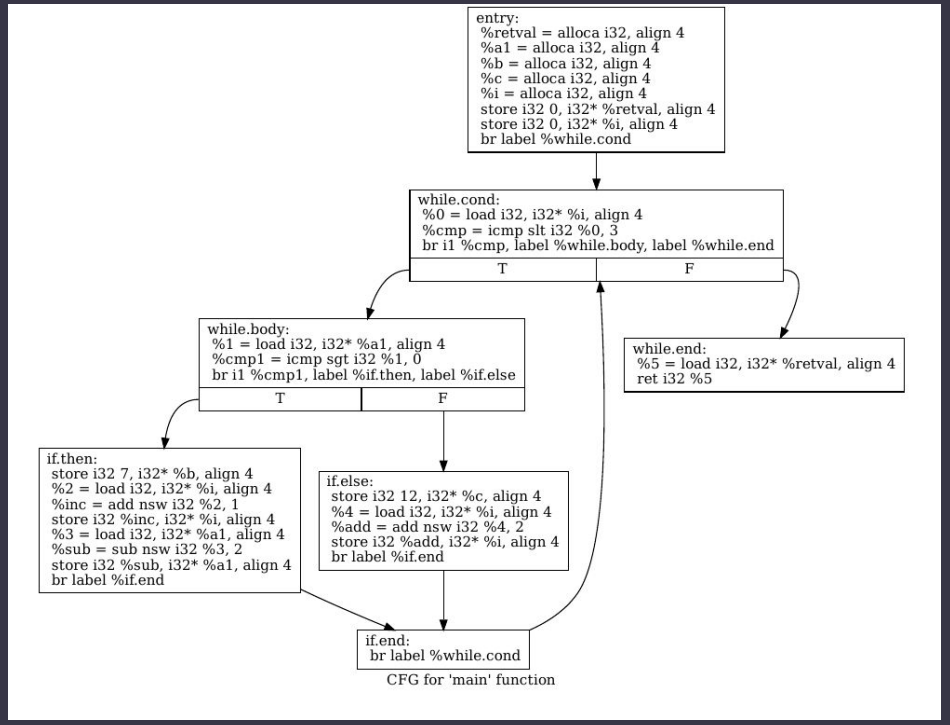
Examples (1)

1st Iteration:	2nd Iteration:	3rd Iteration	4th iteration:
a1 = -73 Sequence of Basic Blocks: entry if.else return block coverage: 75%	a1 = -7 Sequence of Basic Blocks: entry if.else return block coverage: 75%	a1 = -79 Sequence of Basic Blocks: entry if.else return block coverage: 75%	a1 = 79 Sequence of Basic Blocks: entry if.then return block coverage: 100%



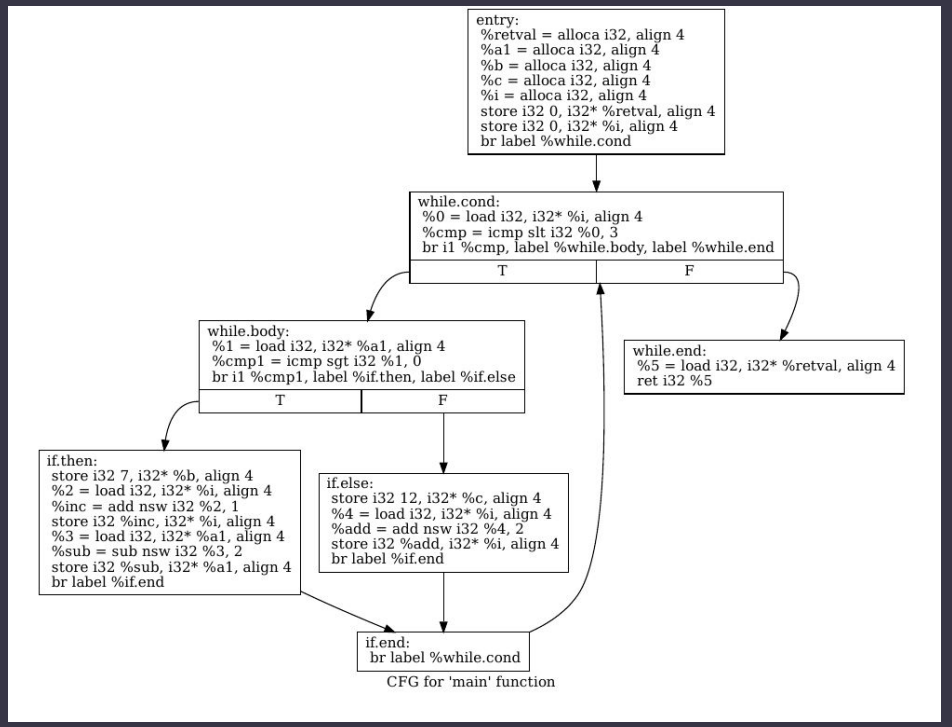
Examples (2)

```
int main() {
    int a1;
    int b,c;
    int i = 0;
    while (i < 3) {
        if (a1 > 0){
            b = 7;
            i++;
            a1 = a1 - 2;
        } else {
            c = 12;
            i=i+2;
        }
    }
}
```



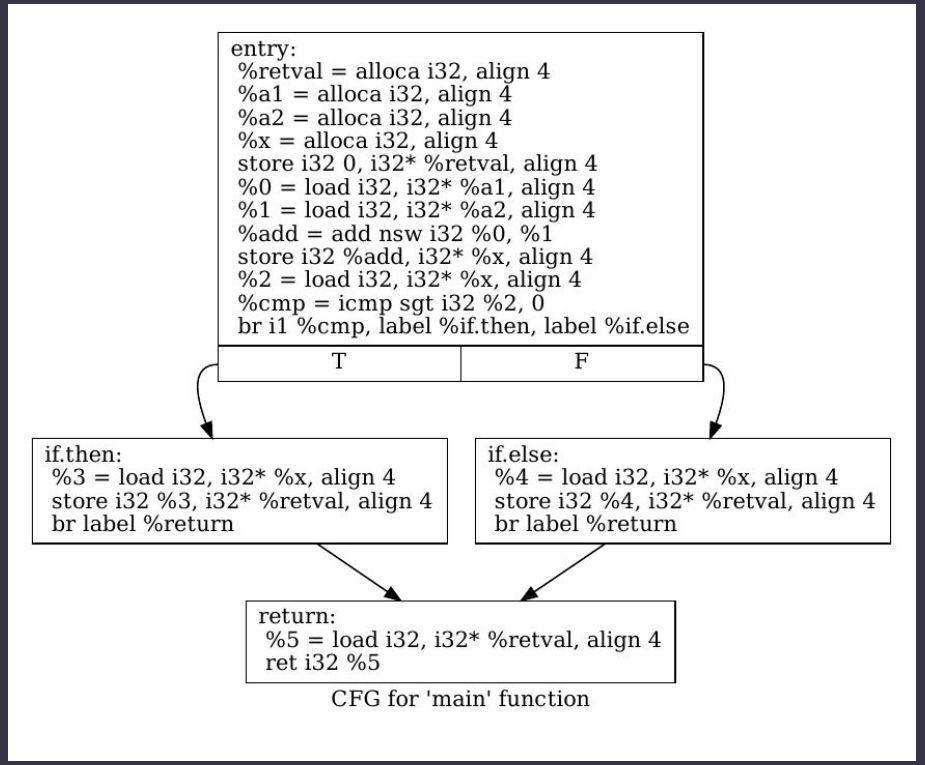
Examples (2)

1st Iteration:	2nd Iteration:	3rd Iteration
a1 = -67 Sequence of Basic Blocks: entry while.cond while.body if.else if.end while.cond while.body if.else if.end while.cond while.end block coverage: 85%	a1 = -7 Sequence of Basic Blocks: entry while.cond while.body if.else if.end while.cond while.body if.else if.end while.cond while.end block coverage: 85%	a1 = 7 Sequence of Basic Blocks: entry while.cond while.body if.then if.end while.cond while.body if.then if.end while.cond while.body if.then if.end while.cond while.end block coverage: 100%



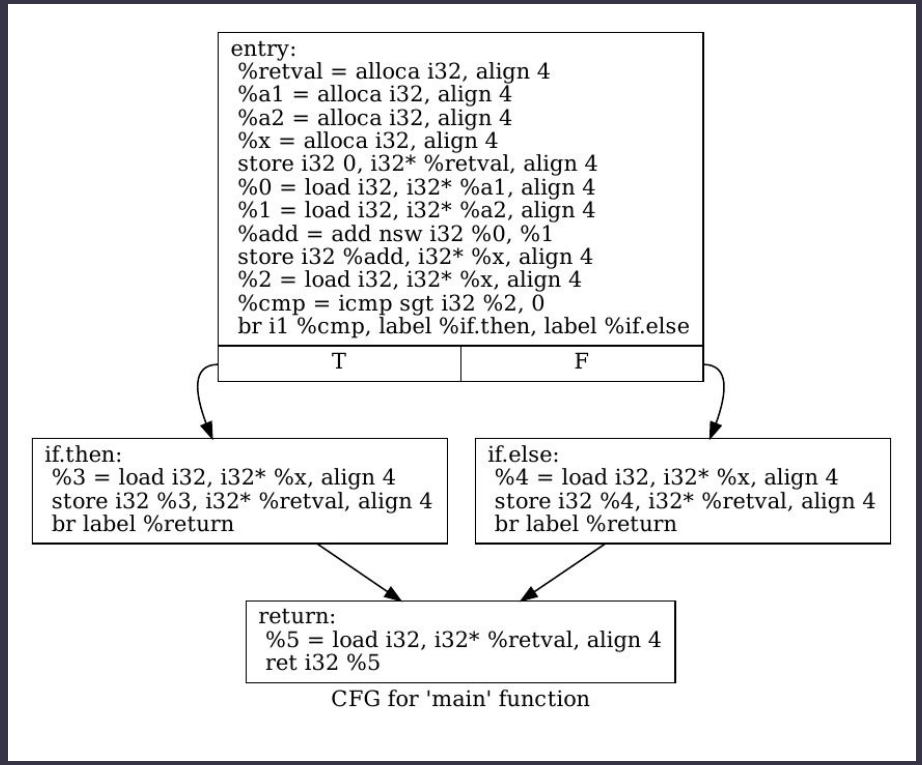
Examples (3)

```
int main() {
    int a1;
    int a2;
    int x = a1+a2;
    if (x > 0) {
        return x;
    }
    else {
        return x;
    }
}
```



Examples (3)

iteration 1:	iteration 2:	iteration 3:	iteration 4:	iteration 5:
a1 = 67	a1 = 66	a1 = 6	a1 = 46	a1 = -46
a2 = -19	a2 = -1	a2 = -3	a2 = -33	a2 = 33
Sequence of Basic Blocks:	Sequence of Basic Blocks:	Sequence of Basic Blocks:	Sequence of Basic Blocks:	Sequence of Basic Blocks:
entry	entry	entry	entry	entry
if.then	if.then	if.then	if.then	if.else
return	return	return	return	return
block coverage: 75%	block coverage: 75%	block coverage: 75%	block coverage: 75%	block coverage: 100%



Phase 3

Introduction

Goals

- Designing a simple dynamic symbolic execution tool
- Gaining exposure to LLVM in general and the LLVM IR which is the intermediate representation
- used by LLVM.
- Using LLVM to perform a sample testing

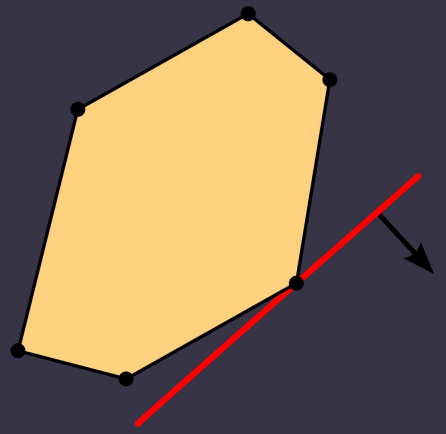
Constrains

- One or more int inputs. (the tester will generate values for these inputs)
- The inputs are defined at the beginning of the program in the form of ai
- Can contain other int variables
- Can contain if statements
- Can contain binary operations (add, sub, mul, div)
- Can contain loops (while, for, …)

Ideas & Methods

Linear Programming

- First Idea in case of same terminology (feasible, constraints, …)
- Works only if all the constraints are in linear form, or in form of $ax_1 + ax_2 + \ldots + ax_n$
- The equations can get solved with the linear programming solving algorithm -> Simplex
- Problem: there is no guarantee that all expressions are in linear form.



Constraint Satisfaction Problem (CSP)

- More General than linear programming problems. It can solve non-linear expressions too.
- Have to declare the value domain for each variable.
- It uses Backtracking algorithms with some heuristic functions:
    - Minimum Remaining Variable (MRV)
    - Degree Heuristic
    - Least Constraining Value (LCV)
- Forward Checking: Delete all inconsistent variables with current assignments.
- Problem: Stuck in simple constraint with large scale domain. (e.g., if x!=1,000,000)

Suggested Method

Interval Analysis Method

- First Checks for simple constraints, for example single variable inequality.
- Easy to calculate the expression and find the next feasible variable.
- Sample expressions: (x!=1000000, x==y, x!=y, y>0, y<0, …)
- Problem: Can’t apply to all types of expressions.

Goal: Interval Analysis + CSP

- The main idea is to do the following steps:
    - Generate random values for input variables in the first iteration.
    - Parse the code to find single variable constraint and easy to calculate ones.
    - Finding new values for input variables with top-down approach.
    - Use CSP problem solver for complicated expressions.