A dynamic algorithm for source code static analysis

Pavel Mezhuev
Chong-Ming laboratory,Moscow Research Center
Huawei Technology Co., Ltd.
Moscow, Russia
pavel.mezhuev@huawei.com
ORCID: 0000-0002-1838-8440

Alexander Gerasimov
Chong-Ming laboratory,Moscow Research Center
Huawei Technology Co., Ltd.
Moscow, Russia
gerasimov.alexander@huawei.com
ORCID: 0000-0001-9964-5850

Petr Privalov
Chong-Ming laboratory,Moscow Research Center
Huawei Technology Co., Ltd.
Moscow, Russia
petr.privalov@huawei.com
ORCID: 0000-0002-8939-5824

Veronika Butkevich
Chong-Ming laboratory,Moscow Research Center
Huawei Technology Co., Ltd.
Moscow, Russia
butkevich.veronika.nikolaevna@huawei.com
ORCID: 0000-0001-9376-9051

2021 Ivannikov Memorial Workshop (IVMEM) | 978-1-6654-2327-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/IVMEM53963.2021.00016

***Abstract*—A source code static analysis became an industrial standard for program source code issues early detection. As one of requirements to such kind of analysis is high performance to provide response of automatic code checking tool as early as possible as far as such kind of tools integrates to Continuous testing and Integration systems. In this paper we propose a source code static analysis algorithm for solving performance issue of source code static analysis tool in general way.**

Keywords—source code static analysis, static analysis algorithm

I. INTRODUCTION

A source code static analysis (SCSA) is widely used in secure software development life cycle. Modern tools for SCSA tend to be integrated to continuous testing end integration environment for security critical software development as far as integrated to developer workspace in form of plugins to commonly used integrated development environments such as Microsoft Visual Studio and JetBrains family of IDE. As the result requirements to performance of SCSA tools becomes critical to provide analysis result as early as possible. Development of SCSA tools based on data flow analysis is quite complex task and quite heavyweight in terms of computational resource utilization. So far it is very important to develop SCSA algorithms as productive as possible.

Commonly used technique to scale SCSA is based on idea function behavior summarization [1]. In paper [2] described analysis of function in first call occurrence, but there is no description of the way how does program source code processed to find this first occurrence. Another approach to process functions in the program in reverse topological order on call graph of the program to consume function behavior summaries on the call sites of functions to avoid repetitive analysis of function call contexts is described in [3]. This task can be solved in different ways. As one of approaches is split analysis to different stages of phases to parse source code into intermediate representation suitable for analysis and build call graph of a program before processing in data flow analysis engine. As far as internal representation in quite large to collect and hold all program properties necessary for analysis frequently it serialized to persistent storage to deserialize and use it in data flow analysis on demand. But any database or disk operation is expensive and affects analysis performance greatly.

In this paper we propose an algorithm to process programs source code in dynamic way hold in memory only necessary working set of program properties needed for precise SCSA, but still scalable to utilize as much CPU resource as possible providing a way to fit analysis working set in RAM.

The paper organized as follows. Section II describes an architecture of SCSA tool. Section III describes dynamic algorithm for source code processing. Section IV highlights experimental results on open-source projects. Section V is devoted to discussion of the proposed solution and highlight future directions of research.

II. ARCHITECTURE OF THE SOLUTION.

In this section we describe the high level architecture of the solution. Let's take a look how the analysis of a single Translation Unit (single source file) is handled (Fig. 1).

The very first step of static analysis is parsing of the source code. Parsing of C++ code itself is quite complex task, so we do not focus our efforts on creating yet another one C/C++ parser. Developed the source code static analysis tool uses LLVM [4] Clang 10.0.1 as a C++ parser, linked as external library.

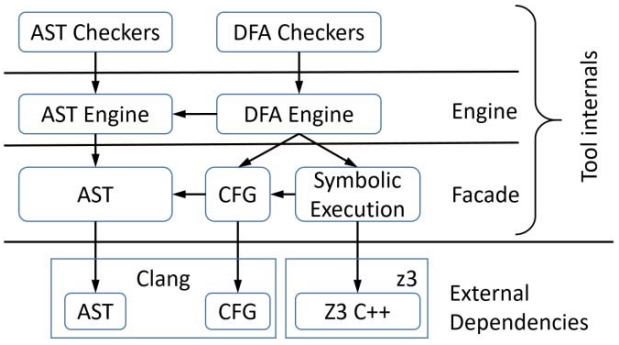

Figure 1. Architecture of solution

But abstract syntax tree (AST) [6, p. 69] provided by Clang is designed for compiling purposes, when we need slightly different AST. At first it should be as tiny as possible, and we also have to be able to delete AST from memory whenever we don't require it anymore. Thus we decided to create our own AST representation. So, we just walk the Clang AST and create our own Nodes based on Clang's ones. This part encapsulated by façade over Clang's AST to have a point

978-1-6654-2327-4/21/$31.00 ©2021 IEEE
DOI 10.1109/IVMEM53963.2021.00016

of disintegration if in future we will need to remove dependency on Clang.

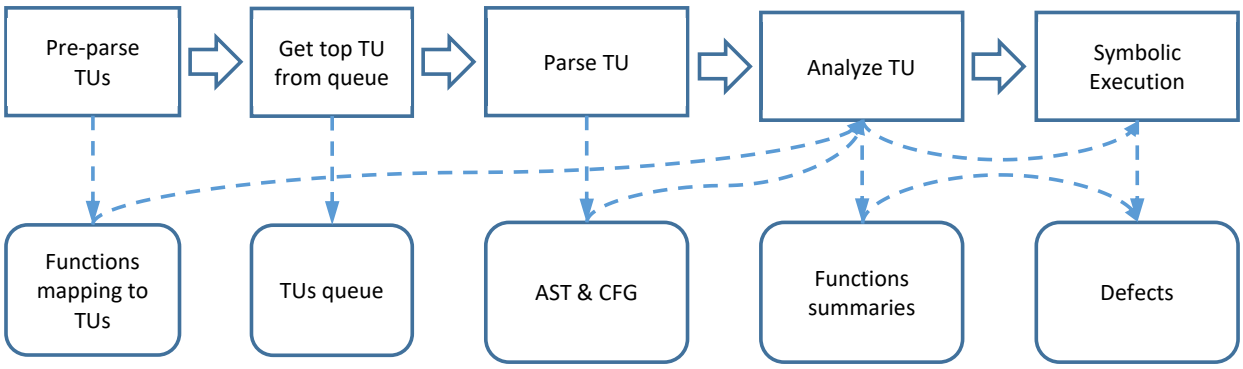

Figure 2. Analysis algorithm

Also this decision has another one, historical, reason. At the very beginning of the project it was not that clear, which technology stack will be approved, so it was decided to create our own AST representation, but not by parsing source code, but transforming it from Clang, or any other parser. And Clang parser can be easily replaced with any other parser, for example GCC [5], without significant efforts.

For the more complex analysis such as data flow or taint analysis and so on, we need also know how data and propagated through the code control flow. Thus for each function we require Control Flow Graph (CFG) [6, p. 401] to be built and stored in translation unit context.

As well as for AST, we build our own CFG by transforming Clang's CFG and link CFG elements with AST nodes. As well as for AST our representation of CFG is built to minimize memory consumption. After all this job is done we can completely remove all Clang's data and free memory.

For the next step – Data Flow Analysis we walk the CFG of the function in reversed order (from end to begin). For each event in program, like read or write variable, pointer dereference, memory allocation and free, or use as divisor, we create an Annotation for the corresponding CFG element. This annotations contains information about the event and element where it arises originally, and the annotation kind used to define how to propagate it through CFG. Some annotations propagated to the declaration of variable, and added to all its usage further, some – added only if arises on all execution branches, some – combine a various propagation rules. Thus for each CFG element we have a list of valid annotations, which sources may be reached from it, with links to theirs sources.

A task to add annotation to state and propagate it through statements are handled by Checkers. There is an interface to define a new annotation kind and setup propagation strategy. To add annotation initially one can use 2 ways – to specify rule when annotation should be added (dereference annotation should be added if dereference operation is faced) – to point out that call of some function add annotation to its return value or arguments (like free adds free annotation to its first argument). Also some specific checkers used to propagate this annotations further in accordance with their propagation strategy. Starting from here the memory consumption is significant, but we still need memory to store intermediate results of analysis. To solve this, functions are analyzed in a call ordered (functions which does not call other functions get analyzed first). Thus, after function is analyzed we do not need to store it anymore, the only data required, is data about function parameters and return value, which takes significantly less space. And both AST and CFG of the function are removed.

III. Dynamic Algorithm of Translation Units Processing.

This section describes the static analysis algorithm implemented in our solution.

As input, the analyzer receives a list of Translation Units (TUs) of C/C++ project and compilation options for each TU. The output of the algorithm is a list of detected problems in the source code.

In general, proposed analysis algorithm is represented on the diagram. (Fig. 2). We tried to keep as best performance as possible that's why we try to keep all necessary for analysis in RAM, but free everything we don't need for analysis anymore. We come from assumption that in real-life programs it is quite rare situation when call graph is spread over a lot of translation units and we need to keep in memory minimal subset of necessary information. We can free all Clang's AST and CFG as far as we get own intermediate representation. At the same time we can free information unnecessary for symbolic execution formulae building after performing data flow analysis with state propagation. In the end we come up to solution when translation units are processed in dynamic way and we can keep only functions behavior in compressed form for later analysis.

A. *Pre-parsing TUs*

The analysis starts with the pre-parsing of all Translation Units (TUs) included to the project. The result of the pre-parsing is a map of functions defined in TUs to TUs itself. So, during the analysis it will be possible to determine in which TU each function is defined. The pre-parsing algorithm is very lightweight since it is not necessary to parse function bodies, only declarations, this phase can be completed very quickly.

B. *Get top TU from queue*

Then all TUs are placed into the queue which is processed one-by-one by the thread pool. The number of threads in the pool is defined depending on the number of available processor cores in the configuration. This allows to achieve the maximum scalability of the analysis via all available CPU cores utilization.

C. Parse TU

At this stage, the source code of TU is compiled by Clang. As a result, we get AST and CFG for all functions and declarations within TU, including the code from included header files. Then received AST and CFG is converted into our own representation and the entire Clang context is released from memory. Our representation of AST and CFG contains only information necessary for analysis and allows us to significantly reduce memory consumption during the analysis.

D. Analyze TU

Then TU is passed for analysis. At the first stage AST-based checkers analyze the TU. After that the TU is passed to the Data Flow Engine.

Data Flow Engine analyzes each function from TU starting from the call graph leaf functions. So, the analysis begins only if the function does not contain calls to other functions or all called functions have already been analyzed. At the end of the analysis we get a summary of the function behavior in the form of:

- annotations for the function declaration (return value, arguments)
- a compressed representation of the function sufficient for its symbolic execution
- a list of potential problems inside the function represented as paths in the data flow graph;
- a compressed representation of the CFG and AST sufficient for its symbolic execution. This representation is a flat data structure with an unfixed size for a single AST element. In average, one AST element takes 4-6 bytes. This allows to keep in RAM all the necessary information for symbolic execution and problem reporting.

AST and CFG of the analyzed function are released from memory when all necessary information gathered.

If the function cannot be analyzed at the moment due to calls to other function(s) pending for analysis then it is placed into the pending list until all called functions are became analyzed. At the same time weight of TU corresponding to called function increases to place lift it up in the TU processing queue based on function-TU mapping retrieved on pre-parsing stage. Thus, the most called functions are processed first in the queue. And after their analysis functions from the pending list will also be analyzed.

This approach involves dynamic re-scheduling of analysis queue and allows to control the size of the pending list and not exceed RAM limitations. According to statistics from real projects, not more than 3% of functions are located in the pending list at the same time. It

E. Symbolic execution

On this stage we check the reachability of potential problems found on the previous stage. Requests to symbolic execution engine are placed into the queue and processed by the thread pool in parallel.

For each request, we create new Z3 [7] context and run symbolic execution from begin of the analyzed function as an entry point. Symbolic execution is performed sequentially for each basic block for compressed CFG representation. During execution, Z3 expressions and their reachability conditions are created. As a result of symbolic execution, we get the formula from the source point to the sink point of the path and check this formula by Z3 solver. A problem is reported only if a positive verdict is received from Z3 solver.

In the end we can split all tasks performed in parallel:

- TU parsing
- Functions data flow analysis with state propagation
- SMT formulae solving.

These three different tasks have different priorities: first – SMT solving, second – function's data flow analysis, third – parsing. Using such kind of parallelization we can utilize as much CPU cores as available for computation. And make sure that we will release resources as early as possible to process remaining TUs.

IV. Experimental evaluation

The quality assurance is performed on the 16 cores CPU, 32Gb RAM with Linux Ubuntu 18 OS.

It is have to be stated that it is quite hard to get modern state of the art source code static analysis tools as far as significant part of them are proprietary and are not available for comparison. That's why we has been forced to evaluate our tool without comparison to state of the art competitors.

The quality assurance performed on the following real-life open-source projects:

Project name	Total size of code	Total line count	Total number of TU
linux-kernel	895 MB	27 275 572	2 617
sqlite	33 MB	955 381	4
zlib	1.5 MB	38 540	17
llvm-project	738 MB	14 828 439	2 554

The result performance statistics is average value of 10 launches on each of projects, and presented in the table below.

Project	Max CPU	Avg. CPU	Avg. RAM	Avg. time (min:sec)	Max. RAM
linux-kernel	66,5%	62,3%	3729.0	04:02.68	4802.50
sqlite	34,6%	28,9%	234.6	01:14.46	3656.16
zlib	30,5%	23,8%	200.0	00:00.62	359.60
llvm-project	90,0%	85,3%	11320.4	16:56.00	15219.30

The number of defects from checkers Null pointer dereference, Use After Free, Uninitialized Memory Usage, Division By Zero checkers are reported on the projects are apresented in the table below.

Checker name	linux-kernel	sqlite	zlib	llvm-project
Division By Zero	53	5	-	27
Null Pointer Dereference	1181	112	1	1073
Uninit Memory Usage	1919	33	-	823
Use After Free	-	-	-	7

V. Conclusion and future works

At first glance it can be found limitations of proposed algorithm. What if there is a very long function call sequence in the program which affects a lot of translation units? In this case the proposed solution has to consume a lot of RAM to hold all translation units intermediate representation AST and CFG with function behavior summaries to provide good

precision of analysis. But experiments show that in real life such projects are rare, most call graph paths pass through a minimal set of translation units and it is possible to hold all necessary information in RAM. So, this assumption can be treated as very rare drawback.

Another case is incremental analysis needed for providing analysis results for a minimal change of the source code (single merge request to code base) as fast as possible. In this case it is necessary to serialize intermediate representation of a program to a persistent storage for the need of reuse it during incremental analysis. The task of efficient intermediate representation serialization and deserialization is one of future research directions.

REFERENCES

[1] M. Sharir, A. Pnueli. Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, ch. 7, pp. 189–233. Prentice-Hall, Englewood CliffM. (1981)

[2] E. Bodden. The secret sauce in efficient and precise static analysis. ISSTA'18 Companion Proceedings for the ISSTA/ECOOP 2018 Workshop, July 2018, pp. 85-93. DOI: 10.1145/3236454.3236500

[3] Бородин А.Е. Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках C и C++. 2016.

[4] C. Lattner, V. Adve LLVM: A compilation framework for lifelong program analysis & transformation. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04) (Palo Alto, California), 2004. DOI: 10.1109/CGO.2004.1281665

[5] B. Gough. An introduction to GCC. Network Theory Limited. 15 Royal Park, Bristol, BS8 3AL, United Kingdom. 2004, ISBN: 0-9541617-9-3

[6] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers. Principles, techniques and tools. Second edition. 2007. ISBN 0-321-48681-1

[7] L. de Moura, N. Bjørner. Z3: An Efficient SMT Solver. International Conference on Tools and Algorithms TACAS'08/ETAPS'08: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems. March, 2008. pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24