Static Analysis: A Survey of Techniques and Tools

Anjana Gosain and Ganga Sharma

Abstract Static program analysis has shown tremendous surge from basic compiler optimization technique to becoming a major role player in correctness and verification of software. Because of its rich theoretical background, static analysis is in a good position to help produce quality software. This paper provides an overview of the existing static analysis techniques and tools. Further, it gives a critique of static analysis approach over six attributes, namely precision, efficiency, coverage, modularity, scalability, and automation.

Keywords Static analysis · Data flow analysis · Abstract interpretation · Constraint solving · Symbolic execution · Theorem proving

1 Introduction

Static analysis is the process of automatically analyzing the behavior of computer programs without executing it [1–3]. Dynamic analysis, on the other hand, analyses programs by executing an instrumented program and generating some form of trace [4]. Static analysis techniques have remained the most popular choice for analysis of computer programs because of the fact that it is very simple and fast [5]. Dynamic analysis has existed in parallel with static analysis providing a complementary alternative to static analysis [4, 6, 7]. This paper is an attempt to provide an overview of static analysis and its associated techniques and tools. The remaining paper is organized as follows: Sect. 2 and its subsections give brief introduction of static program analysis, its taxonomy, and associated techniques and tools.

A. Gosain (✉) · G. Sharma
USICT, Guru Gobind Singh Indraprastha University, New Delhi 110078, India
e-mail: anjana_gosain@hotmail.com

G. Sharma
e-mail: ganga.negi@gmail.com

© Springer India 2015
D. Mandal et al. (eds.), *Intelligent Computing and Applications*,
Advances in Intelligent Systems and Computing 343,
DOI 10.1007/978-81-322-2268-2_59

Section 3 gives a critique of static analysis, namely precision, efficiency, coverage, modularity, scalability, and automation. Finally, Sect. 4 concludes the paper.

2 Static Analysis

Static analysis means automatic methods which can reason about run-time properties of program without actually executing it [2, 8]. Static analysis has been used to detect errors which might lead to premature termination or ill-defined results of the program [9]. Following is a non-exhaustive list of errors that can be addressed using static analysis [3, 9, 10]:

- array out of bound, i.e., accessing an element of array beyond its size.
- null pointer dereference, i.e., a pointer with NULL value is used as though it contained a valid memory address.
- memory leaks, i.e., a program fails to return memory taken for temporary use.
- buffer underflow/overflow, i.e., size of a data value/memory used for holding is not taken care of.
- access to uninitialized variables/pointers, i.e., use of variables/pointers without assigning them values/memory.
- invalid arithmetic operation, e.g., division by zero.
- non-terminating loops, e.g., exit condition of the loop does not evaluate to false.
- Non-terminating calls, i.e., the control flow of a program never returns from a function call.

Most of the aforementioned errors can be detected by software testing. Although testing still remains the major validation activity in software development process [8], the promising nature of program analysis approaches can help software developers in producing correct and reliable software. The main advantages and disadvantages of static analysis over testing are summarized in Table 1.

One of the key ideas behind static analysis is abstraction. Abstraction refers to transformation of a program, called concrete program, into another program that still has some key properties of the concrete program, but is much simpler, and therefore easier to analyze [5]. Over-approximation results when there are more behaviors in the abstraction than in the concrete system [5]. Under-approximation, on the other hand, deals with fewer behaviors than in the concrete system.

Static analysis can be sound or unsound. Soundness guarantees that the information computed by the analysis holds for all program executions, whereas unsound analysis does not [11]. Static analysis can be made path, flow, and context sensitive by making it to distinguish between paths, order of execution of statements, and method call (call site), respectively [12]. Precision of an analysis approach can be measured by classifying the analysis results into three categories [13]: false positives, i.e., nonexistent bugs are reported as potential bugs; false negatives, i.e., bugs that are undiscovered; and true positives, i.e., true and discovered bug. Efficiency is related to computational complexity or cost pertaining to

Table 1 Comparison of static analysis and testing

Static analysis	Testing
Can be applied without executing the code	Can be applied only by executing the code
Can be applied early in the development process	Is applied late in the development process
Results do not depend on inputs	Results depend on inputs
Results can be generalized for future executions	Results cannot be generalized for future executions
Less costly	Very costly
Very fast process	Slow process
False-positive rate is very high	False-positive rate is very less
Approximations are used	Exact results are used
Cannot be used for functional correctness of program	Can be used for functional correctness of programs

space and time requirements of the algorithms used in the analysis [5]. Precision and efficiency are related to each other, i.e., a precise analysis is more costly and vice versa.

2.1 *Static Analysis Techniques*

2.1.1 Syntactic Pattern Matching

This technique is based on the syntactic analysis of a program by a parser. The parser takes as input the program source code and outputs a data structure called abstract syntax tree [1]. When used for bug finding, this technique works by defining a set of program constructs that are potentially dangerous or invalid and then searching the input program's abstract syntax tree for instances of any of these constructs. Syntactic pattern matching is the fastest and easiest technique for static analysis. But it provides little confidence in program correctness and can result in a high number of false alarms [14].

2.1.2 Data Flow Analysis

It is the most popular static analysis technique. The general idea was given in [15–17]. It proceeds by constructing a graph-based representation of the program, called control flow graph, and writing data flow equations for each node of the graph. These equations are then repeatedly solved to calculate output from input at each node locally until the system of equations stabilizes or reaches a fixed point. The major data flow analyses used are reaching definitions (i.e., most recent assignment to a variable), live variable analysis (i.e., elimination of unused assignments), available expression analysis (i.e., elimination of redundant arithmetic expressions), and very busy expression analysis (i.e., hoisting of arithmetic expressions computed on multiple paths) [2]. At each source code location, data flow analysis records a set of facts about all variables currently in scope. In addition to the set of facts to be tracked, the analysis defines a "kills" set and a "gens" set for each block. The "kills" set describes the set of facts that are invalidated by execution of the statements in the block, and the "gens" set describes the set of facts that are generated by the execution of the statements in the block. To analyze a program, the analysis tool begins with an initial set of facts and updates it according to the "kills" set and "gens" set for each statement of the program in sequence. Although mostly used in compiler optimization [16, 17], data flow analysis has been an integral part of most static analysis tools [18–21].

2.1.3 Abstract Interpretation

This technique was formalized by Cousot and Cousot [22]. It is a theory of semantics approximation of a program based on monotonic functions over ordered sets, especially lattices [23]. The main idea behind this theory is as follows: A concrete program, its concrete domain, and semantics operations are replaced by an approximate program in some abstract domain and abstract semantic operations. Let L be an ordered set, called a concrete set, and let L' be another ordered set, called an abstract set. A function α is called an abstraction function if it maps an element x in the concrete set L to an element $\alpha(x)$ in the abstract set L'. That is, element $\alpha(x)$ in L' is the abstraction of x in L. A function γ is called a concretization function if it maps an element x' in the abstract set L' to an element $\gamma(x')$ in the concrete set L. That is, element $\gamma(x')$ in L is a concretization of x' in L'. Let $L1$, $L2$, L' 1, and $L'2$ be ordered sets. The concrete semantics f is a monotonic function from $L1$ to $L2$. A function f' from $L'1$ to $L'2$ is said to be a valid abstraction of f if for all x' in $L'1$, $(f \circ \gamma)(x') \leq (\gamma \circ f')(x')$. The primary challenge to applying abstract interpretation is the design of the abstract domain of reasoning [24]. If the domain is too abstract, then precision is lost, resulting in valid programs being rejected. If the domain is too concrete, then analysis becomes computationally infeasible. Yet, it is a powerful technique because it can be used to verify program correctness properties and prove absence of errors [25, 26].

2.1.4 Constraint-Based Analysis

A constraint-based analysis traverses a program, emitting and solving constraints describing properties of a program [27, 28]. This technique works in two steps. The first step, called constraint generation, produces constraints from the program text, which describe the information or behavior desired from the program. The second step is constraint resolution which involves solving the constraint by computing the desired information. Static information is then extracted from these solutions.

One of the key features of this technique is that algorithms used for constraint resolution can be written independently of the eventual constraint system used [28].

2.2 *Miscellaneous Techniques*

2.2.1 Symbolic Execution

In this method, instead of supplying normal inputs to the program, one uses symbols representing arbitrary values [29]. The execution proceeds as in a normal execution except that the values are now symbolic formulas over input values. As a result, the output values computed by a program are expressed as a function of the input symbolic values. The state of a symbolically executed program includes the symbolic values of program variables, a path condition (PC) and a program counter. The path condition is quantifier-free Boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a program. The tree nodes represent program states, and they are connected by program transitions. Symbolic execution is the underlying technique of several successful bug-finding tools like PREfix [18] and CodeSonar [19].

2.2.2 Theorem Proving

Theorem Proving is based on the deductive logic proposed by Floyd and Hoare [30, 31]. A program statement S is represented as a triple $\{p\}S\{q\}$, where p (precondition) and q (post-condition) are logical formulas over program states. This triple is valid iff for a state t satisfying formula p, executing S on t yields a state t' which satisfies q. Various inference rules are then used to verify system states. One of the most famous theorem provers is Simplify [32] which has been used in tools like ESC/Java [21]. Tiwari and Gulwani [33] have used theorem proving for a new technique called logical interpretation, i.e., abstract interpretation over logical lattices. This technique tries to embed theorem proving technology in the form of engines of proof into the static analysis tools based on abstract interpretation. Then, it is used for checking as well as generating program invariants.

2.3 *Static Analysis Tools*

There are numerous commercial as well as open-source static analysis tools available. A summary of these tools is given in Table 2. We have categorized these tools in the following categories based on the way they are used:

Table 2 Summary of static analysis tools

Tools	Static checking	Error detection/bug finding	Verification	Type inference
Lint family [34–36]	✓			
JLint [37]	✓			
FindBugs [20]	✓	✓		
PMD [39]	✓	✓		
Coverity Prevent [41]		✓		
KlockWork K7 [40]		✓		
ASTREE [25]		✓	✓	
CGS [10]			✓	
Polyspace Verifier [26]			✓	
TVLA [42]			✓	
BANE [44]				✓
BANSHEE [45]				✓
CQual [43]				✓
PREfix [18]		✓		
CodeSonar [19]		✓		
ESC/Java [38]	✓		✓	
ESP [21]		✓	✓	

- Static checking [20, 34–39]: The aim here is to find common coding errors.
- Error detection/bug finding [18, 19, 40, 41]: The aim of the tools in this category is to find bugs and report them.
- Software verification [10, 25, 26, 42]: The tools in this category guarantee the absence of errors by providing proofs.
- Type qualifier inference [43–45]: The tools in this category specify as well as check program properties.

The first static analysis tool was Lint [34]. It was a simple Unix utility command which accepted multiple files and library specifications and checked them for inconsistencies like unused variables and functions, unreachable code, non-portable character, and pointer alignment. But Lint could not identify defects that can cause run-time problems. A series of shallow static analysis tools like FlexeLint/PCLint [35] and Splint [36] then followed. These tools relied on the syntactic information, and the analysis is mainly path or context sensitive. Splint additionally checks for security vulnerabilities. These tools mainly worked for C/C++. JLint [37] is the Lint family tool to check Java code for inconsistencies, bugs, and synchronization problems. Other tools for Java which use syntactic pattern matching include FindBugs [20] and PMD [39]. FindBugs also uses a simple, intra-procedural data flow analysis to check for null pointer dereferences. PMD is an open-source, rule-based, static analyzer that analyzes Java source code based on the evaluative rules that have been enabled during a given execution. The tool comes with a default set of rules which can be used to unearth common development mistakes such as having empty try-catch blocks, variables that are never used, and objects that are unnecessary.

ESP [21] uses path-sensitive data flow analysis for error detection. It uses a combination of scalable alias analysis and property simulation to verify that large C code bases obey temporal safety properties. Symbolic execution is the underlying technique of the successful bug-finding tool PREfix for C and C++ programs [18]. For each procedure, PREfix synthesizes a set of execution paths, called a model. Models are used to reason about calls, and then, fixpoints of models are approximated iteratively for recursive and mutually recursive calls. Polyspace Verifier [26] uses sound analysis based on abstract interpretation which can prove absence of run-time errors related to arithmetic operations. Prevent [41] uses path simulation techniques to prune the infeasible paths and helps in curbing false alarms. CodeSonar [19] uses symbolic execution engine to explore program paths, reasoning about program variables and how they relate. It then uses data flow analysis to prune infeasible program paths from being explored. Only Klockwork K7 [40] supports software metrics in this category apart from other common facilities. CGS [10] is a precise and scalable static analyzer which uses abstract interpretation to check run-time errors in embedded systems and has been used extensively on real NASA space missions.

ASTREE [25] has similar foundational theory as CGS but also is sound. It is used for checking run-time errors as well as assertion violations and has been used in safety-critical systems. Three Valued Logic Analysis Engine (TVLA) [42] is a system that can automatically generate abstract interpretation analysis algorithms from program semantics. It has been successfully used in performing shape analysis of heap allocated data [46]. The most popular static analysis tool which uses theorem proving is ESC/Java [38]. To use ESC/Java, the programmer adds preconditions, post conditions, and loop invariants to source code in the form of special comments called annotations. It then uses Simplify theorem prover [32] to verify that the program matches the specifications.

CQual [43] is a constraint-based analysis tool used for type inference. It traverses the program's abstract syntax tree and generates a series of constraints that capture the relations between type qualifiers. If the constraints have no solution, then there is a type qualifier inconsistency, indicating a potential bug. BANE [44] is a framework which helps in developing constraint-based program analyses. It frees the user from writing the constraint solver which is the most difficult task in constraint-based analysis. The user only writes code to generate constraints from program texts. Banshee [45] is a scalable version of BANE. It inherits a lot of features of BANE and has added the support for incremental analysis using backtracking.

3 A Critique of Static Analysis

3.1 Precision

Static analysis can be imprecise because it makes assumptions or approximations about the run-time behavior of a program. Therefore, it suffers a lot from false positives. One way to avoid false positives is to filter the analysis results, removing errors which are highly unlikely. For example, in PREfix [18], the user of the tool can control the maximum number of execution paths that will be generated for a function. Another way to avoid false positives is to prune the paths which may be infeasible. For example, Coverity Prevent [40] uses SAT solvers for this kind of false path pruning. But filtering or pruning can introduce false negatives. Therefore, use of filtering/pruning should be done with caution.

3.2 Efficiency

Efficiency is an important attribute as it directly links to the cost of the analysis. Efficiency is also related to precision. Static analysis trades off precision over efficiency; that is, in order to provide analysis results faster, it uses approximation or abstraction mechanisms which lead to less precise results. But there exist techniques which can make static analysis precise by making static analysis flow, path, or context sensitive [12]. But this precision comes at the cost of longer time and greater resource need.

3.3 Coverage

It implies the total number of execution paths analyzed [5, 11]. Obviously, the execution paths analyzed must be valid ones. By its very nature, static analysis provides high coverage as it does not depend on specific input stimuli. Therefore, as compared to its dynamic counterparts, static analysis provides high coverage of code for analysis purpose.

3.4 Scalability

In general, a system is said to be scalable if it remains suitably efficient and practical when applied to situations in the large [5]. This is required when one applies analysis approaches to industrial-size software. Incorporating scalability into a static analysis is tedious as it operates on some kind of model of the system. Most static analysis tools employing abstract interpretation [10, 25, 26] have been developed to pursue scalability. ESP [21] is made scalable using property simulation.

3.5 Modularity

A program is analyzed by analyzing its parts (such as components, modules, and classes) separately, and then, analysis results are composed together to get the required information on the whole program [24]. The advantage of using this approach is that precision, efficiency, and scalability can be greatly enhanced. Work has been done on modular static analysis by Cousot and Cousot [47] to enhance scalability. Dillig [48] describes a fully modular, summary-based pointer analysis that can systematically perform strong updates to abstract memory locations reachable through function arguments. ESC/Java [38] is a static analysis tool which supports modular analysis.

3.6 Automation

It implies the extent to which the analyzer analyses the code by itself without requiring human intervention [12]. Static analysis is a fully automated technique and thus is the most popular choice for analysis of source code [5].

4 Conclusion

This paper studies the foundational techniques of static program analysis. It provides a list of static analysis tools with a brief overview of these tools. It further provides a critique of static analysis over six attributes, namely precision, efficiency, coverage, scalability, modularity, and automation. Static analysis is the most widely used technique for program analysis because of its high efficiency, coverage, and automation, but suffers from high false-positive rates. Still, static analysis techniques are being widely adopted by software community in various fields like bug finding and software verification.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, Tools. Addison Wesley, Boston (1986)
2. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. 1st edn, p. 452, Springer, Berlin. (1996) ISBN: 978-3-540-65410-0
3. Kirkov, R., Agre, G.: Source Code Analysis: An Overview. Cybernetics and Information Technologies, Bulgarian Academy of Sciences (2010)
4. Ball, T.: The concept of dynamic analysis. In: Proceedings of 7th ACM/SIGSOFT Conference on Foundations of Software Engineering (1999)
5. Binkley, D.: Source Code Analysis: A Road Map. Future of Software Engineering, pp. 15–30. Minneapolis, USA, 23–25 May 2007
6. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: Proceedings of the 5th ACM Workshop on Program Analysis for Software Tools and Engineering (2004)
7. Gosain, A., Sharma, G.A.: Survey of dynamic program analysis techniques and tools. In: Proceedings of 3rd International Conference on Frontiers in Intelligent Computing Theory and Applications, Bhubaneshwar, vol. 1, pp. 113–122 Nov (2014)
8. Bentonino, A.: Software testing research: achievements, challenges, dreams. Future Softw. Eng. (2007)
9. Emaneulsson, P., Nilson, U.: A comparative study of industrial static analysis tools. Electron. Notes Theor. Comput. Sci. **217**, 5–21 (2008)
10. Brat, G., Venet, A.: Precise and scalable static program analysis of NASA flight software. In: IEEE Aerospace Conference, March (2005)
11. Jackson, D., Rinard, M.: Software analysis: a road map. IEEE Trans. Softw. Eng. (2000)
12. D'Silva, V., Kroenig, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. CAD (2008)
13. Cifuentus, C.: BegBunch—benchmarking for C bug detection tools. DEFECTS (2009)
14. Pemdergrass, J.A., Lee, S.C., McDonnell, C.D.: Theory and practice of mechanized software. Johns Hopkins APL Technical Digest, **32**(2) 2013
15. Kildall, G.A.: A unified approach to global program optimization. POPL (1973)
16. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. J. ACM **23**(1), 158–171 (1976)
17. Kennedy, K.A.: Survey of data flow analysis techniques. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, pp. 5–54. Prentice-Hall, Englewood Cliffs (1981)
18. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. Softw. Pract. Experience **30**(7), 775–802 (2000)
19. GrammaTech Inc. Overview of grammatech static analysis technology. White paper (2007)
20. Hovemeyer, D., Pugh, W.: Finding bugs is easy. http://www.cs.umd.edu/˜pugh/java/bugs/docs/findbugsPaper.pdf (2003)
21. Das, M., Lerner, S., Siegel, M.: ESP: path sensitive program verification in polynomial time. PLDI'02, Berlin (2002)
22. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of 6th ACM Symposium on Principles of Programming Languages. California, pp. 238–252, (1977)
23. Jones, N.D., Nielson, F.: Abstract Interpretation: A Semantics Based Tools for Program Analysis. Handbook of Logics in Computer Science, vol. 14. Oxford University Press, Oxford (1995)
24. Cousot, P.: Abstract Interpretation Based Formal Methods and Future Challenges. Lecture Notes in Computer Science#2000, pp. 138–156. Springer, Berlin (2001)
25. Cousot, P., Cousot, R., Feret, J., Mine, A., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of static analyzer: a comparison with astree. In: 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE), June (2007)

26. Polyspace Verifier. http://www.polyspace.com
27. Aiken, A.: Introduction to set constraint-based program analysis. Sci. Comput. Program. **35,** 79–111 (1999)
28. Gulwani, S., Shrivastava, S., Venkatraman, R.: Program analysis as constraint solving. PLDI, June (2008)
29. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
30. Floyd, R.: Assigning meanings to programs. In: Proceedings of Symposium on Applied Mathematics (1967)
31. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
32. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. www.hpl.hp.com/techreports/2003/HPL-2003-148.html
33. Tiwari, A., Gulwani, S.: Logical interpretation: static program analysis using theorem proving. In: Proceedings of Conference on Automated Deduction (2007)
34. Johnson, S.C.: Lint: A C program checker. Unix programmer's manual, Computer Science Technical Report 65. AT & T Bell Laboratories (1978)
35. FlexeLint/PCLint. http://www.gimpel.com/html/lintinfo.htm
36. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Softw. **19**, 42–51 (2002)
37. JLint. http://artho.com/jlint
38. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 234–245 (2000)
39. PMD/Java. http://pmd.sourceforge.net
40. Klocwork. Klocwork K7. http://www.klocwork.com
41. Chelf, B., Chou, A.: Next generation of static analysis: Boolean satisfiability and path simulation-a perfect match. http://www.coverity.com/library/pdf/coverity_white_paper_SAT_next_Generation_Static_Analysis.pdf. Downloaded on Dec 2012
42. Ami, T.L., Sagiv, M.: TVLA-a system for implementing static analyses. In: Static Analysis Symposium (2000)
43. Foster, J.S.: Type qualifiers: lightweight specifications to improve software quality. Ph.D. thesis, UCB (2002)
44. Aiken, A., F¨ahndrich, M., Foster, J., Su, Z.: A toolkit for constructing type- and constraint-based program analyses. In: Proceedings of the 2nd International Workshop on Types in Compilation, LNCS #, vol. 1473, pp. 76–96, March (1998)
45. Kodumal, J., Aiken, A.: Banshee: a scalable constraint- based analysis toolkit. In: Proceedings of the 12th International Static Analysis Symposium, pp. 218–234 (2005)
46. Ami, T.L., Reps, T., Sagiv, M., Wilhelm, R.: Putting static analysis to work for verification: a case study. ISSTA (2000)
47. Cousot, P., Cousot, R.: Compositional separate modular static analysis of programs using abstract interpretation. In: Proceedings of 2nd International Conference on Advances in Infrastructure for E-Business, E-Science, E-Education on the Internet (2001)
48. Dillig, T.: A modular and symbolic approach to static program analysis. Ph.D. Dissertation, Department of Computer Science, Stanford University (2011)