Type and spatial safety in SafeC

October 22, 2019

1 Introduction

The goal of the `SafeC` 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, array within structures, 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. In this assignment, we will ensure size invariant, type invariant, and spatial safety in `SafeC`.

2 Size invariant [3 marks]

The size invariant ensures that a pointer is either null or points to a memory area that is big enough to store at least one instance of the pointer's base type. In other words, if a pointer variable `ptr` of type `struct List*` is not null, then it points to a memory area that is greater than or equal to `sizeof(struct List)`. The size of the memory area, referred by a pointer, needs not to be an exact multiple of the base type of the pointer. `SafeC` already checks the above invariant during the object creation (Section 5). However, this invariant may be violated if the program typecasts an object, with a bigger type.

```
struct A {
 unsigned long *a;
};

struct B {
 unsigned long *a;
 unsigned long *b;
};
```

E.g., if we typecast an object `obj` of type `struct A*` to `struct B*`, we can not guarantee size invariant. Because the only thing we know for sure about obj that either it is null, or it is pointing to a memory area of size greater than or equal to sizeof(struct A). To ensure the size invariant, we need to insert the routine checkSizeInv (see support.c) before such typecasts to check size invariant at runtime. On the other hand, typecasting of struct B* to struct A* is safe. Because, if the size invariant holds for struct B* then it must also be true for struct A*.

3 Type invariant [6 marks]

The type invariant ensures that a pointer field in an object is never interpreted as non-pointer and vice versa. The allocator (Section 5) stores the object type in the object header during allocation. SafeC rejects a program when it can statically disprove the type invariant from the known facts (assuming size invariant) about the casted object. SafeC inserts a runtime check before the typecast when it is unable to prove or disprove type invariant statically. We discuss below some of the examples and the expected behavior of SafeC.

```
struct A {
 char *a;
};

struct B {
char a;
};
```

In the example above, SafeC doesn't allow casting of an object of type struct A* to struct B* because allowing that would enable programmers to interpret a pointer to char as char. In this case, SafeC aborts the compilation because it can disprove type invariant statically.

```
struct A {
 char *a;
};

struct B {
char *a;
int b;
};
```

In the example above, SafeC doesn't allow casting of an object obj of type struct A* to struct B* for the following reason. The size invariant requires obj to point to a memory area of size at least sizeof(struct B). Assuming this fact, SafeC can infer that obj is pointing to an array of at least two elements of type struct A. If typecasting of obj to struct B* is permitted, then the program can interpret the second element of the array as an integer (using the ``b'' field of struct B) that was initially a pointer (``a'' field of struct A). Therefore, SafeC rejects this program.

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

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

In this example, let us say that program is trying to typecast an object of type `struct A*` to type `struct B*`. Here, `SafeC` cannot disprove type invariant for all possible sizes. E.g., if the object size is 32, both `struct A` and `struct B` satisfy size invariant, and none of the pointer/non-pointer fields can be interpreted as non-pointer/pointer. However, if the object size is 40, then the type invariant cannot be guaranteed, although the size invariant holds. `SafeC` also needs to check the type invariant at runtime, because, as of now, it can only say that the object size is at least 24 bytes (i.e, `sizeof(struct A)`). Size invariant requires object size to be at least 32 bytes. In this case, `SafeC` inserts `checkSizeAndTypeInv` (see `support.c`) routine before the typecast to check both size and type invariant.

On the other hand, if the program tries to typecast an object of type `struct B*` to `struct A*`, then only type invariant check is required. To check the type invariant at runtime `SafeC` inserts `checkTypeInv` (see `support.c`) before the typecast.

```
struct A {
 unsigned long long  a;
 unsigned long long* b;
};

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

In this case, the compiler can prove that typecasting of `struct A*` to `struct B*` is safe for all possible sizes. To prove the type invariant for all possible sizes, and any two types (say `type A` and `type B`), we have to prove that the type invariant holds if the object size is “least common multiple” of `sizeof(type A)` and `sizeof(type B)`. Typecasting of `struct A*` to `struct B*`, only requires a dynamic check for size invariant (`checkSizeInv`). Typecasting from `struct B*` to `struct A*` does not require any check.

4 Spatial safety [4 marks]

Thanks to size invariant, `SafeC` doesn't need to check the bounds while accessing the first element of an array. For other cases, we need to insert dynamic checks to ensure that the memory access is within the array bounds. Our allocator stores the object size in the object header (Section 5). On encountering an out of bounds array access, `SafeC` aborts the program. You have to inline the bounds checks in the function IR. You are not supposed to call a routine at runtime (similar to `checkSizeInv`) to do the checking. Before accessing an array element, `LLVM` first computes the address of target element (using getelementptr) and then access the element using the calculated address.

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

struct A *obj = (struct A*)mymalloc(sizeof(struct A) * 4);
obj[2].c = NULL;

%struct.A = type { i64, i64, i64* }

/* first compute %arrayidx = &obj[2] */
%arrayidx = getelementptr inbounds %struct.A, %struct.A* %3, i64 2
/* then compute %c = &%arrayidx->c */
%c = getelementptr inbounds %struct.A, %struct.A* %arrayidx, i32 0, i32 2
/* access %c */
```

However, we need to be careful with `getelementptr`. In the example above, instead of calculating &obj[2].c directly, `LLVM` first computes `%arrayidx = &obj[2]` and then it computes `%c = &%arrayidx->c`. In this case, you should obtain the base address of the object from the first `getelementptr` and the target address (which is eventually going to be accessed) from the second `getelementptr`.

5 Allocator

We are using `SafeGC` allocator for `SafeC`. `mymalloc` routine inserts an object header before every object, that contains the size and type information of the object. `mymalloc` always returns an object of type `i8*`. Because we disallow unsafe typecasts in `SafeC`, we call `mycast` routine in `SafeGC` library to do the unsafe typecast. `SafeGC` is not compiled using `SafeC` compiler; therefore, it can do arbitrary typecasts. In addition to returning the correct type of the object, `mycast` also sets the type field in the object header. `SafeC` computes a bitmap corresponding to the type of dynamically allocated objects. 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 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.

6 Environment

Sync your local `SafeC` repository by running `git pull origin master`. To build the project, follow the steps in the `README.md` file. You have to implement size and type invariant checker in the `''llvm/lib/CodeGen/SafeC/TypeChecker.cpp''` file. The spatial safety checks need to be implemented in the `''llvm/lib/CodeGen/SafeC/ArrayChecks.cpp''`. You also need to implement runtime check, `checkTypeInv`, in the `''support/SafeGC/support.c''` file.

7 Test cases

`''tests/PA3''` folder contains few test cases. Run "make" in the "tests/PA3" folder to compile the test cases. You can run a test case by manually running the generated executable. The makefile uses `llvm-dis` tool to print the `LLVM IR` in a file.

8 Report

You have to submit a report that contains the following details.

- Discuss your spatial safety check implementation.
- Discuss your implementation of `checkTypeInv`.
- For each test case, report the compilation status, what runtime checks were inserted, and runtime behavior of the test case.

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 a lot of code from there in your implementation. `TypeAssigner.cpp` inserts call to `mycast`; you can insert runtime checks for type and size invariant in a similar way. You can reuse some of your implementation from the `NullChecks.cpp` to implement array bounds checks.

11.1 How to submit

Create a zip folder that contains `ArrayChecks.cpp`, `TypeChecker.cpp`, `support.c`, and your report. Upload the zip folder to the submission link at `Backpack`.