Type and spatial safety in SafeC

October 27, 2021

1 Introduction

This exercise aims to enforce *spatial* and a weaker *type* safety for the C language. Our tool is called `SafeC`. `SafeC` provides "`mymalloc`" and "`myfree`" API for memory management. "`mymalloc`" routine keeps track of the size and type information of the object for dynamic enforcement of memory safety. In this assignment, we will ensure spatial safety and verify the validity of pointer fields after a memory write.

2 alloca to mymalloc [6 marks]

Whenever a stack address or something derived from a stack address is passed to a routine or stored in memory, convert the corresponding `alloca` instruction to `mymalloc`. Also, insert `myfree` API to free the memory whenever the `alloca` that was transformed in the previous step goes out of scope.

Consider the following example.

```
void bar() {
1. int arr[5];
2. int *a = arr;
3. a++;
4. foo(a, 3);
}
```

In this example, the parameter `a` at line-4 is a stack address. Transform this code as follows.

```
void bar() {
  int *arr = mymalloc(20);
  int *a = arr;
  a++;
  foo(a, 3);
  myfree(arr);
}
```

Similarly, if a stack address is stored in a memory location, convert the corresponding stack allocation to mymalloc and manually reclaim the memory using myfree. You can ignore the stack addresses passed to the library routines during this analysis.

3 Disallowing out-of-bounds pointers [4 marks]

At runtime, if an out-of-bounds pointer is passed to a function, or returned from a function, or stored in memory; abort the program. Consider the following example.

```
1. void foo() {
2.   int *arr = mymalloc(20);
3.   f1(&arr[48], 3);
4.   int **ptr = malloc(8);
5.   *ptr = &arr[48];
6.   f2(ptr, 3);
7.   return &arr[48];
8. }

10. void f1(int *arr, int offset) {
11.    arr[offset] = offset;
12. }
```

In this example, an out-of-bounds pointer (&arr[48]) is being passed to f1 (at line-3), stored in memory (at line-5), and returned from function (at line-7). After these events, the out-of-bound pointer may be visible to other parts of the code. For example, memory access at line-11 is an out-of-bounds access. It is possible that the out-of-bounds pointer is actually pointing to a valid object (i.e., other than the object allocated at line-2). However, at line-10, we can't infer that the argument arr was actually derived from the object allocated at line-2. To prevent this, you need to add dynamic checks to abort the program in all these cases.

To compute the bounds for &arr[48], you first need to compute the base arr. To compute, arr, starting from &arr[48], you can recursively backtrack all the getelementptr and bitcast operations until you find a definition that is not a getelementptr or bitcast instruction. Once you statically identify the base, you can compute the real base from the statically inferred base using the SafeGC API you implemented for the previous assignment. You can ignore the out-of-bounds pointers passed to the library routines during this analysis.

4 Adding bounds check [2 marks]

Before every memory access, insert a dynamic check to abort the program if the memory access is not within the bounds. For example, consider the following case.

```
void bar(int *arr, int idx) {
  arr[idx] = 0;
}
```

In this example, the memory access size is 4-bytes, and the statically inferred base corresponding to the memory access is `arr`. You can find the actual base from the statically inferred base using the SafeGC API. The upper bound of the object can be computed after adding the size of the object (obtained from the object header) to the base address. For stack and global objects, you can compute the size from the statically available information. The bounds checking logic aborts the program if [arr, arr + access_size - 1] is not within the bounds of the object.

5 Allocator

We are using the `SafeGC` allocator for `SafeC`. `mymalloc` routine inserts an object header before every object, which contains the size and type information of the object. `mymalloc` always returns an object of type `i8*`. The `typeassigner` pass inserts code after `mymalloc` to store the type of the allocated object in the object header. `SafeC` computes a bitmap to represent the type of an object. If the type doesn't contain a pointer field, then the bitmap is set to zero; otherwise, the bitmap is computed as follows.

`SafeC` divides the type into chunks of eight-byte fields starting from the top. Every field has a corresponding bit in the bitmap. The bit position of the first field in the bitmap is zero; the second field is one, and so on. A set bit in the bitmap represents a pointer, and a bit value zero represents a non-pointer. For types with pointer fields, the `nth` bit in the bitmap is set to one (where `n` is the number of eight-byte fields) to identify the size of the type at runtime. `SafeC` does not support types (with pointer fields) of size more than `‘‘63 * 8’’` bytes. This scheme works because, by default, `LLVM` generates eight-byte aligned offsets for pointer fields in a composite data structure. However, the application can use type attributes to override the default behavior. `SafeC` only supports applications that satisfy the above constraint.

```
struct A {
  unsigned long long a;
  unsigned long long *b;
  unsigned long long c;
  unsigned long long *d;
  unsigned long long e;
};
```

For example, the bitmap corresponding to `struct A` is `101010 (0x2a)`. You can refer to the `computeBitMap` routine in `TypeAssigner.cpp` for the bitmap computation. `TypeAssigner.cpp` inserts `mycast` calls after the object allocation that stores the type in the object header of the allocated object.

6 Write barriers [3 marks]

After every write to an object, you need to insert a write barrier. Write barrier checks if any pointer field was updated due to the write. If yes, then it asserts that the updated value is either `NULL` or points to a valid object. The write barrier aborts the program if the above assertion fails. You can identify whether a field is a pointer or not using the type information stored in the object header. For a stack-allocated object, you can obtain the type from the corresponding `alloca` instruction.

Consider the following example,

```
struct A {
   int *fld1;
   unsigned long long fld2;
   int *fld3;
   unsigned long long fld4;
};

void foo(int offset) {
1.  struct A *a = (struct A*)mymalloc(sizeof(struct A));
2.  a->fld1 = mymalloc(4);
3.  a->fld3 = mymalloc(4);
4.  char *ptr = (char*)a;
5.  a[2] = 0;
6.  a[8] = 0;
}
```

In this example, the write at line-5 updates `fld1` (a pointer field) of the object allocated at line-1. Therefore, the write barrier checks whether `a->fld1` still points to a valid object or contains a `NULL` value after the update. However, the write at line-6 updates a non-pointer filed (`fld2`) of the object, so the write barrier doesn't check anything for this update. Notice, a single write can partially update multiple fields if the starting address of the write is some internal address of a field.

7 Environment

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 implement `LLVM` specific code in the `‘‘llvm/lib/CodeGen/SafeC/MemSafe.cpp’’` file. The other routines that are called by your instrumented code need to be implemented in the `‘‘support/SafeGC/support.c’’` file.

8 Test cases

`‘‘tests/PA4’’` folder contains some test cases. Run “make” in the “tests/PA4” folder to compile the test cases. You can run a test case by manually running the generated executable. You can find sample test inputs in the makefile. The makefile uses `llvm-dis` tool to print the `LLVM IR` in a file. Please feel free to add more test cases to test your implementation.

9 Tools

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

10 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`.

11 Other LLVM details

By this time, you might have already explored several APIs in `LLVM`. The best way to start this assignment is to go through `llvm/lib/CodeGen/SafeC/TypeAssigner.cpp`. You can reuse some of the code from there in your implementation. `TypeAssigner.cpp` also inserts calls to `mycast`; you can insert routines that implement runtime checks in a similar way. You can use `isLibraryCall` implementation in the `‘‘llvm/lib/CodeGen/SafeC/MemSafe.cpp’’` file to check if a call instruction is actually a library call. Please feel free to post your query on the classroom page if you need any help with the `LLVM` APIs.

11.1 How to submit

Create a zip folder that contains `MemSafe.cpp` and `support.c`. Upload the zip folder to the submission link.