

Code Clone Detection in Clang Static Analyzer

Detecting Copy-paste using Static Analysis

Kirill Bobyrev & Vassil Vassilev
MIPT & CERN

kirillbobyrev@gmail.com
vvasilev@cern.ch



Motivation

The copy-paste is a common programming practice. Recent studies [?] show that large 5-20% of code in large codebases is either equal or similar to other code pieces in this project. This causes troubles and makes development process way harder. Having a code clone in a system almost always means that its design needs serious improvements. Large projects are developed by numerous people and fixing a bug found in one code clone instance may cause unexpected results, because this fix most likely won't be applied to every other instance similar to the fixed one. Thus said, the practice of copy-pasting a piece of code into multiple locations. Having a tool that detects these clones and produces a human-readable report allows finding such pieces of code and improving the architecture.

Infrastructure choice

Even though such tools exist, most of them are not easy to use. Most of them are also outdated and because of that they can't catch up with the latest language standards and features. Building a Code Clone Detection Tool around Clang and Clang Static Analyzer allows not worrying about these problems. LLVM and Clang provide a great infrastructure for building such a tool and provide the stability. Therefore the tool is implemented as additional checkers for the Clang Static Analyzer. Clang SA along with scan-build are really easy to use and can generate informative reports containing all the warnings.

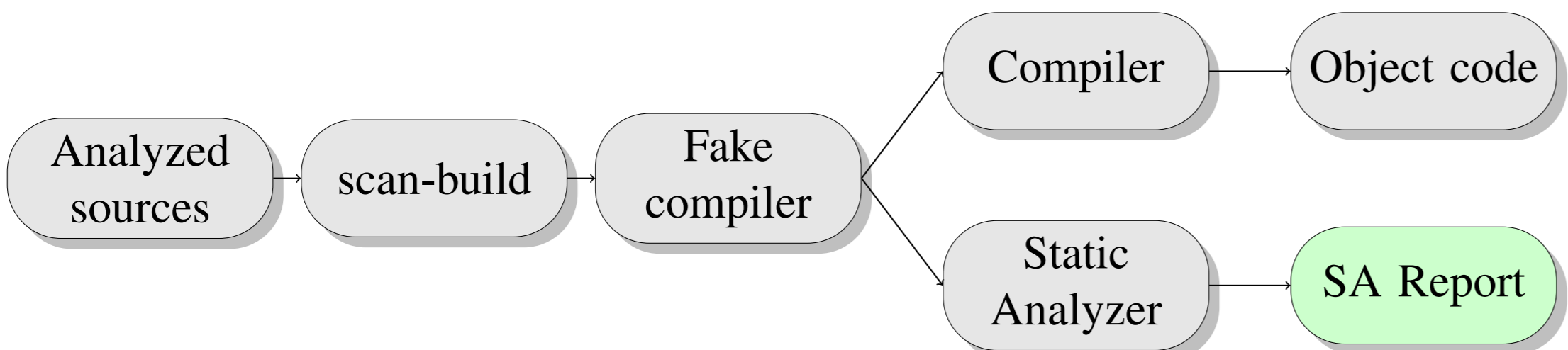


Figure 1: Full process of analysis and compilation

Clone types

Let's quickly review how clones are different from each other and which types are there to get a better understanding what can the current implementation do and what can it not.

Type I Identical code fragments except for variations in whitespace, layout and comments.

Type II Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type III Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type IV Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

clone.BasicCloneChecker

BasicCloneChecker is designed to be scalable and therefore we're focusing on high precision and speed rather than high recall. *BasicCloneChecker* is assumed not to cause huge compilation time losses and performance issues. This checker defines a "clones" to be a certain pair of AST subtrees.

This checker processes AST generated by Clang and performs enhanced Profiling technique used to match AST nodes and their subtrees. It uses hashing based on AST nodes' Profiles to divide AST nodes into clone groups and reports each clone group separately cutting-off few false-positives and ensuring that the AST tree accused of being a clone contains more than THRESHOLD AST nodes.

BasicCloneChecker is able to find code clones of types I and II with high precision. However, it is very sensitive to the slight changes in analyzed AST.

Computational complexity of analyzing AST tree with n nodes is $O(n^2)$. The heaviest part of implementation is building of profiles for an AST node and its subtrees. It's easy to understand that incremental profiling + caching may reduce the complexity to $O(n)$, but checker relies on current recursive implementation of profiling.

clone.AdvancedCloneChecker

AdvancedCloneChecker combines the Tree- and PDG-based techniques and implements the approach proposed by Jens Krinke in his paper [?]. It detects clones of type 1, 2 and 3. It relies on the implemented Fine-Grained PDG data structure, which is basically a variation of PDG built from AST nodes. Then it runs the maximum similar subgraph searching algorithm to detect similar FGPDG subgraphs and creates if these subgraphs contain more than THRESHOLD nodes.

The general problem "Finding maximum similar subgraphs" is NP-complete, that is why the implemented algorithm is just an approximate approach, which is the best thing one could possibly do.

Granularity is very important part of this approach, because it only processes the comparison units and the partitioning part is essential. If chosen badly, it can lead to the state, in which the checker is right now: even if the algorithm extracts subgraphs with fairly good accuracy not many clones can

be detected, because the comparison units are either too big or do not fit the needed conditions. The implemented algorithm partitions into connected components, which are compared against each other afterwards. This approach has many negative effects I encountered while processing large codebases.

AdvancedCloneChecker should be used for a deeper analysis; it processes an AST with n nodes in $O(n^4)$ time and it can not be reduced. It's quite slow, but the approach allows to detect even type 3 clones.

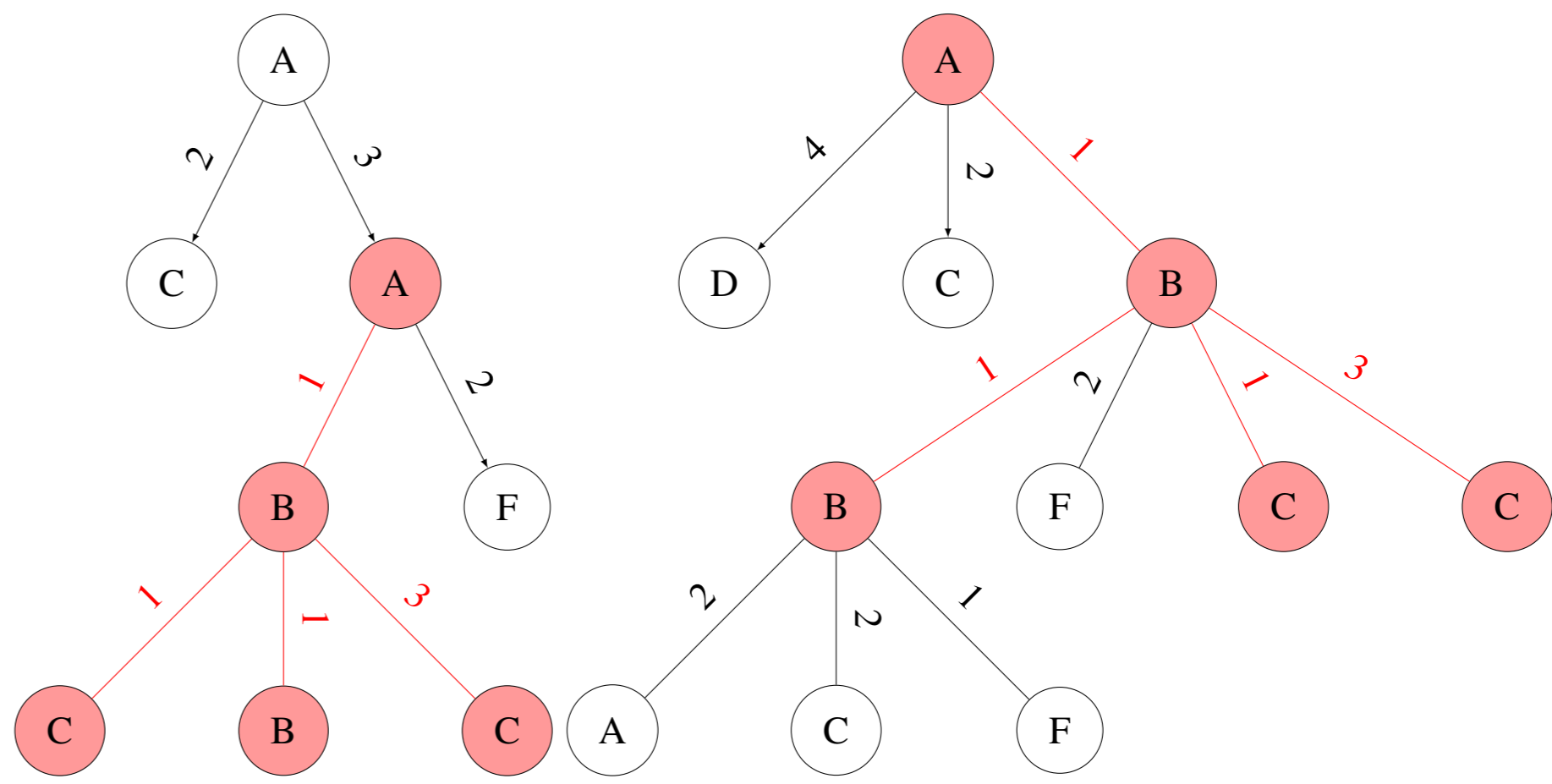


Figure 2: Similar subgraphs

Common problems

Both checkers are built in Clang Static Analyzer and therefore have certain restrictions.

- Clang SA design doesn't allow the analysis of the whole codebase, it only allows analyzing each Translation Unit separately. Obviously, some code clones may occur in different Translation Units.
- The diagnostics generated by both checkers are quite specific as warnings of each clone class objects should come consequently. Therefore these diagnostics aren't hooked to Clang SA BugReporter and aren't displayed in scan-build report.
- Both checkers use magic numbers (THRESHOLD) while deciding whether to report a code piece or not. It would be logical to give a control of these constants via some Clang SA options. However, this isn't done for any of the existing SA checker and I doubt that it meets expectations of the original Clang SA developers.

Getting real

Even though both checkers are WIP, they detected code clones in few built open-source project. Here's a performance report for each of built projects using *BasicCloneChecker* only. I only count clones with more than 50 AST nodes.

```
scan-build --use-analyzer=${PATH_TO_BUILT_CLANG} \
-enable-checker clone.BasicCloneChecker \
-disable-checker core \
-disable-checker unix \
-disable-checker cplusplus \
-disable-checker deadcode \
-k make
```

Project	without fakec invocation	BasicCloneChecker	Clones found
LLVM + Clang	18m10s	17m53s	0
OpenSSL	1m26s	9m27s	180
Git	0m26s	2m46s	34
SDL	0m26s	1m59s	170

Table 1: Running time measurement

Conclusions & Future plans

The results show that the *BasicCloneChecker* is stable and scalable enough, though it needs few enhancements, which will make it even better and production-ready. However, *AdvancedCloneChecker* is totally experimental and needs serious improvements even though it detects clones in large codebases.

Acknowledgements

I would like to thank ISP RAS researchers who were so kind to share the results of their work with me.

I should also thank Nick Lewycky, who did some initial work on detecting useless conditions for Clang. My first checker borrows few ideas from that work.