Null pointer checks in SafeC

August 19, 2019

1 Introduction

The goal of the safe-C project is to add memory and type safety in C. Towards this goal we disallow programmers to do pointer arithmetic, use the address of (&) operator, static array allocation, malloc and free. SafeC provides "mymalloc" API for memory management. "mymalloc" routine keeps track of the size and type information of the object for dynamic enforcement of memory safety.

2 Detection of null pointer dereference at runtime

In C-Safe world, a pointer may also contain a null value. A null pointer is useful for checking the terminating condition. E.g., the last node of a linked list may point to a null location. A null pointer is never meant to be dereferenced. However, it is valid to compare a pointer value with null. Let us look at the following example:

```
val = ptr[0];
ptr[0] = 100;
if (ptr == NULL) {
    ptr = mymalloc(4);
}
```

Here, "ptr == NULL" condition compares the value of ptr with null. By comparing the above values, the program is not trying to read from or write to the null location. On the other hand, "ptr[0] = 100" is writing to a memory location stored in ptr. If ptr is null at runtime, "ptr[0] = 100" tries to write to a null location. The statement "val = ptr[0]" is reading from a memory location stored in ptr. If ptr contains a null value, then the above statement tries to read from the null location. A program must handle reading or writing to a null location gracefully to ensure safety. If a user-defined handler corresponding to a null pointer dereference is not defined, the default handler should terminate

the program. It is not possible to eliminate all null pointer dereferences at compile-time because the value of a memory operand is available during the actual execution. However, a compiler can minimize the number of checks that are needed to track null pointer dereferences at runtime. For example, consider the following example:

```
ptr[0] = 1;
ptr[1] = 100;
```

One naive approach would be to add a check before every memory dereference. However, if a compiler can statically prove that store to ptr[0] always executes before ptr[1], then check before ptr[1] is not required because the program terminates before the first store if ptr is null. In this assignment, we will implement data flow analysis to identify instructions that can access a null location and add dynamic checks before them to terminate a program if the target memory location is null at runtime.

3 Data flow analysis [7 Marks]

In this assignment, you are to implement a data-flow analysis to identify memory operands that are statically proven to be not null. Let us call these memory operands safe operands. The data flow analysis tracks all the arguments and local variables that are pointers. Because a function can be called from multiple places, and it is valid to pass a null value as an argument, the pointer arguments are treated as unsafe. Similarly, the return value of a function is always unsafe. However, there is one exception to the above rule. A pointer can be initialized using "mymalloc" routine. You can assume that "mymalloc" always returns a valid memory location.

The local variables and arguments are stored on the stack. LLVM uses alloca instruction to allocate space on the stack. It is always safe to load/store from stack-allocated memory operand. When the analysis pass encounters a memory dereference with an unsafe memory operand, it selects the corresponding instruction for the dynamic check. After insertion of the dynamic check, an unsafe operand becomes safe in that basic block because the first check ensures the termination of the program if the operand value is null during execution. Similarly, when a safe operand is getting stored in a stack location, the corresponding stack location is also marked as safe. Future, load from a safe stack location in that basic block yields a safe value.

Your goal is to implement an LLVM pass to identify instructions that are unsafe and need dynamic checks. For this, you need to define transfer functions for every LLVM instruction and a meet operator to merge results from predecessors.

```
int max(int *arr1, int *arr2) {
  if (arr1[0] > arr2[0]) {
    return arr1[0];
  }
  return arr2[0];
}
```

(a) Without checks

```
int max(int *arr1, int *arr2) {
  if (arr1 == NULL) {
    goto out;
  }
  if (arr2 == NULL) {
    goto out;
  }
  if (arr1[0] > arr2[0]) {
    return arr1[0];
  }
  return arr2[0];
out:
  abort();
  return 0;
}
```

(b) With checks

Figure 1: Dynamic checks for null detection

4 Dynamic checks [6 Marks]

After you identify the instructions that needed dynamic checks, your next goal is to insert the actual checks before them. The dynamic check compares the value of the memory operand with null and calls abort if the memory routine is null. For example, Figure 1 shows a transformation with dynamic checks.

5 Environment

For this assignment, you need to clone the project repository.
To clone, run, git clone `https://github.com/Systems-IIITD/CSE601`.
If you have already cloned, sync your local repository by running `git pull origin master`. To build the project, follow the steps in the `README.md` file. You have to make all the changes in the ‘‘`llvm/lib/CodeGen/SafeC/NullChecks.cpp`’’ file. The name of this pass is `nullcheck`. You can use `opt` tool to run `nullcheck` pass.

6 Test cases

‘‘`tests/PA1`’’ folder contains few test cases. You are encouraged to add more test cases. You can earn extra credit up to three marks if your test case covers a corner case that is not already covered by the existing test cases. The expected dynamic checks insertion points are given in the test cases. Run “make” in the “tests/PA1” folder to run the test cases. The makefile uses `llvm-dis` tool to print the `LLVM IR` in a file. `nullcheck(*)_opt.ll` files contain the `LLVM IR`

after the nullcheck pass. nullcheck(*).ll files contain the original `LLVM IR` before the nullcheck pass.

7 Tools

You can use cscope, ctags, and vim to navigate the source code. Sublime text 3 also works well with LLVM.

8 Other resources

You can refer to "`https://llvm.org/doxygen/`" for quick reference to the `classes` in LLVM. E.g., to search all the public functions in the LLVM `Function` class, type "llvm Function" in the google search bar for a doxygen page related to the Function class in LLVM.

9 Other LLVM details

You can refer to "`http://llvm.org/docs/WritingAnLLVMPass.html`" to read about an LLVM pass. A function in LLVM is represented using ``` ``class Function'' ```. In an LLVM function pass, `runOnFunction` routine is called for every function with a handle to ``` ``class Function'' ``` of that routine.

```
To iterate all the basic blocks in a function F,
for (BasicBlock &BB : F)

To iterate all the instructions in a basic block BB,
for (Instruction &I : BB)

To check if an instruction is Alloca, Call, Load, Store, GetElementPtr or Cast:

AllocaInst *AI = dyn_cast<AllocaInst>(&I); // where I is of type Instruction
if (AI != NULL) {
  // This means that the instruction is Alloca instruction.
}
CallInst *CI = dyn_cast<CallInst>(&I);
LoadInst *LI = dyn_cast<LoadInst>(&I);
StoreInst *SI = dyn_cast<StoreInst>(&I);
GetElementPtrInst *GEP = dyn_cast<GetElementPtrInst>(&I);
CastInst *CAI = dyn_cast<CastInst>(&I);

To find the allocation type of a AllocaInst(AI):
Type *Ty = AI->getAllocationType();

To check if a Type(Ty) is pointer:
Ty->isPointerTy()

To check if a CallInst(CI) is indirect:
```

```
CI->isIndirectCall()

To get the name of the called function (in case of direct call):
CI->getCalledFunction()->getName();

To get the pointer operand of LoadInst/StoreInst/GetElementPtrInst:
Value *PtrOp = (LI/SI/GEP)->getPointerOperand();

To get the value of a StoreInst:
Value *Val = SI->getValueOperand();

To check if a value is Constant:
isa<Constant>(Val)

To check is a constant value is Null:
dyn_cast<Constant>(Val)->isNullValue()

To add a new basic block (with name newBB) in function (F):
BasicBlock::Create(F.getContext(), "newBB", &F)

To get a handle of the given function foo:
getOrInsertFunction("foo", ...);

To add new instructions at the end of a basic block.
IRBuilder<> IRB(BB);
IRB.CreateCall(...);
IRB.CreateBr(...);
The above statements will append a call and branch instruction to the basic block BB.

To split a basic block at Instruction I:
SplitBlock(BB, I);

To get the next instruction of instruction I:
I->getNextNode();

To add a conditional branch to a basic block BB:
IRBuilder<> IRB(BB);
auto Cmp = IRB.CreateICmp(...,..., ...);
IRB.CreateCondBr(Cmp, ..., ...);


To add a return statement:
IRBuilder<> IRB(BB);
IRB1.CreateRetVoid(); // return value is void
IRB1.CreateRet(...);  // takes return value as an argument

To create a input map for all basic blocks in a function.
Study std::map in c++.
std::map<BasicBlock*, std::map<Value*, bool>>
```

```
To walk all the successors of basic block BB:
for (BasicBlock *SuccBB : successors(BB))

To walk all the predecessors of basic block BB:
for (BasicBlock *PredBB : predecessors(BB))

To implement DFS or BFS you can use std::vector or std::deque.
```

9.1 How to submit.

Submit the "nullcheck.cpp" file at the submission link on Backpack. If you want to suggest a test case, please send a pull request to the project repository.