Poster: Implementation and Evaluation of Cross Translation Unit Symbolic Execution for C Family Languages*

Gábor Horváth†
Péter Szécsi
Zoltán Gera
xazax@caesar.elte.hu
ps95@caesar.elte.hu
gerazo@caesar.elte.hu
Eötvös Loránd University,
Department of Programming
Languages and Compilers
Budapest, Hungary

Dániel Krupp
Ericsson Ltd.
Budapest, Hungary
daniel.krupp@ericsson.com

Norbert Pataki‡
Eötvös Loránd University,
Department of Programming
Languages and Compilers
Budapest, Hungary
patakino@elte.hu

ABSTRACT

Static analysis is a great approach to find bugs and code smells. Some of the errors span across multiple translation units. Unfortunately, it is challenging to achieve cross translation unit analysis for C family languages.

In this short paper, we describe a model and an implementation for cross translation unit (CTU) symbolic execution for C. We were able to extend the scope of the analysis without modifying any of the existing checks. The analysis is implemented in the open source Clang compiler. We also measured the performance of the approach and the quality of the reports. The implementation is already accepted into mainline Clang.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Information systems →** *Open source software*; • **Software and its engineering →** *Software maintenance tools*;

KEYWORDS

static analysis, symbolic execution, cross translation unit, Clang

ACM Reference Format:
Gábor Horváth, Péter Szécsi, Zoltán Gera, Dániel Krupp, and Norbert Pataki. 2018. Poster: Implementation and Evaluation of Cross Translation Unit Symbolic Execution for C Family Languages. In *Proceedings of 40th International Conference on Software Engineering Companion, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18 Companion),* 2 pages.
https://doi.org/10.1145/3183440.3195041

*Produces the permission block, and copyright information
†Supported by ELTE Eötvös Loránd University in the frame of the ÚNKP-17-3 New National Excellence Program of the Ministry of Human Capacities.
‡Supported by ELTE Eötvös Loránd University in the frame of the ÚNKP-17-4 New National Excellence Program of the Ministry of Human Capacities.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195041

1 INTRODUCTION

Symbolic execution [**?**] is a major *abstract interpretation* technique. Symbols are used to represent unknown values (e.g. user input), and symbolic calculations are carried out on them.

The open-source *Clang* compiler [**?**] has a component called *Static Analyzer* (SA). It implements a powerful symbolic execution engine for C, C++, and Objective-C. Despite being more than 10 years old, it did not support CTU analysis until our recent addition.

The scope of the analysis has a big impact on precision. Its extension also has the benefit of eliminating false positives by inferring infeasibility for more paths.

In this short paper, we describe a model and an implementation for cross translation unit analysis in Clang. We target the following research questions: Can we improve analysis precision? How does the quality of the findings change? How does the execution path length of the errors change? Is there any coverage loss when we extend the scope of the analysis? Is our method scalable with respect to memory and time?

2 THE CLANG STATIC ANALYZER

The Clang SA builds an *exploded graph* [**?**] during symbolic execution. The analyzer collects constraints on the symbolic expressions using a constraint solver, which are then used to skip the analysis of infeasible paths and to hold information about program states. If the analyzer finds a critical issue it stops the analysis on that execution path.

The SA also supports context-sensitive interprocedural analysis. When there is a call to a function with a known body, the analyzer can continue the analysis inside the callee while preserving all the information from the call site. This is called *inline analysis*. A function interpreted without the calling context is called a *top level function*.

It is intractable to enumerate all execution paths as the number of paths is exponential in the branching factor. The analyzer employs various thresholds to limit the size of the exploded graph which are collectively called the *analysis budget*. While extending the scope of the analysis, it is important to measure coverage loss so that the analysis budget is not exhausted before interesting issues are found.

3 CROSS TRANSLATION UNIT ANALYSIS

Originally, the SA analyzed each translation unit in one pass. To implement cross translation unit analysis we introduced a *two-pass analysis*. The first pass parses all translation units and serializes their ASTs (*Abstract Syntax Trees*) to disk. We also create an *index* which holds function definition locations. We use USR's (*Unified Symbol Resolution*) as definition identifiers, which can be generated for each declaration by Clang.

In the second pass, the SA analyzes each translation unit. When it encounters a call to a function that does not have an implementation in the current translation unit, it searches for the function in the index. If the definition is found, the analyzer can locate the corresponding AST dump and load its content into the memory.

After loading the dump into the memory we have two separate ASTs with separate symbol tables and separate representations of the same types. Their merging can be accomplished by Clang's `ASTImporter` module, although its current lack of C++ support limits its use to projects written in C. Unfortunately, the size of the AST dumps can be significant. We implemented on-demand reparsing for users who would like to reduce disk usage. According to our measurements, this version was about 30% slower. As both merging the AST and loading it from disk can be expensive, we keep loaded ASTs in memory and never do the merge twice.

The analyzer might find the same issue on multiple execution paths. It deduplicates these reports and only reports the shortest one. If we use CTU, multiple invocations of the analyzer might find the same issue in a similar manner, but will not be able to deduplicate the results. We can use a third optional pass to deduplicate these findings. All the measurements in the next section regarding the number of findings and their quality are done after deduplication.

4 MEASUREMENTS

We have measured different aspects of the analysis on popular industrial-scale open-source projects. The number of analyzed paths significantly increased by extending the scope of the analysis, on average by a factor of 4. The number of times a function was evaluated conservatively due to reaching the call depth limit is the most significant increase, larger than an order of magnitude. Other cuts are increased in proportion to the number of paths.

We observed a 1.5 to 3 times increase in the number of reports with the median being close to 2. It is important to note that some errors were only reported during regular analysis, which is partly due to changes in the coverage pattern. While we did analyze more execution paths with CTU enabled, we exhausted the analysis budget earlier. Some of the findings might be lost because they were false positives. The number of lost findings is small compared to the total findings.

We also measured path length medians. The longer the path, the harder it is to understand the error [**?**], and the higher the chance that the path is infeasible. As expected, findings that span across multiple translation units usually have longer execution paths, but a decrease was also spotted for some projects.

Report quality has not declined, their understanding was not hindered by longer execution paths. False positive findings were not the byproduct of the implementation, the analyzer would report the same errors using a unity build. We identified true positive results.

Analysis time and memory increased roughly in proportion with the number of findings. When only a single translation unit can be analyzed at a time, the user cannot truly adjust cut thresholds to balance between analysis time and the number of findings, since translation unit barriers represent a hard upper limit on the latter. After extending analysis scope there is more space to adjust these parameters. According to the measurements, it is still feasible to run the analysis on a regular personal computer.

Previously the analyzer had not reported any coverage information, so we had to implement this feature. In general, coverage slightly decreased with extended scope, some of which might be retained by increasing the performance budget. It is important to note that some of the coverage loss is intentional, as the analyzer discovers the infeasibility of more paths. Also, there are more findings which terminate the analysis of a given path. Unfortunately, it is a hard problem to determine whether a certain uncovered code is not covered deliberately or is merely the result of a cut in the analysis.

5 CONCLUSION

Cross translation unit support can improve the precision of symbolic execution significantly. This implies both useful new findings for users and less false positive results to deal with.

We implemented cross translation unit analysis support in a popular open-source compiler. Our solution did not introduce any changes to the existing checks and only minor changes to the analyzer engine. According to our measurements the coverage, the analysis performance, and the quality of results are satisfactory and there is an improvement in the number of results. In the future, we plan to improve the cut heuristics.

The main setback of cross translation unit analysis is the lack of support for consistent results when we use incremental analysis. Solving this issue is going to take a significant amount of research and engineering effort.

ACKNOWLEDGMENTS

We would like to thank Artem Dergachev and Aleksei Sidorin for the first implementation prototype [**?**] of the cross translation unit analysis that we reused and improved upon to perform our measurements.