Clang Static Analyzer: A Tryst with Smart Pointers

Deep Majumder

Indian Institute of Technology, Kharagpur

LLVM Developers' Meeting

November 17, 2021

GSoC 2021

The talk revolves around my Google Summer of Code project this year under The LLVM Foundation. The project was Making Smart Pointer Checkers default checkers in the Static Analyzer.

GSoC 2021

The talk revolves around my Google Summer of Code project this year under The LLVM Foundation. The project was Making Smart Pointer Checkers default checkers in the Static Analyzer.

My mentors were:

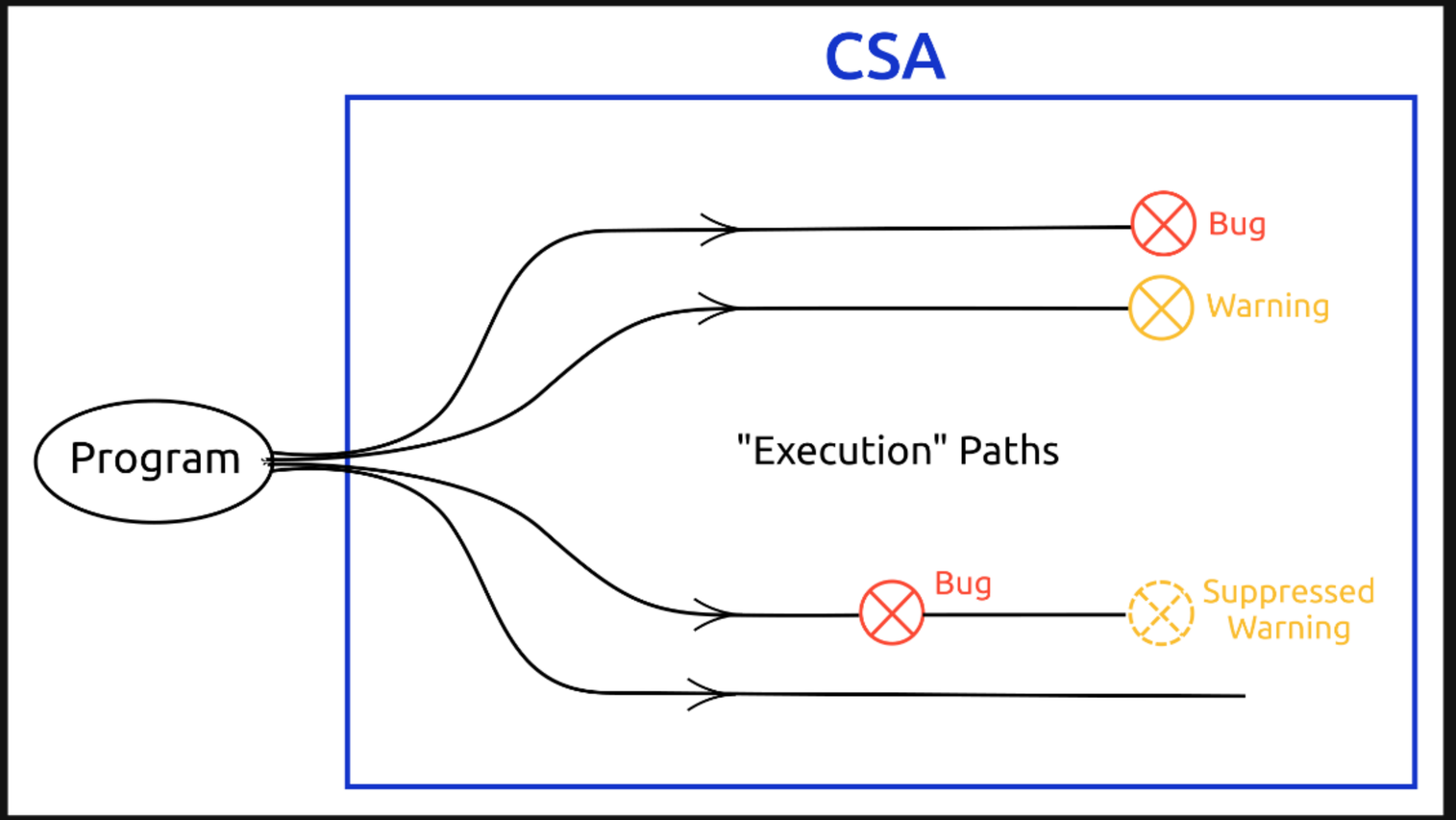
- Artem Dergachev (Apple Inc.)
- Gábor Horváth (Microsoft Corp.)
- Valeriy Savchenko (Apple Inc.)
- Raphael Isemann (Vrije Universiteit Amsterdam)

What is the Clang Static Analyzer?

The Clang Static Analyzer (CSA) is a static-analysis framework that uses **symbolic execution** over all possible paths in a program to flag various buggy scenarios.

What is the Clang Static Analyzer?

The Clang Static Analyzer (CSA) is a static-analysis framework that uses **symbolic execution** over all possible paths in a program to flag various buggy scenarios.



The CSA tries to have no (low) false positives. This means:

- A bug reported by the CSA is definitely a bug.
- Absence of bug reports by the CSA does not indicate absence of bugs.

This is different from the model followed by some other static analysis systems, such as the one in Rust.

It's Checkers all the way down!

Checkers are the backbone of the CSA.

It's Checkers all the way down!

Checkers are the backbone of the CSA.

Checkers can:

- Create data
- Modify existing data
- Use the data to detect bugs

It's Checkers all the way down!

Checkers are the backbone of the CSA.

Checkers can:

- Create data
- Modify existing data
- Use the data to detect bugs

Roughly speaking, data stands for constraints on the possible values of a variable (or expression) at some point in the program. Also, we can store metadata to help dealing with complex scenarios.

For dealing with smart-pointers, we have currently have two checkers:

1. `SmartPtrModelling`, which actually creates the requisite data
2. `SmartPtrChecker`, which uses the data to detect null-dereferences

This de-coupling allows us to have multiple checkers to model the various smart-pointers, while having a *single checker emit the bug reports*.

How is it modelled?

`SmartPtrModelling` models the behaviour of `std::unique_ptr` by keeping track of the *nullity of the raw-pointer inside the smart-pointer*.

```
void foo() {
    unique_ptr<int> ptr = make_unique<int>(13);
    // ptr definitely is not null now.
    cout << *ptr << "\n";
    // This statement is safe.
    ptr.reset();
    // ptr now is definitely null.
    magicFunc(*ptr);
    // This statement is definitely a
    // null-ptr dereference.
}
```

```
void bar(unique_ptr<int> ptr) {
    // At this point we know nothing about ptr
    *ptr;
    // Since we aren't sure, we emit no report here
    if (!ptr) {
        // We are now sure that ptr is null here
        magicFunc(*ptr);
        // This statement is definitely a
        // null-ptr dereference.
    }
}
```

How is it modelled?

`SmartPtrModelling` models the behaviour of `std::unique_ptr` by keeping track of the *nullity of the raw-pointer inside the smart-pointer*.

```
void foo() {
    unique_ptr<int> ptr = make_unique<int>(13);
    // ptr definitely is not null now.
    cout << *ptr << "\n";
    // This statement is safe.
    ptr.reset();
    // ptr now is definitely null.
    magicFunc(*ptr);
    // This statement is definitely a
    // null-ptr dereference.
}
```

```
void bar(unique_ptr<int> ptr) {
    // At this point we know nothing about ptr
    *ptr;
    // Since we aren't sure, we emit no report here
    if (!ptr) {
        // We are now sure that ptr is null here
        magicFunc(*ptr);
        // This statement is definitely a
        // null-ptr dereference.
    }
}
```

How is it modelled?

`SmartPtrModelling` models the behaviour of `std::unique_ptr` by keeping track of the *nullity of the raw-pointer inside the smart-pointer*.

```
void foo() {
    unique_ptr<int> ptr = make_unique<int>(13);
    // ptr definitely is not null now.
    cout << *ptr << "\n";
    // This statement is safe.
    ptr.reset();
    // ptr now is definitely null.
    magicFunc(*ptr);
    // This statement is definitely a
    // null-ptr dereference.
}
```

```
void bar(unique_ptr<int> ptr) {
    // At this point we know nothing about ptr
    *ptr;
    // Since we aren't sure, we emit no report here
    if (!ptr) {
        // We are now sure that ptr is null here
        magicFunc(*ptr);
        // This statement is definitely a
        // null-ptr dereference.
    }
}
```

How is it modelled?

`SmartPtrModelling` models the behaviour of `std::unique_ptr` by keeping track of the *nullity of the raw-pointer inside the smart-pointer*.

```
void foo() {
    unique_ptr<int> ptr = make_unique<int>(13);
    // ptr definitely is not null now.
    cout << *ptr << "\n";
    // This statement is safe.
    ptr.reset();
    // ptr now is definitely null.
    magicFunc(*ptr);
    // This statement is definitely a
    // null-ptr dereference.
}
```

```
void bar(unique_ptr<int> ptr) {
    // At this point we know nothing about ptr
    *ptr;
    // Since we aren't sure, we emit no report here
    if (!ptr) {
        // We are now sure that ptr is null here
        magicFunc(*ptr);
        // This statement is definitely a
        // null-ptr dereference.
    }
}
```

How is it modelled?

`SmartPtrModelling` models the behaviour of `std::unique_ptr` by keeping track of the *nullity of the raw-pointer inside the smart-pointer*.

```
void foo() {
    unique_ptr<int> ptr = make_unique<int>(13);
    // ptr definitely is not null now.
    cout << *ptr << "\n";
    // This statement is safe.
    ptr.reset();
    // ptr now is definitely null.
    magicFunc(*ptr);
    // This statement is definitely a
    // null-ptr dereference.
}
```

```
void bar(unique_ptr<int> ptr) {
    // At this point we know nothing about ptr
    *ptr;
    // Since we aren't sure, we emit no report here
    if (!ptr) {
        // We are now sure that ptr is null here
        magicFunc(*ptr);
        // This statement is definitely a
        // null-ptr dereference.
    }
}
```

How is it modelled?

`SmartPtrModelling` models the behaviour of `std::unique_ptr` by keeping track of the *nullity of the raw-pointer inside the smart-pointer*.

```
void foo() {
    unique_ptr<int> ptr = make_unique<int>(13);
    // ptr definitely is not null now.
    cout << *ptr << "\n";
    // This statement is safe.
    ptr.reset();
    // ptr now is definitely null.
    magicFunc(*ptr);
    // This statement is definitely a
    // null-ptr dereference.
}
```

```
void bar(unique_ptr<int> ptr) {
    // At this point we know nothing about ptr
    *ptr;
    // Since we aren't sure, we emit no report here
    if (!ptr) {
        // We are now sure that ptr is null here
        magicFunc(*ptr);
        // This statement is definitely a
        // null-ptr dereference.
    }
}
```

What is there to model?

Everything!

What is there to model?

Everything!

When we are modelling a C++ class like `` `std::unique_ptr` ``, we need to model all aspects of it. This includes modelling at least all member functions, constructors, and destructors.

What is there to model?

Everything!

When we are modelling a C++ class like `` `std::unique_ptr` ``, we need to model all aspects of it. This includes modelling at least all member functions, constructors, and destructors.

The aim of modelling a class in a checker is to add information which is specific to the semantics of the class at hand.

What is there to model?

Everything!

When we are modelling a C++ class like `` `std::unique_ptr` ``, we need to model all aspects of it. This includes modelling at least all member functions, constructors, and destructors.

The aim of modelling a class in a checker is to add information which is specific to the semantics of the class at hand.

This information is not necessarily discoverable automatically because the source code may not be available. After all, the interface to a C++ library is via the header file, which may not contain all the code.

The Perils of Incomplete Modelling

Loss of information: Suppose we don't model the `reset()` method for `std::unique_ptr`, and assume it's source code is not available.

CSA will default to conservative evaluation, which basically removes information which may have been modified by the modelled method.

The Perils of Incomplete Modelling

Loss of information: Suppose we don't model the `reset()` method for `std::unique_ptr`, and assume it's source code is not available.

CSA will default to conservative evaluation, which basically removes information which may have been modified by the modelled method.

Then, the following code will report no errors, when there is obviously one (a *false-negative*).

```
void foo() {
    unique_ptr<string> strPtr = make_unique<string>("Oh no!");
    strPtr.reset();
    auto len = strPtr->size(); // This is null-pointer dereference!
    // But we have no warning!
}
```

Inconsistent information: Suppose we don't model the `destructor` of `std::unique_ptr`.

Since the code for the destructor is usually available, CSA will automatically evaluate it, in a process known as inlining.

Inconsistent information: Suppose we don't model the `destructor` of `std::unique_ptr`.

Since the code for the destructor is usually available, CSA will automatically evaluate it, in a process known as inlining.

The destructor for `std::unique_ptr` is (very) roughly as follows:

```
~unique_ptr() {
    auto &rawPtr = innerPtr.getPtr();
    if (rawPtr != nullptr)
        get_deleter()(std::move(rawPtr));
        // This defaults to `delete rawPtr;`
}
```

Inconsistent information: Suppose we don't model the `destructor` of `std::unique_ptr`.

Since the code for the destructor is usually available, CSA will automatically evaluate it, in a process known as inlining.

The destructor for `std::unique_ptr` is (very) roughly as follows:

```
~unique_ptr() {
    auto &rawPtr = innerPtr.getPtr();
    if (rawPtr != nullptr)
        get_deleter()(std::move(rawPtr));
        // This defaults to `delete rawPtr;`
}
```

Sometimes "bug" reports generated by the CSA are suppressed when we believe that they are probably false-positives. "Bugs" that come from the standard library are one such class.

But as a side-effect, the following code's leak warning (which comes a different checker, `MallocChecker`) also gets suppressed.

```
void bad() {
    auto smart = std::unique_ptr<int>(new int(13));
    auto *raw = new int(29);
    // There is a leak here.
    // But that warning gets suppressed!
}
```

But as a side-effect, the following code's leak warning (which comes a different checker, `MallocChecker`) also gets suppressed.

```
void bad() {
    auto smart = std::unique_ptr<int>(new int(13));
    auto *raw = new int(29);
    // There is a leak here.
    // But that warning gets suppressed!
}
```

Moral of the story: Model C++ classes completely to avoid such side-effects.

Closure

Closure

- We have a almost complete model for `std::unique_ptr` in `SmartPtrModelling`

Closure

- We have a almost complete model for `std::unique_ptr` in `SmartPtrModelling`
- The current modelling provides a framework for modelling other smart-pointers, such as `std::shared_ptr` or `std::weak_ptr`.

Closure

- We have a almost complete model for `` `std::unique_ptr` `` in `` `SmartPtrModelling` ``
- The current modelling provides a framework for modelling other smart-pointers, such as `` `std::shared_ptr` `` or `` `std::weak_ptr` ``.
- New developers are very much welcome! Please feel free to hit us up on the cfe-dev mailing list.