Warn if virtual calls are made from constructors or destructors

Xin Wang
25.03.2017

Abstract

In C++, virtual functions let instances of related classes have different behavior at run time. Pure function called from constructors and destructors will make the C++ program crash and virtual function called from constructors and destructors may not do what you expect. This proposal is about implementing a path-sensitive checker to find virtual calls made from constructors and destructors.

1. Introduction

A virtual function makes its class a polymorphic base class. Derived classes can override virtual functions. Virtual functions called through base class pointers/references will be resolved at run-time. A pure virtual function is a virtual function whose declaration ends in =0. A class with a pure virtual function is "abstract" (as opposed to "concrete"), in that it's not possible to create instances of that class. A derived class must define all inherited pure virtual functions of its base classes to be concrete.

```
class A {
public:
  A(int i);
  ~A() {};
  virtual int foo() = 0;
  virtual int bar();
};
A::A(int i) {
}
int main() {
  A *a=new A(1);
}
```

In the code above, the function foo() and bar() are both virtual function and foo() is also a pure virtual function. If the foo() is called during from the constructor or destructor of class A, C++ compiler will print "Pure virtual function called" (or words to that effect), and then crashes the program. The function bar() can be called during the constructor or destructor, but it may not do what you expect.

The virtual call checker is useful to check virtual function calls during construction or destruction of C++ objects. To reduce the false positives, the checker is better to be implemented in a path-sensitive way. The checker also need to use the path diagnostic to highlight both the virtual call and the path from the constructor. Finally, the checker should mark that if the virtual call is pure, because the pure virtual functions is always an error and non-pure virtual function is more of a code smell and may be a false positive.

2. State of the current virtual calls checker

The virtual call checker has already be implemented in the 'optin' package. The AST-based interprocedural analysis in the checker was turned off by default. The checker use the *StmtVisitor* to go through the AST by using the DFS algorithm. The false positives could happen when a called function uses a member variable flag to track whether initialization is complete and relies on the flag to ensure that the virtual member function is not called during initialization.

The current virtual call checker has the following functions:

1. It can check virtual function calls during construction or destruction of C++ objects by default.
2. It can check all functions reachable from a constructor or destructor by adding *-analyzer-config optin.cplusplus.VirtualCall:Interprocedural=true -DINTERPROCEDURAL=1* flag and output the call path. As this is not in a path sensitive way, it will result in false positives.
3. By adding the *-analyzer-config optin.cplusplus.VirtualCall:PureOnly=true -DPUREONLY=1* flag, the checker will output the calls to pure virtual functions only.

My plan is to re-implemented the checker in a path-sensitive way to reduce the false positives. I achieve this by constructing the CFG and implementing path sensitive code analysis by symbolic execution. This plan is discussed in more detail in the next section.

3. Goals and Implementation Details

The goals of this project can be divided into three parts:

1. Rewrite the virtual call checker to be path sensitive.
2. Use the path sensitive diagnostic rather than diagnostic message to highlight both the virtual call and the path from the constructor.
3. Evaluate if the warning should be issued for both calls to pure virtual functions and non-pure virtual functions.

The deliverables of this project should be a set of patches to the virtual call checker. I'll primarily work on the library of clang static analyzer. Most of the time it is going to be VirtualCallChecker.cpp,

As we can see, rewriting the virtual call checker to be path sensitive is the most important part of the project, to achieve this, I'll first build the CFG by walking the AST. Then using the worklist algorithm to analyze reachability and extend ExplodedGraph. This will need to work with the ExplodedGraph, the SValBuilder and the ConstraintManager, etc.

4. Project Timeline

Community bonding period (May)

Read source code and documentation of the static analyzer, make more detailed work plan and breakdown in tasks, discuss the tasks with the mentor and community.

First month of coding (June)

Determine the high-level structure of the checker's class such as the types of the events and the callback function and the ProgramState of the checker.

Second month of coding (July)

Write the code of the checker, build the CFG from the constructors and destructor, symbol execution through the CFG. Through the reachability of the ExplodedGraph, determine the place of the virtual calls and report the bug through the BugReporter.

Third month of coding (August)

Work on testing, bug fixing, incorporating mentor’s suggestions, possibly also work on documentation.

After Google Summer of Code

Nevertheless, I would stay in the LLVM community, finding bugs and providing patches occasionally. Moreover, if time permitted, let's see whether we can work together and ship more checkers to the clang static analyzer!

About Me

Personal Profile

Full name: Xin Wang
Preferred E-mail Address: wangxinds@gmail.com
IRC usernames: wangxindsb, wangxin
Time Allocated: Minimum of 20 hours per week in June (because of some exams and school activities) and a minimum of 40 hours per week on the other two months.
Previous Years: I haven't participated in the Google Summer of Code before.

Biography

I am Xin Wang, a master student in the School of Chemical Engineering at Tianjin University in China. I have always been interested in computers but haven’t systematically studied the knowledge of the computer science.

This year, I started to learn computer programming and I became a self-motivated individual, with ever more self-taught skills. Until now I have learned many things about various computing related topics, such as operating system, computer networking, compiler technique, etc. I have built a

mini-OS by consulting the source code of minix. Then I started learning the compiler technique and built a tiny-C compiler front end. I also tried to do some algorithm problems on leetcode in my spare time. By doing all of these, I have also gained experience in Linux system and learn to write program with vim. Currently I can code in a lot of languages, some of them are: Assembler, C/C++, Java, Python. The language I use most and that I am more experienced with is C and C++. My development tools include Vim, the GNU compiler collection and the GNU debugger. I am also familiar with the gdb debugger which will be useful to debug the checker.

The GSoC 2017 is the first time I participate in a computer project and I think it will become a window to contribute to the open source community. Although this is my first time to write code for the open source community, I'm not unfamiliar with it. I know something about github and fork some project to configure my router. I am skilled in using tools of open source community such as git, svn, mailing lists and IRC. I learn quickly, I'm not afraid to read deep code and I have the motivation to do this project. Hopefully I can fulfill all that is required to do so.

Reference

[1] http://clang-analyzer.llvm.org/checker_dev_manual.html
[2] http://www.artima.com/cppsource/pure_virtual.html
[3] https://reviews.llvm.org/D26768
[4] https://llvm.org/svn/llvm-project/cfe/trunk/lib/StaticAnalyzer/Checkers/VirtualCallChecker.cpp