International Journal of Soft Computing and Engineering (IJSCE)
ISSN: 2231-2307 (Online), Volume-10 Issue-6, July 2021

# Generation and Visualization of Static Function Call Graph for Large C Codebases

**Sourabh S Badhya, Shobha G**

***Abstract*: As software systems evolve, there is a growing concern on how to manage and maintain a large codebase and fully understand all the modules present in it. Developers spend a significant amount of time analyzing dependencies before making any changes into codebases. Therefore, there is a growing need for applications which can easily make developers comprehend dependencies in large codebases. These applications must be able to analyze large codebases and must have the ability to identify all the dependencies, so that new developers can easily analyze the codebase and start making changes in short periods of time. Static analysis provides a means of analyzing dependencies in large codebases and is an important part of software development lifecycle. Static analysis has been proven to be extremely useful over the years in their ability to comprehend large codebases. Out of the many static analysis methods, this paper focuses on static function call graph (SFCG) which represents dependencies between functions in the form of a graph. This paper illustrates the feasibility of many tools which generate SFCG and locks in on Doxygen which is extremely reliant for large codebases. The paper also discusses the optimizations, issues and its corresponding solutions for Doxygen. Finally, this paper presents a way of representing SFCG which is easier to comprehend for developers.***

***Keywords*: Static function call graph, Static analysis, Duplicate functions, Doxygen, Cytoscape.js**

## I. INTRODUCTION

Large software companies write quality software on a daily basis and track individual components very well. Codebases written in C language have been in use in system development, network development and many applications mainly because C language is easier to interface with machine hardware, consumes less memory and has faster runtimes. It also provides great control to the programmer to create efficient programs.

Over the years, it has been found that large codebases are difficult to analyze due to its sheer scale and complexity involved and the different ways in which files can be linked with each other. There could be several links of a function present in a file with several other functions present within the file or other files which makes it hard for the developer to comprehend these links while making changes to the codebase. There exists a study by T. D. LaToza et al. [1] which indicates ease of comprehending complex codebases using call graphs and it mentions that developers who use visual tools for call graphs were more likely to complete a task much faster than developers who do not use visual tools for call graphs. Hence, there is a need for a visual representation which can be used for easily visualizing all function call dependencies amongst files in the codebases.

The early mention of using Call Graphs for static analysis was done in B. G. Ryder et al. [2] which explicitly defines a call graph as a representation in which "the nodes of the graph are the procedures of the program; each edge represents one or more invocations of a procedure $P_j$ by a procedure $P_i$". The first efforts in this direction were in the Fortran language in which call graphs were represented as a directed acyclic graph (since Fortran 77 is a non-recursive language). However, from then onwards several implementations of call-graphs have arisen in different languages and are still under active research. Most of the implementations make use of the abstract syntax tree (AST) of the underlying language in order to get the call dependencies amongst different functions. However, most of these languages have a single compiler hence making it simple to extract the required information. This is unlike the C language wherein multiple compilers exist and all these compilers were written for some use case such as Intel's C/C++ compiler was written for Intel processors to achieve best runtimes.

There have been several efforts in making developers comfortable with analyzing large C codebases. Compiler specific solutions for static analysis have been present for several years, such as Clang Static Analyzer [3]. However, these tools are compiler specific and since there are many compilers, using any of these tools might result in compatibility problems. Hence the required call graph tool must not be dependent on a specific compiler. This paper discusses the usage of Doxygen [4] as one such tool which is scalable, almost accurate and compiler independent. It further discusses how the functions and function calls can be visualized by using Cytoscape.js [5] which is a state-of-the-art graph theory visualization JavaScript framework to enable a better viewing experience for developers.

## II. RELATED WORK

Several implementations have been developed in this field in order to accurately represent function call graphs. The early formal research into the extraction of function calls was achieved by D. Callahan et al. [6] which proposes an algorithm to extract function calls from Fortran 8x language which handles recursion and has a time complexity

Manuscript received on June 05, 2021.
Revised Manuscript received on June 12, 2021.
Manuscript published on July 30, 2021.
* Correspondence Author
**Sourabh S Badhya***, Department of Computer Science, R.V. College of Engineering, Bengaluru (Karnataka), India. Email: sourabhsbadhya.cs17@rvce.edu.in
**Shobha G**, Professor, Department of Computer Science, R.V. College of Engineering, Bengaluru (Karnataka), India. Email: shobhag@rvce.edu.in

© The Authors. Published by Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP). This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

dependent exponentially on the number of procedures. It also asserts that polynomial time complexity can be achieved by setting an upper bound on the number of procedures. This was one of the earliest algorithms which could identify both normal function calls as well as recursion correctly, which is used in modern programming languages.

A more rigorous overview of call-graph as a graph theory problem was done in T. Reps et al. [7] whose focus was to find solutions to a large set of interprocedural dataflow problems in polynomial time. He also proposes that instead of calculating the worst-time complexity of each algorithm, it is better to bound the total cost of the operations performed at each aspect.

An in-depth dissertation on the effect of function pointers on call graphs was given in G. Antoniol et al. [8] wherein it proposes an algorithm to identify function pointers called the 'points-to' algorithm. It also provides a quantitative evaluation of function pointers and the key role it plays in call graph construction.

An algorithm to specifically identify virtual function calls as well as interfaces was also proposed in X. Zhuo et al [9]. wherein it makes use of type flow analysis to get the call dependencies. This algorithm also takes less time and space usage when compared to points-to algorithm. However, this algorithm only applies to object-oriented languages.

An elaborate framework for call graph construction was proposed by D. Grave et al. [10]-[11]. These papers propose a general parameterized algorithm which provides a detailed vocabulary for depicting call graph algorithms, illustrates the differences and similarities of different algorithms and investigates the design space of call graph algorithms. It also assesses call graph algorithms with respect to an optimizing compiler (Vortex compiler) and the algorithms can be applied to any functional language.

An alternative method for call-graph construction was proposed in Y. Terashima et al. [12] wherein a tool by the name 'dcgg' was proposed which made use of DWARF2 debugging information. This paper illustrates a method which combines both binary analysis and debugging information in order to extract function call dependencies. This method could extract inline functions in C code as well virtual function calls in C++ apart from the default functions present in code.

One of the earliest tools in order to generate call graphs is described in G. Antoniol et al. [13] by the name 'XOgastan'. It was developed as a static analysis tool which makes use of gcc/g++ compiler. It exploits the internal representation of gcc/g++ compiler which is the abstract syntax tree and translates it into graph exchange language representation. The final output is in the XML format which can be easily parsed with XML parsers. However, since it is compiler dependent it cannot be used for codebases which are not built on gcc/g++ compiler.

A framework was also proposed in H. Hoogendorp [14] which describes the complete steps right from data extraction to visualization of call graphs. This was one of the first frameworks which accounted for scalability of the system. It also explains about the different ways in which visualization can be done. The problem with this framework is that compiler wrapping must be done for every compiler. This makes the system compiler dependent.

Another alternative method for call graph generation was done in F. Zhang et al. [15] which proposes a static analysis method that analyses the LLVM IR (Internal representation) generated by compilation of source programs. This method was used in order to analyze the parent-child relationship in between threads which are created using the pthread library. Compiler dependency of LLVM is the drawback of this method. A recent and similar tool which was developed for static analysis which is described in P. D. Schubert et al. [16] by the name 'Phasar'. Phasar is built as an extension to the LLVM compiler infrastructure. The analysis is done on LLVM IR, since solving data-flow problems on IR is easier than source code itself. However, this makes the implementation compiler dependent in nature.

Another tool for static analysis of source code was proposed in M. L. Collard et al. [17] which is named as srcML. It is a highly scalable tool and robust tool for source code analysis. It converts source code into XML format by making use of Clang AST, hence making the tool compiler dependent. A similar approach for C/C++ source code is also stated in A. M. Bogar et al. [18] in their implementation MLSA wherein it is designed to provide support for multiple languages. MLSA is lightweight, scalable and is written as an island grammar (a technique used to support multiple languages). The C/C++ code is parsed using Clang AST which is again compiler dependent.

From all the papers featured above, we see that most of the tools are compiler dependent. Some tools are not scalable and hence do not work for large codebases. In order to address these issues, we are making use of Doxygen which is an effort to document large codebases. However, the parser that is used in Doxygen can be exploited to extract function call dependencies. The proposed system also discusses a unique way of representing function calls using Cytoscape.js which is interactive, user friendly and scalable by the number of nodes.

## III. PROPOSED SYSTEM

The proposed system makes use of Doxygen as its tool for static analysis. It can be broken down into 2 phases - Preprocessing phase and visualization phase.

### A. Preprocessing phase

This phase mainly involves extraction of call dependencies by using Doxygen. Doxygen requires a config file called Doxyfile which specifies the input files that must be analyzed, along with the output format that should be generated. Doxygen supports various output formats like HTML, XML, Latex, SQLite etc. In this system, we will use SQLite [19] database since it is lightweight, reliable and easily portable. The steps involved in preprocessing phase are as follows –

- **Identification of C files that are used during build / compilation** - The C files which are used to create final executables / images are being used as input to the Doxyfile. The identification can be either as simple as looking into all C files in the codebase or looking into the Make files / past execution logs and deriving information about the C files. Once the C files are identified which are used for a specific build, it is inserted into the Doxyfile in the INPUT tag.


© Copyright: All rights reserved.

```
INPUT= main1.c main2.c … mainN.c
EXTRACT_ALL = YES
EXTRACT_PRIVATE = YES
EXTRACT_STATIC = YES
REFERENCES_RELATION = YES
REFERENCED_BY_RELATION = YES
LOOKUP_CACHE_SIZE = 2
GENERATE_XML = NO
GENERATE_HTML = NO
GENERATE_LATEX = NO
GENERATE_SQLITE3 = YES
SQLITE3_OUTPUT = sqlite
SQLITE3_RECREATE_DB = YES
```

**Fig. 1. A sample Doxyfile**

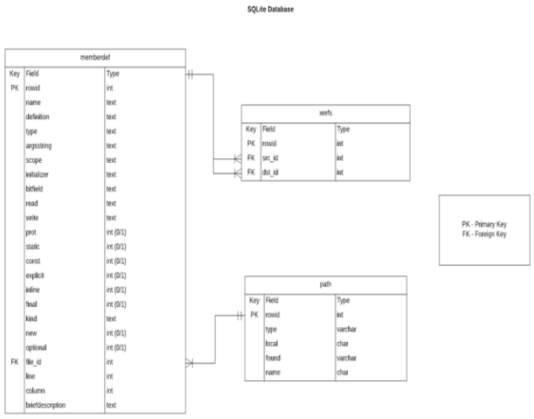


**Fig. 2. Doxygen Sqlite Database**

- **Use Doxygen to extract call dependencies** - Run Doxygen by using the Doxyfile. A sample Doxyfile would have the following tags as specified in Fig. 1. Some additional tags such as LOOKUP_CACHE_SIZE, NUM_PROC_THREADS can be set for faster execution time for very large codebases. This will be discussed in detail in the experimental results section.
- **Create support for duplicate functions** - Duplicate functions are functions with the same name and same arguments but are implemented more than once. This is usually found in codebases wherein a functionality has several implementations and all of them are equally important. However, they are compiled separately in different images. These functions are usually seen in networking codebases wherein a single functionality has multiple implementations and depending upon the use case, one function is used over another. However, such dependencies are not handled in static analysis tools and hence support needs to be created. This can be done as follows -
    - Identify all the functions which have the same name, return type and arguments.
    - Get all the caller dependencies of all the duplicate functions involved.
    - Insert a link between the caller dependencies with all the duplicate functions.

  In other words, the caller dependencies are shared amongst all the duplicate functions. This will make sure that the call graph generated will reflect all the dependencies of the duplicate functions. The data extracted by Doxygen and the SQLite database output is described in Fig. 2. Similarly, other fields are defined which indicate the properties of the member indexed. The table 'xrefs' has the information pertaining to call dependencies. The fields 'src_rowid' and 'dst_rowid' are foreign keys to 'memberdef' table's 'rowid'. A record in 'xrefs' table signifies that there is a call dependency from member with ID 'src_rowid' to member with ID 'dst_rowid' provided that both these members are functions.

This is a unique way in which nodes and edges are stored and this can be easily used in the visualization phase. As discussed earlier, the links to the duplicate functions are inserted to the 'xrefs' table and no other table is modified. The 'path' table has information about the location of the members indexed in 'memberdef'. In summary, the entire preprocessing phase can be shown in Fig. 3.

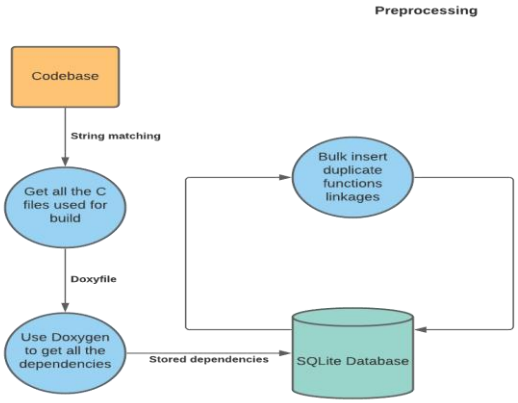


**Fig. 3. Preprocessing phase**

## B. Visualization phase

The visualization phase involves using the SQLite database in a Node.js application. Cytoscape.js is used for graph visualization since it is highly optimized, scalable and user-friendly. The function call graph is represented by making use of a compound graph wherein the file in which the function is present is symbolized as the parent node and function present in the file is represented as the child node. This creates a sense of inclusivity of functions within files and is easily understandable to the developer.

The caller and callee edges are present in between different functions and no edges are present in between files. Cytoscape.js expects JSON formatted data in order to include edges and nodes. Each node and edge are uniquely identified by its ID, hence this ID is set to the row ID in 'memberdef' table for nodes and row ID of 'xrefs' table for edges to make it unique. In order to facilitate the usage of compound graph, Cola extension [20] is also used along with Cytoscape.js. It is also used to preassign the position of nodes, without manual setting of positions.


© Copyright: All rights reserved.

Every node and edge are randomly positioned with a predefined distance amongst them in order to avoid overlapping.

Interactive functionalities can be implemented by creating buttons on the nodes of the graph. Since Cytoscape.js uses canvas element to plot the nodes and edges of a graph, all these buttons must be plotted on top of the canvas. The buttons are created using the context-menus extension [21].

The architecture of the visualization phase can be shown in Fig. 4. When an input such as a function name and file name is taken from the sidebar window, appropriate results from the SQLite database are extracted and given to the webview which uses Cytoscape.js and plots the nodes and directed edges on the graph. All the information transfer between the extension backend and the webview happens in the form of messages sent from application backend to webview and vice-versa.

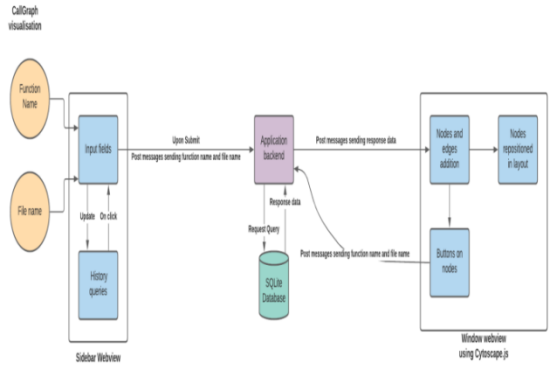


**Fig. 4. Visualization phase**

## IV. EXPERIMENT RESULTS

The experiments were conducted in a RHEL 8 machine with system specifications of 8 cores, 8 GB RAM, 512 GB hard disk. Several validations were done in order to identify duplicate functions correctly. The source of errors was seen in unused function pointers. However, unused function pointers are usually considered a bad practice, but they are valid. An example for duplicate function is shown in Fig. 5. The graph represents a small codebase with 3 files – main.c, a.c and b.c. The file main.c has the driver function 'main' which calls a function called 'type1' which is present in both a.c and b.c. The definition of 'type1' is different in both files and hence they call different functions. However, when the code is compiled, only one file is passed into compilation thereby having only one definition of 'type1' during compilation.

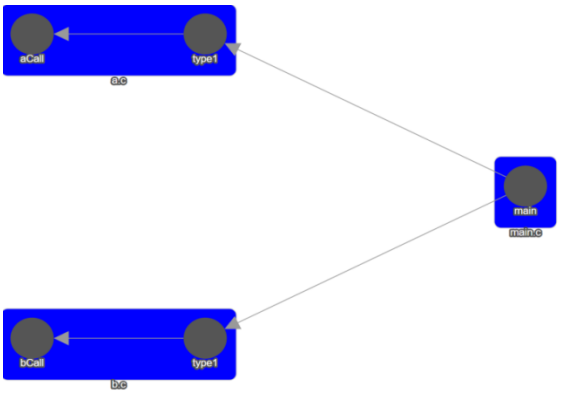


**Fig. 5. Example of duplicate functions**

**Table - I: Comparison of codebases on different metrics**

| Codebases | Size of codebase (kLOC) | Time taken to analyze (s) | Output SQLite database (MB) |
|---|---|---|---|
| Redis | 181.118 | 9.861 | 4.4 |
| OpenSSL | 522.121 | 34.607 | 14 |
| PostgreSQL | 1469.177 | 62.677 | 25 |
| Linux | 20550.393 | 10255.646 | 732 |

A comparison was done amongst 4 codebases - Redis [22], OpenSSL [23], PostgreSQL [24] and Linux [25]. Table. I illustrate the results obtained when Doxygen was run on these codebases. This table indicates that smaller codebases having smaller number of functions generate less amount of data when compared to its larger counterpart which is the obvious norm. It also indicates that if function call dependencies are more in a codebase, then a lot of data is generated, hence taking more time. This is especially seen in Linux codebase since the order at which memory increases is higher than order at which size of the codebases increases. Higher variable usage is also a potential reason for large database output as well since Doxygen generates data for variables as well.

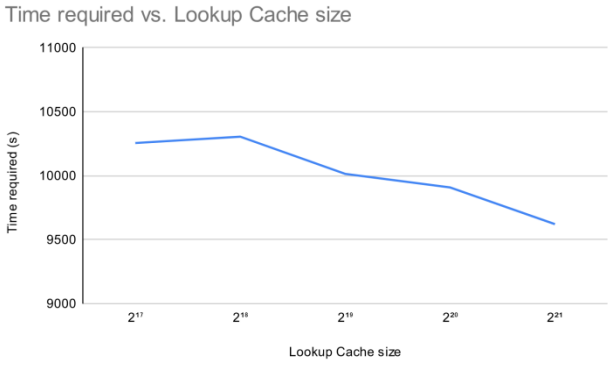


**Fig. 6. Time required v/s. Lookup cache size**

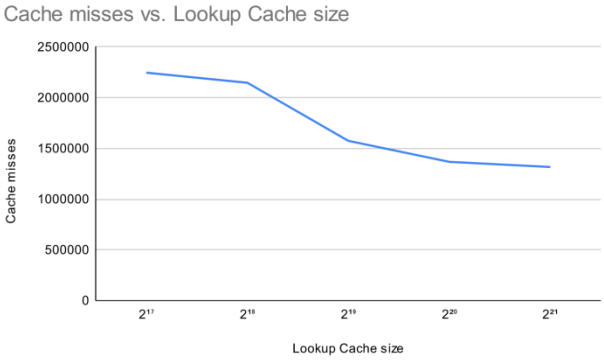


**Fig. 7. Cache misses v/s. Lookup cache size**

Optimization on execution performance was also considered. Doxygen was not originally coded for providing the best runtime behavior, however efforts were made to bring in efficiency. Doxygen makes use of a symbol lookup cache while parsing the code whose sizes can be manipulated to bring better performance. It is controlled by the following tag - LOOKUP_CACHE_SIZE.

*Retrieval Number: 100.1/ijsce.F35070510521*
*DOI: 10.35940/ijsce.F3507.0710621*
*Journal Website: www.ijsce.org*



*Published By:*
*Blue Eyes Intelligence Engineering and Sciences Publication*
*© Copyright: All rights reserved.*

The values of this tag ranges from 1 to 9, which indicates that the size of the symbol lookup cache ranges from $2^{17}$ to $2^{25}$ symbol storage. Given the codebase, an optimal number needs to be set to decrease execution time as well as not use excessive memory. The effect of lookup cache size is not considerable for small codebases. The graph in Fig. 6 indicates how execution time decreases with increase in lookup cache size for Linux codebase. After the size of $2^{21}$ (when LOOKUP_CACHE_SIZE = 5), the number of symbols remains constant and further increase in size is wasteful. Similar experiment was also conducted on the symbol lookup cache misses with the lookup cache size for Linux codebase whose graph is indicated in Fig. 7. Both graphs indicate that optimal size for symbol lookup cache must be reached to achieve the best execution time.

The graphs generated by using Cytoscape.js are represented in the form of compound graphs by using Cola extension. One such graph is represented in Fig. 8. It is observed that as the complexity of the graph and number of nodes increases, the time required to render the graph also increases since there is automatic positioning of nodes. Time required to render the graph also increases with increase in the number of extensions used. Adding buttons on top of nodes by using the context-menus extension slows down the rendering of large graphs. However, if the number of nodes is 300-500 which is normally the number of nodes viewed by developers, this does not affect performance significantly.

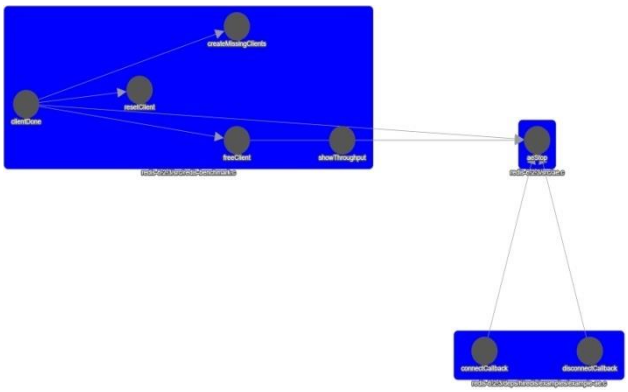


**Fig. 8. Call graph generated for a small component of Redis**

## V. CONCLUSION

This paper illustrates how Doxygen can be utilized to generate graphs which are scalable and efficient. It highlights the effectiveness of using compiler-independent tools for generating static function call graphs and focuses on visual representation which is dynamic and compatible with the web. It also provides a way of identifying duplicate functions. Additionally, it also explains the use of optimizations of Doxygen and compares it with several codebases.

Possible improvements such as execution performance and better parsing methods can be used in order to bring down errors. Further improvements such as real-time syncing with the codebases must be explored which can make the tool more effective for developers.

## REFERENCES

1. T. D. LaToza, B. A. Myers, “Visualizing call graphs” in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 117-124.
2. B. G. Ryder, “Constructing the Call Graph of a Program” in *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, May 1979, pp. 216-226.
3. Clang Static Analyzer, Available: https://clang-analyzer.llvm.org/
4. D. V. Heesch, Doxygen. Available: https://www.doxygen.nl/index.html
5. M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, G. D. Bader. ‘Cytoscape.js: a graph theory library for visualisation and analysis.’ *Bioinformatics*. Volume 32, Issue 2, 15 January 2016, pp. 309-311.
6. D. Callahan, A. Carle, M. Wolcott Hall, K. Kennedy, “Constructing the Procedure Call Multigraph” in *IEEE Transactions on Software Engineering*, Volume 16, Issue 4, April 1990, pp. 483-487.
7. T. Reps, S. Horwitz, M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995, pp. 49-61.
8. G. Antoniol, F. Calzolari, P. Tonella, "Impact of function pointers on the call graph" in *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, 1999, pp. 51-59.
9. X. Zhuo, C. Zhang, “A Relational Static Semantics for Call Graph Construction” in *Formal Methods and Software Engineering, ICFEM 2019, Lecture Notes in Computer Science*, Springer, Cham, Volume 11852, 2019, pp. 322-335.
10. D. Grove, C. Chambers, “A framework for call graph construction algorithms” in *ACM Transactions on Programming Languages and Systems*, Volume 23, Issue 6, November 2011, pp. 685-746.
11. D. Grove, G. DeFouw, J. Dean, C. Chambers, “Call graph construction in object-oriented languages” in *ACM SIGPLAN Notices*, Volume 32, Issue 10, October 1997, pp. 108-124.
12. Y. Terashima, K. Gondow, “Static Call Graph Generator for C++ using Debugging Information” in *14th Asia-Pacific Software Engineering Conference (APSEC'07)*, 2007, pp. 127-134.
13. G. Antoniol, M. Di Penta, G. Masone, U. Villano, "XOgastan: XML-oriented gcc AST analysis and transformations" in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, 2003, pp. 173-182.
14. H. Hoogendorp, “Extraction and visual exploration of call graphs for Large Software Systems” in *Master’s thesis, University of Groningen*, 2010.
15. F. Zhang, N. Gu, J. Su, “A Static Call Graph Construction Method Based on Simulation Execution” in *IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, 2019, pp. 72-76.
16. P. D. Schubert, B. Hermann, E. Bodden, “PhASAR: An Inter-procedural Static Analysis Framework for C/C++” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019, pp. 393-410.
17. M. L. Collard, J. I. Maletic, “srcML 1.0: Explore, Analyze, and Manipulate Source Code” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 649-649.
18. A. M. Bogar, D. Lyons, D. Baird, “Lightweight Call-Graph Construction for Multilingual Software Analysis”, *13th International Conference on Software Technologies (ICSoft)*, July 2018, pp. 362-371.
19. D. R. Hipp, SQLite, Available: https://www.sqlite.org/index.html.
20. Cytoscape-cola extension. Available: https://github.com/cytoscape.js-cola
21. U. Dogrusoz, A. Karacelik, I. Safarli, H. Balci, L. Dervishi, and M.C. Siper, “Efficient methods and readily customizable libraries for managing complexity of large networks” in *PLoS ONE*, 2018.
22. Redis source code. Accessed on 24th May 2021. Link: https://download.redis.io/releases/redis-6.2.3.tar.gz
23. PostgreSQL source code. Accessed on 24th Many 2021 Link: https://ftp.postgresql.org/pub/source/v13.3/postgresql-13.3.tar.gz
24. OpenSSL source code. Accessed on 24th May 2021. Link: https://www.openssl.org/source/openssl-3.0.0-alpha17.tar.gz
25. Linux kernel source code. Accessed on 24th May 2021. Link: https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.12.7.tar.xz


*© Copyright: All rights reserved.*

## AUTHORS PROFILE

**Sourabh S Badhya,** is an undergraduate student who is pursuing Computer Science & Engineering in R.V. College of Engineering, Bengaluru. His area of interests lies in Machine Learning and Natural Language Processing. He has 3 publications in his name in the Natural Language Processing domain.

**Dr. Shobha G,** is a professor in R. V. College of Engineering, Bengaluru. Her area of interests lies in data mining, image processing and networking. She has a teaching experience of 25 years and research experience of 14 years. She has 145 publications in international journals and conferences. She has guided 30 UG projects, 40 PG projects and 7 Ph.D. students. She also has 4 patents and has been a reviewer of several books.


*© Copyright: All rights reserved.*