Source Code Analysis: Current and Future Trends & Challenges

Alexey Ponomarev[1,a], Hitesh S. Nalamwar[1,b], Ragesh Jaiswal[2,c]

[1]National Research Tomsk Polytechnic University, 30 Lenin Avenue, Tomsk, 634050, Russia

[2]Indian Institute of Technology Delhi, Hauz Khas, New Delhi, Delhi 110016, India

[a]aaponomarev@tpu.ru, [b]Hitesh@tpu.ru, [c]rjaiswal@cse.iitd.ac.in

Keywords: source code (SRC) analysis, software maintenance, reverse engineering, applications of SRC analysis, challenges in SRC analysis, tentative trends in SRC analysis.

Abstract. The increasing complexity of software systems is making source code analysis a more economic option to automate the identification of defects, vulnerabilities and inefficiencies. This paper initially outlines the general anatomy of automatic source code analyzers, dimensions of analysis that can be performed with today's state-of-the-art tools, various limitations of automatic source code analysis in the areas of programming language coverage, quantity of false positive claims, system architecture breakdowns and code/time complexity. The paper is concluded by presenting future tentative trends of source code analysis.

Introduction

As software systems evolve, their size and complexity grows. Nowadays the software development and maintenance efforts driving this evolution is mostly a man-powered codding, reviewing and troubleshooting effort, and therefore, an error-prone and costly process. As a consequence, better analysis and visualization of software is demanded to aid in the system comprehension. Thus, this paper describes the process of automatically extracting program information from source code or binary artifacts [1] to comprehend the behavior and functionality of software in order to identify defects, vulnerabilities and inefficiencies.

Background

The high-level overview to the concept of SRC analysis, and what tools can aid SRC analysis, and to what software engineering tasks it can be applied are given below.

Phases of SRC analysis. The processes involved in SRC analysis commonly used by SRC analyzers can be broken down into three major phases as follows:

Parsing phase is the process of analyzing a text, made of a sequence of tokens to determine its grammatical structure with respect to a given formal grammar. The output from the parser is used for deeper analysis of the source code.

Internal representation involves abstracting particular details from the SRC parse tree into various internal representations, i.e control flow graphs, trace-flows, call graphs, or abstract syntax trees for automated analysis.

Analysis phase involves the actual analysis of the various internal representations in the previous step. Analysis of SRC is obtained purely through the use of rigorous mathematical methods. Such mathematical techniques used include abstract interpretation, lexigraphically, and denotational, axiomatic and operational semantics.

State-of-the-art industry tools. The most of SRC analyzers tools available on the market perform some form of static or dynamic automated analysis; some are capable of accomplishing both. The set of detectable issues by various SRC analyzer tools marked with (●) is presented in Table 1. The results have been derived from experimenting and past experience with the listed tools.

Table 1. List of available functions for SRC analyzers

Detectable Issue	**Avalanche**	**Clone Doctor**	**Coverity**	**Fortify**	**Klocwork**	**Lint**	**Valgrind**
Arithmetic Error Overflow, division by zero	●			●	●	●	
Attack Vulnerability Code/file/command-line/SQL injection			●	●	●		
Buffer Overflow Array out of bounds, stack/heap corruption	●		●	●	●		●
Code Redundancy Unused calculation, duplicate code		●			●		
Dead Code Non executed function/ branch/loop, unused variable/ argument/return value			●	●	●	●	
Poor Design Practice Hard code credentials, use invalidated data, details leak.					●		
Race Condition Deadlock, time-to-check vs. time-to-use a value			●		●		●
Resource Leak Memory, file/socket handle			●	●	●		●
Type Mismatch Sign/unsigned conversion, small /big-endian encoding mismatch, function prototype mismatch				●	●	●	
Undefined Usage Uninitialized variable, null pointer	●		●	●	●	●	●

dereference, use after free, double free							

Applications of SRC analysis

What is SRC analysis used for? What is the need behind the use and what are different ways of achieving a certain need? These questions are answered in this section.

Architectural recovery. It plays an important role in understanding the system and therefore maintaining it. Software architecture of a program or a computer system is a representation of system elements and the relationships between them. The recovery of such artifacts deals with the ways to reproduce the main decisions made by experts during the design phase of the system. [2] Moreover, analyzing source code helps to understand what the code does; this helps to improve the overall comprehension of program functionality.

Clone detection. SRC analysis techniques are used for identifying similar section of code that shares the same behavior. Some of these techniques are dynamic programming, data mining, program dependence graphs, and execution traces. [3]

Debugging. Debugging is the activity of finding bugs and reducing the number of bugs in a software system. Debugging is the hardest activity in software engineering [4]. Some recent debugging techniques include algorithmic debugging, delta debugging, and statistical debugging.

Fault location. A lot of tools are currently used to locate faults. They can be classified as knowledge-based, i.e. where the system itself localizes the faults through interpreting the generated information and non-knowledge-based, i.e. where the user has to guide the system through inputs and data. [4]

Reverse engineering. Reverse Engineering does not mean architectural recovery. The process of reproducing all the artifacts from the existing ones is reversed engineering. Decompilers are an example of reverse engineering, where the compiler recompiles machine language to be understandable to the programmers. [5]

Code security. During implementation phase of software life cycle, programmers insert many fraudulent code-blocks. [7] Software managers cannot go through the code every time it is changed. Tools may help reduce or even overcome this problem.

Software maintenance. The concept of changing software system after delivering is called software maintenance. Debugging, which is mentioned above, is part of this process. SRC analysis is used to locate the bug in the source code. [7]

Challenges and tentative solutions

Variation in programming paradigms. Most languages that are being used currently are composed of segments that are written in different programming languages. A lot of new concepts have also been implemented in these languages, e.g. dynamic class loading, pointer arithmetic, data types and exception handling are making parsing a difficult task. Better tools with higher and more precise analysis are needed that can handle various problems related to the language issues. Tools that can handle the various problems like pointer analysis, different languages, etc., will provide more flexibility at runtime, and a more powerful analysis. Also there is a necessity of meta-model that can capture the needs of a certain programming language, new techniques for parsing, syntax mapping and semantic analysis to improve SRC analysis. [8]

Tools that cry wolf. An annoying drawback of various SRC analyzers is that they report excessive false-positives. This is particularly true for static analysis since the discovery of defects is depending on a limited number of abstract representations and not on full SRC context. This drawback limits the automation for these tools since developer intervention is required to audit these claims. Furthermore, SRC analyzers tend to report the same real defect multiple times as different issue categories. Fixing the real defect resolves multiple reported issues. This makes it difficult for managers to access the actual code quality.

To compensate for the excessive number of false-positives, the reported issues should be managed as part of the software development process. The developer should audit all reported issues by source code analyzer tools. If deemed legit, a fix request should be opened. If the issue is indeed considered a false positive, then the SRC analyzer should be calibrated [9] to suppress this instance of the issue or the false-positive issue should be documented as not a real defect.

Program architecture. Nowadays, software programs are designed and developed with more than one programming language. Plug-ins can be written in different programming language for a particular piece of software program. This is a particular challenge to source code analysis with large systems that contain different languages. A solution to the above challenge is the construction of common models automatically once we have specified mappings of the language specific meta-models of concrete front-ends to the common meta-model. The common meta-model captures program information in language independent of representation. [9,2]

Real time analysis. Real time analysis may take place in compile time and run time. There is no real time analysis during compilation of the code, though it exists in some integrated development environments in a limited way. The challenge is to have full real time analysis during the compilations of the code. Self-healing program is type of run time analysis where the program fixes itself when an internal error takes place in a data structure. The healing process happens in real time during the program execution. Even though there is no specfic definition of self-healing system, the idea is still growing and research is being conducted. [9]

Recent algorithms are based on a single core processor and limited memory, which may cause running out of memory before running out of time. [9] We would suggest that for dynamic analysis, distributed architecture could be used to achieve multiprocessor system. It may consume lots of resources but it will do the job in less time than central system. In this case we should have the same code to be analysed in each node in the distributed system and every time change takes place in one it should take place in all of the others in the system. In this case it is advisable to combine a continuous integration procedures at the end of the development phase, analysis of the code could be part of that phase.

Conclusions

The use of sophisticated SRC analysis capable of automatically identifying defects, vulnerabilities and inefficiencies can certainly enhance the quality of software systems and reduce maintenance costs. A wide range of analysis can be performed on various programming languages. All of which involve the phases of parsing the SRC to create call-graphs, syntax trees, execution-traces and control-flow for abstract representations to be lexicographically, denotationally, axiomatically or operational semantically analyzed. This analysis can detect various types of arithmetic, buffer overflows, code redundancy, race conditions, resource leaks, type mismatches and undefined usage issues. All of which are helpful in architectural recovery, debugging, fault location and reverse-engineering tasks. Some limitations of SRC analysis include the amount of false-positives they report as well as the time/memory resource intensiveness required. As the field of SRC analysis matures, these factors should be kept in mind to successfully incorporate it in the software development process.

References

[1] D. Binkley, Source Code Analysis: A Road Map, Future of Software Engineering, Minneapolis MN, 2007(104 – 119).

[2] Chung-Horng Lung, Agile software architecture recovery through existing solutions and design patterns, Proc. of 6th Int'l Conf. on Software Engineering and Applications (SEA), Boston, MA, Nov. 2002(539–545).

[3] Ghulam Rasool, and Nadim Asif, Software architecture recovery, World Academy of Science, Control, Quantum and Information Engineering 2007(4/34).

[4] James S. Collofello, Larry Cousins, Towards automatic software fault location through decision-to-decision path analysis, Proceedings of the National Computer Conference1987 (539).

[5] Mathew Schwartz, How to Reverse Engineering, 2011. Information on http://www.computerworld.com/s/article/65532/Reverse_Engineering

[6] Gert van der Merwe, Jan H.P. Eloff., Software source code, visual risk analysis: an example, Department of CS, Rand Afrikaans University, Johannesburg 2006, South Africa. 1998(17(3)/233-252).

[7] Penny Grubb, Armstrong A. Takang, Software Maintenance: Concepts and Practice, World Scientific Publishing Company, 2nd edition, September 2003.

[8] D. Cruz, P.R. Henriques and J.S. Pinto, Code analysis: past and present, Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009), University of Minhom.

[9] A. Chou, False positives over time: a problem in deploying static analysis tools coverity. Information on http://www.cs.umd.edu/~pugh/BugWorkshop05/papers/34-chou.pdf