Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

Comprehensive Evaluation of Cross Translation Unit Symbolic Execution*

Endre Fülöp, Norbert Pataki

ELTE Eötvös Loránd University, Budapest, Hungary
Faculty of Informatics, 3in Research Group, Martonvásár, Hungary
gamesh411@gmail.com,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 (TUs). Symbolic execution is a primary static analysis technique. Symbols are used to represent unknown values (e.g. user input), and symbolic calculations are carried out on them. Clang Static Analyzer (SA) is an open-source symbolic execution engine for C/C++/Objective-C. The default behaviour of the SA does not support cross translation unit analysis, but it can be parametrized to enable analysis techniques spanning across many TUs.

In this paper, we evaluate the cross translation unit symbolic execution in a comprehensive way. Different caching methods, different approaches are considered. We compare the analysis of open source projects. The aim is an optimal configuration for the tool.

Keywords: symbolic execution, cross translation unit, Clang

MSC: 68N15 Programming languages

CEUR-WS.org/Vol-2650/paper12.pdf

1. Introduction

Static analysis is a well-known method to detect bugs without execution of code [7]. Static analysis works with source code, mainly focuses on bug detection [3]. However, refactoring, obfuscation and complexity metrics tools also use static analysis. Static analysis tools build up the abstract syntax tree (AST) in order to run

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications)
Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

b.cpp:

```
int f( int& r );

void g()
{
  int x = 14;
  f( x );
  if ( x != 15 )
  {
    int * p = new int;
  }
}
```

a.cpp:

```
int f( int& r )
{
  ++r;
  return 100 / ( r - 15 );
}
```

Figure 1: TUs with cross-referencing function definitions

AST-consumers on them, which are used to implement algorithms over syntax trees [5].

Symbolic execution is a major static analysis in which symbols are used to represent unknown, and calculations are carried out on them [9].

Unfortunately, separate compilation makes cross translation unit analysis challenging for C family languages. Therefore, many tools do not support cross translation analysis [1]. Unity build is a technique for using a single translation unit, but creating unity builds also has many difficulties [10]. However, the scope of the analysis has a significant impact on the precision [6].

Let us consider the code snippets on Figure 1 that belong to two different translation units:

If one were to analyze the first translation unit via single-TU analysis, starting the symbolic execution from function `f`, the value of the parameter could not be reasoned about. Therefore producing a warning for the return statement, and stating that the expression is potentially a division by zero is not in line with the conservative policy of static analysis. The goal of the analyzer is to identify real bugs and help the programmer fix error-prone code constructs. However, too many bug-reports are also discouraged for practical reasons. In function `g`, no knowledge about the value of `x` right after the function call to `f`. In this case, one gets warning about the memory leak, but it is dead code; thus it is a false positive finding.

C/C++ programmers have been eager for a more precise solution, therefore we improve the Clang SA for the cross translation unit analysis, but the potential configuration settings of the new version have not been evaluated.

The rest of this paper is organized as follows. We present the approach of cross translation unit symbolic execution in section 2. We define what are the parameters of the improved analysis in section 3. We evaluate the analysis processes and present results in section 4. Finally, this paper concludes in section 5.

2. Cross Translation Unit Symbolic Execution

Clang Static Analyzer (SA) is a powerful symbolic execution engine for the C/C++ and the Objective-C languages. Moreover, it is based on the Clang compiler infrastructure [2]. However, it was not able to perform cross translation unit analysis for a long time. However, many problems span across multiple translation units. We improved it to achieve a more sophisticated approach [6].

The SA used a one-pass analysis initially, however, the CTU analysis needs preprocessing on the project. Thanks to this dependency, we we had to extend the analysis driver to support two-pass analysis. In the first phase, an index file is created based on the compilation database and source code. This index file contains the mapping of function definition and translation unit. The source code is parsed; thus the AST is built. The ASTs are serialized in binary format. In the second phase, SA analyzes all translation units. When it reaches a function call that has no definition in the current unit, SA finds its serialized AST snippet based on the index file. In this case, a unique merging approach is required. Both the compiled in-memory form of the current file's AST and the binary serialized external one contain their distinct symbol tables and type representations. They have different managers regarding the source locations as well. Loading and merging ASTs have runtime cost; therefore we developed caching mechanisms.

3. Evaluation

The symbolic execution engine of Clang SA implements interprocedural analysis by inlining the definition of the called function when the analysis reaches said function call. This inlining is not always performed however, for example if the definition is not available inside the translation unit. Single-TU analysis will disregard the call expression, and performs some invalidation on values possibly reachable by the function (e.g. parameters taken by reference, global variables). CTU analysis makes the definitions from other TU-s available to the analyzer, therefore increasing the number inlined functions, and at the same time decreasing the number and effect of invalidations. The analysis proceeds with the consumption of the statements of the function body as if they were lifted into the current scope. The analysis employs thresholds to limit the execution time of the analysis.

The configuration of the CTU symbolic execution contains some settings. These settings affect the runtime performance and memory consumption significantly [4]. With inlined function definitions, the analyzer has the capability of exploring a bigger part of the project code. This does not necessarily lead to more bugs found, but measurements seem to indicate, that a higher amount of bugs is detected when the analysis is ran in CTU mode.

One of the most critical parameters is the maximal number of translation units to process when an external code snippet is required for a more sophisticated approach. On the one hand, the increase in this parameter means more precise

analysis. On the other hand, there are limitations in the symbolic execution engine to cancel the analysis, even if the maximal number of allowed TUs is not exceeded.

There are two different modes for loading external AST. The users can select between the two-pass analysis and on-demand loading of external AST approaches.

We provide caching mechanisms, as well. Function-wise and translation unit-wise mechanisms are supported. If an external code snippet is required, function-wise solution means that only the AST of the called function is cached. The translation-unit-wise approach provides the caching the AST of the entire translation unit, not just part that belongs to the called function.

We tested the CTU symbolic execution with respect to the following projects:

- Tmux[1], an open-source project written in C
- Xerces[2], an open-source project written in C++

The following metrics were collected:

- Wall time of execution
- Resident memory usage
- Disk usage of the analysis

We collect the metrics based on the following parameters:

- Method used
  - non-CTU as the baseline
  - AST-dump based CTU
  - on-demand-parsed CTU
- TU unit threshold

4. Results

The measurements were run on a Intel(R) Xeon(R) CPU X5670 @2.93GHz workstation with 24 virtual cores. Each measurement is driven by CodeChecker[3], with the help of run orchestrator CSA testbench[4]. The runtime metrics wall clock time, and memory usage were taken with time tool[5].

The evaluation of CTU analysis methods shows a definite increase in both analysis time and result-count in case of both simple and on-demand modes for Tmux as seen on Table 1 and for Xerxes on Table 2. The CTU modes make

[1]`https://github.com/tmux/tmux`
[2]`https://github.com/apache/xerces-c`
[3]`https://github.com/Ericsson/CodeChecker`
[4]`https://github.com/Xazax-hun/csa-testbench`
[5]`https://www.gnu.org/software/time`

Method	Threshold	Bugs	Time (s)	Max memory (kB)	Disk (kB)
Non-CTU	N/A	21	662.04	190128	1294
Dump based CTU	0	21	779.71	190948	166403
	8	36	1143.99	240500	167481
	16	121	1715.64	290636	173590
	24	154	1946.96	332740	176691
	32	159	2073.77	376432	177159
	40	161	2096.85	421852	177185
	48	162	2074.01	442892	177253
On-demand CTU	0	21	719.13	190636	1405
	8	36	1261.06	245156	2577
	16	121	1975.93	300012	8733
	24	154	2281.72	347184	11954
	32	159	2426.03	393232	12473
	40	161	2482.41	442000	12512
	48	162	2489.72	467704	12586

Table 1: Tmux analysis method comparison

Method	Threshold	Bugs	Time (s)	Max memory (kB)	Disk (kB)
Non-CTU	N/A	109	1437.24	236404	6492
Dump based CTU	0	109	1607.94	239104	637517
	8	225	10858.03	316660	683215
	16	289	17461.60	363252	693284
	24	335	21300.30	426760	697806
	32	362	22549.00	488412	699926
	40	366	23630.78	536008	701161
	48	368	23938.71	559980	701594
On-demand CTU	0	109	1544.32	237916	6597
	8	225	12084.36	323416	52295
	16	296	21402.23	382732	62364
	24	329	25186.48	456744	66886
	32	355	26988.73	536848	69006
	40	360	28349.59	591512	70241
	48	362	28790.37	614456	70674

Table 2: Xerces analysis method comparison

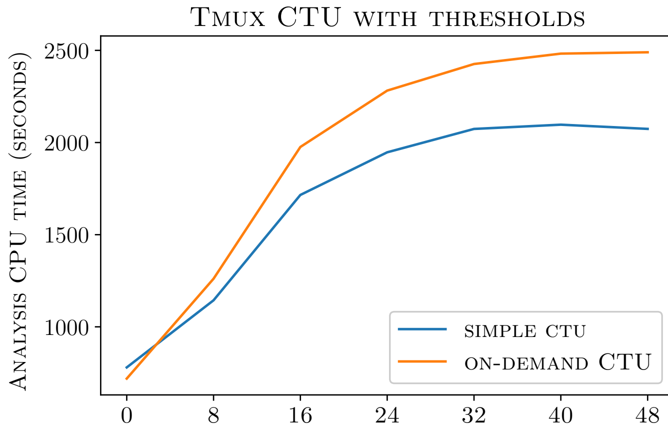

Figure 2: Analysis time vs TU threshold (Tmux)

the analyzer a more significant part of the project available, thus increasing the amount of information accessible to the analyzer. The runtime cost of the different analysis methods can be seen on Figure 2 and Figure 3 for Tmux and Xerces respectively. The increase in result-count could also potentially mean that more false positives are produced. The programmer must make the decision whether a finding is positive or not on an individual basis. This means that CTU analysis could potentially provide more results in the project at the cost of an increase in the development time. The results also show that the nature of bugs being found varies, as multiple domains get connected by CTU, that are separated by modularization. For example, bugs concerning memory access are found along deeper bug paths, as the memory handling logic is most of the time separated into different translation units. CTU analysis introduces more statements to be analyzed. These statements use the same budget as the non-imported ones. Even with the same statement-budget value, the exact characteristics of found bugs depend on many factors, including the path-exploration strategy employed, the position of the inlined calls, and the structure of the inlined functions body. The CTU analysis therefore can lead to deeper bug paths as well as shallower ones. Consider the example of a function which uses a call very early during its execution. In the non-CTU case, the call is ignored, and execution proceeds with statements after it. There are some bugs found with either deep or shallow paths. Now consider the analysis of the same function but with CTU mode enabled. The aforementioned call is now matched with a definition from another TU, and is inlined. There is a possiblity now that the inlined function is very complex, or maybe more functions are inlined during the evaluation of said function. This could lead to early exhaustion of the analysis budget, and potentially the exclusion of the latter half of the original functions body. Totally disjoint sets of results are possible.

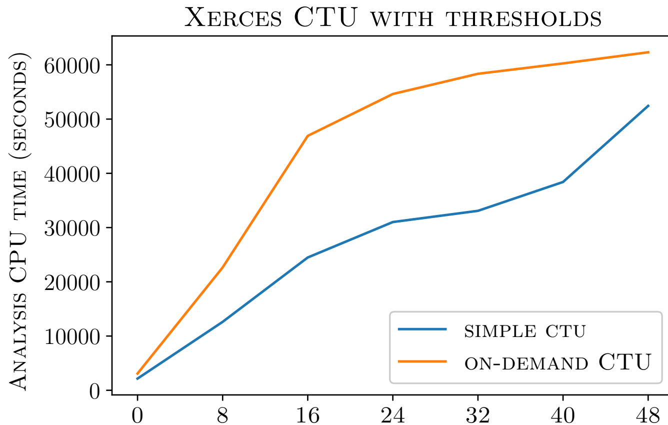

Figure 3: Analysis time vs TU threshold (Xerces)

There is also an interesting behaviour in case of analyzing projects up to their *CTU-threshold-limit*. TMUX was analyzed with a TU threshold of 100 in cases the threshold value was not explicitly mentioned. During this analysis we have tracked the messages of the analyzer. We found that even if a maximal amount of 100 was given, there were no TUs that triggered an import of more than 47 other TUs. This means that with default settings, the analyzer considered no more than this amount TUs during analysis, which we call the *CTU-threshold-limit*. So with thresholds ranging from 0 (which signifies equivalent behaviour as non-CTU) up to 48, the whole range of possible values were measured. Thus the threshold charts saturation-like shape in case of bugs found.

5. Limitations and Conclusion

The disk usage of CTU analysis could prove to be a significant hindrance, as there is a usually a magnitude of difference between the size of the results produced, and the size of the dumped AST-nodes in case of simple CTU analysis. The on-demand CTU analysis can mitigate this, but at the cost of parsing the source files on-the-fly. These tradeoffs should be considered before switching analysis modes.

On-demand CTU analysis has further weaknesses. The produced AST is not precisely equivalent to the dumped AST. We have identified three possible causes of the non-equivalence. One could be that the serialization and deserialization steps do not give back the original AST, this is still under investigation. Another reason could be the liberal handling of language elements in case of creating serialized AST dumps, as opposed to be more strict policy employed during on-demand parsing. This is more likely, but the exact implementation details of narrowing the gap is under revision. The last reason could be the liberal detection of compilation

command flags, and is deemed the least probable. Further investigation is needed, but recent discussions suggest that an architecturally more robust and more scalable approach could circumvent the first two reasons. In the case of medium-sized projects, the differences of AST are not numerous enough to produce different results; however, in the case of more complex projects, even the bugs are found differ. Currently we are verifying whether the AST serialization is to be held accountable for this phenomenon.

We found that the caching is necessary for the analysis to be successful. Clang is relying on every part of the AST to be held inside the memory. When switching off the caching, we found that the analysis asserted on multiple invariant properties of the AST being violated, as the AST nodes were already destructed when the analysis reached them.

References

[1] Anand, S., Godefroid, P., Tillman, N., Demand-driven Compositional Symbolic Execution, in Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 367–381.

[2] Arroyo, M., Chiotta F., Bavera F., An user configurable Clang Static Analyzer taint checker, in Proc. of the 2016 35th International Conference of the Chilean Computer Science Society (SCCC), pp. 1–12.

[3] Babati, B., Horváth, G., Májer, V., Pataki, N., Static Analysis Toolset with Clang, in Proc. of the 10th International Conference on Applied Informatics (ICAI 2017), pp. 23–29.

[4] Baldoni, R., Coppa, E., Cono D'elia, D., Demetrescu, C., Finocchi, I., A Survey of Symbolic Execution Techniques, *ACM Computing Surveys*, Vol. 51(3) (2018), Article No.: 50.

[5] Emanuelsson, P., Nilsson U., A Comparative Study of Industrial Static Analysis Tools, *Electronic notes in theoretical computer science*, Vol. 217 (2008), pp. 5–21, 2008.

[6] Horváth, G., Szécsi, P., Gera, Z., Krupp, D., Pataki, N., Challenges of Implementing Cross Translation Unit Analysis in Clang Static Analyzer, in Proc. of 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM 2018), pp. 171–176.

[7] Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R., Why don't software developers use static analysis tools to find bugs? in Proc. of the 2013 International Conference on Software Engineering, ICSE '13. (2013), pp. 672–681.

[8] Joshi, A., Tewari, A., Kumar, V., Bordoloi, D., Integrating static analysis tools for improving operating system security, *International Journal of Computer Science and Mobile Computing*, Vol. 3(4) (2014), pp. 1251–1258.

[9] King, C., Symbolic execution and program testing, *Communications of the ACM* Vol. 19 (1976), pp. 385–394.

[10] Mihalicza, J., How #includes Affect Build Times in Large Systems, in Proc. of the 8th International Conference on Applied Informatics (ICAI 2010), Vol. 2, pp. 343–350.