Ben Mishkanian

Cppcheck vs Clang Static Analyzer: A Case Study

Introduction

As software projects have increased in complexity, the difficulty of finding bugs has increased, yet the need to release fast and often has not changed. Developers need quick, effective tools to find bugs automatically. This paper describes a user study of Cppcheck and Clang Static Analyzer, two static analyzers for C. These tools claim to automatically find several types of common programming errors, such as null pointer dereferences. However, they use different bug-finding algorithms, and as a result they different in the number, type, and quality of bugs they report. Cppcheck checks for bounds violations, memory leaks, null pointer dereferences, uninitialized variables, invalid STL usage, exception safety, unsafe function usage, and dead code. Clang Static Analyzer can check for division by zero, null pointer dereferences, uninitialized variables, dead code, insecure API usage, and other platform-specific errors. These tools are both free for non-commercial use.

In this paper, we evaluate Cppcheck and Clang Static Analyzer along several dimensions:

- Real bugs found
- False positives found
- Ease of installation and use
- Classes of bugs found
- Sophistication of bug detection

We perform two case studies:

The first case study is to write a program filled with simple bugs that would be obvious to the average programmer, and run the tools on this program to get qualitative information on the capabilities of the tools. This program is called buggyprogram.c. It includes some “false bugs” to try to trip up the tools and make them generate a false positive.

The second case study is to run the tools on a medium-sized open-source project to see if they can find real bugs in a real project. Some of the bugs reported will be investigated to determine if they are real bugs or false positives.

Motivating Example

Consider the following code snippets from buggyprogram.c:

```
// 1. Null Pointer Dereference
int *p1;
p1 = NULL;
*p1 = 3;
```

```
free(p1);

// 2. Null Pointer Always Properly Allocated Before Dereference
int *p2;
p2 = NULL;
if (1) {
      p2 = (int*)malloc(sizeof(int));
}
*p2 = 3;
free(p2);
```

Part 1 contains an obvious null pointer dereference. Part 2 contains null pointer that is conditionally allocated, but the condition is always true, so there is technically no possible null pointer dereference in this situation. In our experiment, Cppcheck flags both p1 and p2 as “possible null pointer dereference” and in a later message, flags p1 as a null pointer dereference.

Clearly, even in a simple case like this, Cppcheck is unable to rule out the fact that p2 is involved in a bug, even though it can never happen. Although it does make a distinction between possible bugs and real bugs (since it flags the p1 access as a confirmed bug), the message that the p2 access may also be a bug adds noise. In a large project, where many of these “almost-bugs” can exist, a developer may ignore everything but the confirmed bugs, as it would take too long to manually check which of the maybe-bugs are real ones, and which can never happen. On the other hand, these messages may still be useful as indicators of poor coding style. After all, unsafe design patterns like the one above may someday become bugs, after changes to the control flow or branching conditions.

The overall purpose of this study is to evaluate the strengths and weaknesses of each tool, by discovering behaviors such as the one described above.

Technical Approach and Evaluation Methodology

Part 1 – Testing Tool Capabilities

buggyprogram.c contains several types of bugs, and for each bug type, it has some variations on that bug that are designed to test the sophistication of the static analyzers. For example, the null pointer dereference test has three variations: the basic obvious null pointer, a null pointer initialized in a branch that is always taken, and a null pointer that is initialized if the current time is divisible by 2. Each of these tests a different capability. The first one simply tests if the tool can identify (with high confidence) a bug that a developer would easily identify. The second one tests if the tool can track branching possibilities to identify when a pointer is always allocated before use, and thus not issue a warning. The third one evaluates whether the tool can catch a nondeterministic bug, and if it gets any distinction from the other two cases. Ideally, it would be flagged as a nondeterministic bug (or at least a confirmed bug), since it can

occur under some circumstances. We prefer that it not be flagged as a “possible” bug, since we know with 100% certainty that it sometimes gives the wrong result.

The tests in buggyprogram.c provide some insight into the behavior of these tools. From there, we can evaluate the strengths and weaknesses of each one. The test can be downloaded from Github: https://github.com/benmishkanian/static-analyzer-comparison

Part 2 – Testing Tool Effectiveness

We will run the two tools on a mid-sized open source project downloaded from Github to see how well they perform on real projects. Evaluation criteria includes number of confirmed bugs, unconfirmed bugs, false positives, and false negatives, as well as the usefulness of the bug reports and the importance of the bugs found. Lastly, we evaluate the usability of the programs, including the difficulty of setup and configuration.

Initial Results of Experimental Evaluation

Basic Differences

From the start, there are some basic differences between Cppcheck and Clang Static Analyzer. Although the source code for both tools are available, Cppcheck only has precompiled binaries for Windows, and Clang Static Analyzer only has precompiled binaries for Mac. Compiling Cppcheck requires Qt, and Clang Static Analyzer has multiple dependencies. As a result, the time investment required to get up-and-running with either of these tools depends greatly on your OS. These tools also have different UI bindings and available plugins. For example, Cppcheck has plugins for several additional development environments, including Eclipse and gedit, which Clang Static Analyzer does not. Cppcheck also has an explicit design goal of having no false positives, while Clang Static Analyzer aims for “a low false positive rate for most code on all checks."

Null Pointer Dereference

Cppcheck does a decent job of detecting null pointer dereferences. It handily confirms the obvious case of a pointer that is dereferenced directly after nullification. However, case #2, which contains a pointer that is initialized in an always-taken if-block, is flagged as a possible bug as well, which is not true. Case #3, which contains a nondeterministically initialized pointer, is also flagged as a possible bug, which is a sub-optimal result, since it is a real bug that is triggered some of the time. Overall, Cppcheck provides conservative answers. It meets its design goal of having no false positives in this case, but it does so at the cost of precision, making the “possible bug” messages much less reliable and useful.

Related Work

Wikipedia (http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C) provides a list of static analyzers, with an overview of their design goals. In class, we looked at several other tools which automatically find bugs. We learned about KLEE, an

automated symbolic execution system that generates and executes test cases to find bugs. A similar tool is CUTE (now known as CREST), which is a concolic testing tool used to achieve high code coverage. Although these tools can be effective in finding bugs, they require manually instrumenting the source code, so they do not fill the need for a tool that provides quick, quality bug reports “out of the box.” Furthermore, CREST does not generate bug reports; it only reports the code coverage it achieved. There are also plenty of other static analysis tools for C, but many of them are commercial software. In this paper, we focus on the Cppcheck and Clang Static Analyzer because they are readily available and generate bug reports without having to instrument the code.