Static Analysis Taxonomies

PALANIAPPAN MUTHURAMAN*, University of Paderborn, Germany

Context: Static analysis is a fundamental technique in software engineeering used to analyse program code without executing it. It helps in finding security vulnerabilities, defects in the software system early in the development lifecycle. Despite its effectiveness, it still faces challenges related to scalability, precision impact, performance overhead, computational complexity. Traditional static analysis techniques may generate false positives, struggle to scale for large code bases, and demand significant computational resources. Prior research has demonstrated that the performance of the static analyses can be significantly enhanced through various optimization techniques, such as staging, sparse analysis, and parallelization.

Objective: However, these optimizations were implemented as one-off optimizations, applied to a single static analysis in a single analysis context. Despite the extensive work on static analysis, there remains a lack of a systematic assessment of the various static analysis optimizations techniques,their potential combinations, and the conditions under which combinations are effective. The objective is to provide a comprehensive overview of the state-of-the-art static analysis optimizations, examining their individual and joint impact on analysis outcomes. By analyzing these optimizations both in isolation and in combination, we aim to assess their influence on key performance metrics such as precision, recall, and runtime. Furthermore, this study seeks to identify which classes of program benefit most from specific optimizations or combinations of them, thereby offering guidance on the applicability of these optimization techniques in practical analysis scenarios.

Method: We conducted a Systematic Literature Review (SLR) by analyzing 124 research papers published in static analysis, program analysis, and software engineering venues over the past 15 years (January 2009 to October 2024). The primary objective of this review is to gather insights into the problems addressed by these approaches, the fundamental techniques employed, the static analysis sensitivities considered, and the potential for optimization.

Result: TODO: Write the results at the end

Conclusion: TODO: Write what has been done and what is been lacking for the futire research to be taken care of

ACM Reference Format:
Palaniappan Muthuraman. 2025. Static Analysis Taxonomies. 1, 1 (August 2025), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

With the introduction of the planned EU Cyber Resilience Act, software developers will soon be required to adopt software assurance techniques to ensure the security and reliability of their software products. Static analysis is a cornerstone of modern software engineering, examining the code without executing it. By systematically analysing the code base, static analysis can detect a wide range of security vulnerabilities, bugs, and code smells early in the development process. This proactive approach not only enhances code quality but also reduces the cost and effort associated with fixing issues later in the software lifecycle. Static analyses techniques needs to process millions of lines of code, which result in substatial computational overhead. Moreover, increasing the precision of the analysis -

Author's Contact Information: Palaniappan Muthuraman, palaniappan.muthuraman@upb.de, University of Paderborn, Paderborn, Germany.

Manuscript submitted to ACM

while reducing false positives - typically comes at the cost of longer runtimes and higher resource computation [9]. Traditional static analyses approaches often struggle to meet all three objectives simultaneously.

To address these challenges and to improve performance, researchers have proposed numerous optimization techniques, For example, a **staged approach**, where the analysis is divided in successive phases, each one becoming progressively more precise and computationally expensive [3, 4, 13, 20, 24, 28, 29]. The core idea is to include early stages that perform efficient pre-analyses, whose results help avoid unnecessary computations in the later, most costly phases. **Sparse Analysis** which restricts computations to only the relevant portions of the program by leveraging value-flow graphs [6, 25, 27, 30]. While traditional static analyses often operate over the entire program's control flow graph, sparse analysis focuses solely on code segment relevant to the specific analysis at hand, utilizing a value-flow graph composed of the def-use chains. Prior studies have shown that these optimizations can significanlty enhance performance, reduce analysis time, and in some cases even improve precision. However, these optimizations are often implemented as one-off solutions tailored to a specific static analysis and a particular context.

The existing literatute is rich with individual optimization techniques; however, there is a lack of a systematic understanding regarding their compared effectiveness, their potential synergies, and the conditions under which they perform best - whether in isolation or in combination. When combined, it is crucial to identify which combinations yield the best results, as one optimization can sometimes impede or counteract another.

This study seeks to fill this gap by conducting a **Systematic Literature Review** of 124 research papers published between Januray 2009 and October 2024 across leading venues in static analysis, program analysis. and software engineering. Throug this SLR, we aim to:

1. Catalog and classify state-of-the-art static analysis optimization techniques.
2. Examine the interplay between the different optimizations and identify effective combinations.
3. Determine which classes of programs benefit most from specific optimizations.

2 Background

With the advancement of software development techniques, modern software systems have become more than collection of sourcecode. They represent intricate networks of interdepent components focusing on performance constraints, memory management rules, concurrency, and security expectations. As complexity of these systems increases, so does the necessity of uncovering bugs, finding performance bottlenecks, and violations of security and privacy policies. Static analysis has emerged as a fundamental technique in this context, enabling the examination of code without execution to predict program behavior, ensure correctness, and optimize performance. Static analysis isn't one monolithic tool. It encompasses a collection of specialized techniques, each characterized by distinct objectives, input requirements, and analysis goals. To illustrate the practical significance of static analysis in real-world applications, consider the scenario of a software engineer tasked with developing a critical real-time system. Despite extensive testing efforts, certain execution paths, such as rare thread interleaving or subtle memory errors may remain undetected. Static analysis addresses this limitation by examining all potential execution paths without requiring program execution. Thus, enabling the early detection of bugs and security problems in safety-critical systems.

We can categorize static analysis techniques based on their underlying principles and intended functionality.

2.1 Understanding by Input: What do we need to know?

For static analysis to begin, it requires certain essential inputs. One is **control-flow information**, which maps out the program's execution structure, such as which function calls which other functions, and the possible execution path through the code. Another is **data-flow information**, which keeps track of how data flows through the program, such as who writes to a variable, who reads them, and the sequence of these operations. Espeically with the advancements in objec-oriented programming, understanding the method behavior often depends on the runtime of the objects. For static analysis to handle this, it requires additional information such as type and object details. In programs with limited dynamic behavior, virtual calls can often directlt linked to their respective target methods. However, as dynamic features become more prevalent in object-oriented languages, resolving the actual method being called becomes increasingly challening. This is where **Points-to-analysis** [19] plays a crucial role - it helps determine the relationships between the program variables and objects they may refer to, enabling more accurate call resolution. It often serves as a foundational brick for many static analyses.

2.2 Security First: Guarding the Gates

One group of static analysis techniques focus on software security. These methods are designed to detect potential vulnerabilities by systematically analyzing the codebase for threats. One notable example of the security based techniques is **Taint Analysis** [1, 12, 18, 33], which tracks the propagation of untrusted or user-controlled inputs throughout the program. By identifying whether such inputs reach sensitive operations - such as database queries, or system commands. Taint analysis helps prevent security breaches, such as injection attacks. **Buffer overflow analysis** [11, 17, 35] ensures that the program doesn't write past the allocated memory - a common cause of runtime exceptions and potential security vulnerabilities. **Resource Leak analysis** [34] focuses on the proper management of the system resoures.It verifies that all acquired resources - such as file handles, locks, or network sockets, are eventually released after use. This analysis helps in reduction of the performance degradation due to resource exhasution.

2.3 Verifiers of Truth: Ensuring Program Correctness

These static analysis techniques prioritize correctness about all else - speed, size , runtime, performance are secondary concerns. Their sole objective is to establish the truth about the program behavior. **Abstract Interpretation** [8] over-approximates the program behavior ensures soundness while striving to maintain scalability albeit precision. **Symbolic Execution** [14] explores all possible execution paths by using symbolic inputs instead of concrete values. It generates logical constraints that must hold for each path to be taken. This technique can also be used to verify whether certain properties may be violated, making it effective for detecting potential errors or proving program properties [2].

2.4 Performance-Hungry: Trimming the Fat

This set of techniques targets performance optimizations. **Loop Optimizations** [26] and redundant load elimination aim to identify and eliminate inefficiencies in how loops are constructed and how values are accessed from memory. **Escape analysis** [7] determines whether an object is confined to a method; if so, it suggests allocating the object on the stack rather than heap, which is faster, safer and more efficient. **Deadcode Elimintation** [5, 16] analyses the code to detect portions of code that never gets executed under any inputs, allowing them to be removed safely.

2.5 Concurrency and Beyond: Taming Parallelism

As software increasingly targets multi-core and distributed environments, concurrency analysis becomes vital. Static analysis techniques such as **LockSet analysis** [10], tracks how threads acquired and release locks to detect potential race conditions. Some researchers have explored parallelizing the pointer analysis by distributing the workload across multiple threads [21, 32]. Moreover, advanced models like **Pushdown Systems** [31] are employed to model the recursive procedural calls enabling more precise interprocedural analysis.

2.6 A Shared Landscape: Interdependecies

Static analysis does not operate in isolation. Many type of analyses are build upon others - for example, taint analysis relies on data-flow analysis which itself depends on control-flow graphs. Rather than being a simple hierarchy of independent tools, static analyses form a layered and interdependent ecosystem, where different analyses support and complement each other. These ecosystem can be understood along two key dimensions: the goals of the analysis and the inputs they require. The first dimension addresses what the analysis is trying to achieve be it performance, optimization, correctness, or another objective. The second dinmension focuses on the analysis prerequisites - what program properties it must understand to function or reason effectively.

3 Methodology for the SLR

For this SLR, we followed the guidelines provided by Kitchenham [15]. Figure 1 presents the protocol that we have designed to conduct the SLR.

1. Firstly, we formulate the research questions that guides this SLR, and further identify which information to be extracted from the literature.
2. Nextn, we defined the search keywords aimed at retrieving the broadest possible set of relevant publications within the scope of the SLR.
3. To limit our studies on highly relevant papers, we apply exclusion criteria to filter out publications of likely interest.
4. Finally we perform a lightweight backward-snowballing on the selected publications. The resulting set of studies are referred to as the primary publications.

3.1 Research Questions

RQ1: What are the purpose of these static analysis techniques/optimizations? With this research question, we will survey the various optimization techniques in static analysis.

RQ2: How are the analyses designed and implemented?

In this research question, we conduct a detailed study of the analysis that have been developed. It also includes several sub-questions:

RQ2.1 What fundamental techniques are used for by this static analysis optimization?

RQ2.2 What sensitivity features are applied?

RQ2: Are the research outputs publcly available? We aim to investigate whether the developed tools are open-source or publicly available, reproducible, and easily accessible for use by other practitioners.

RQ3: What challenges remain to be addressed? This question addresses issues that have not yet received significant research attention. It also examines how the focus of the research has evolved over time. Additionally, it helps in

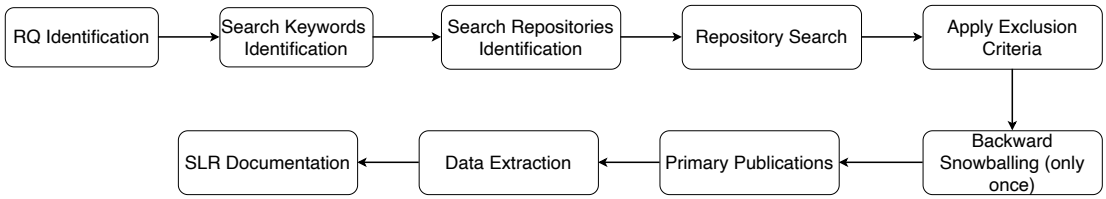

Fig. 1. An overview of the systematic literature review process.

identifying the research gaps in the current knowledge base, and aims to understand the emerging trends and shifts in priorities within the field.

3.2 Search Strategy

This section discusses the keywords we used in our search and the datasets employed to find the relevant publications.

3.2.1 Search Keywords. We used the PICOC strategy to develop our search term. Since the **Intervention (I)** and **Comparison (C)** terms were not relevant to our scope, they were left empty.

Each of the terms from **Population (P)**, **Outcome (O)**, and **Context (C)** formed a seperate line in the search string. The final search string was constructed by logically combining these lines, using **AND** to connect different categories (P, O, C), and **OR** within each category for synonyms or related terms. i.e., s =: P **AND** O **AND** C
Table 1 shows the actual keywords we used, which were derived from a manual investigation of relevant publications.

PICOC	Search Terms
P	"control-flow analysis", "data-flow analysis", "static analysis"
O	"accuracy", "efficiency", "memory usage", "overhead", "performance", "precision", "scalability", "speedup"
C	"control-flow analysis", "data-flow analysis", "static analysis"

Table 1. Search Terms

3.2.2 Search Datasets. We used four well-known repositories, namely ACM Digital Library [1], IEEE Xplore Digital Library [2], Springer Link [3], and Google Scholar [4]. Some of these repositories impose restrictions on the amount of search result metadata that can be downloaded. For instance, Google Scholar does not allow frequent search requests from a single device via its API. To overcome this limitation, we used Publish or Perish [5], a tool that helps retrieve academic documents from Google Scholar. Similarly, Springer Link limits metadata downloads to the first 1,000 search results. However, our search query yielded approximately 10,000 results on this repository. Manually downloading metadata in batches would have been tedious and time-consuming. To handle this efficiently, we used Python scripts to extract data from Springer Link, IEEE Xplore, and ACM Digital Library.

[1] http://dl.acm.org/
[2] http://ieeexplore.ieee.org/
[3] http://link.springer.com/
[4] https://scholar.google.com/
[5] https://harzing.com/resources/publish-or-perish

3.2.3 Exclusion Criteria. The search terms we used were quite broad, resulting in an exhaustive list of publications. Due to this broad scope, many papers in the search results may be irrelevant to our review. To refine our selection and exclude non-relevant papers, we applied specific exclusion criteria, which are detailed below:

1. Given that the majority of scientific publications today are in English, we exclude all non-English papers from our review.
2. Papers under 5 pages in double-column format or under 7 pages in LNCS single-column format were excluded. Additionally, papers exceeding 30 pages in LNCS single-column format were also excluded.
3. If multiple papers described the same or similar approaches, we included only the one with the most comprehensive description. For example, an extended journal paper [23] was selected over its shorter conference version [22].
4. Papers lacking sufficient technical details about their approaches were excluded.
5. Papers that did not focus on optimizing static analysis itself were excluded. For example, papers that use static analysis to optimize the analyzed program were considered out of scope and excluded.
6. Papers that focus on dynamic or hybrid analysis were excluded.

3.2.4 Primary publications selection. Table 2 summarizes the results of the search process. For each paper, we first reviewed the title and keywords to determine its relevance to our use case. If the relevance was unclear, we proceeded to read the abstract. If the abstract was still insufficient to make a decision, we skimmed the paper to assess its suitability. TODO: Usually here we should discuss how many reviewers did read the paper, and how did you overcome the inconsistencies the results? In the end, we got 96 papers in total.

3.2.5 Backward snowballing. To ensure the completeness of our study and to capture relevant works not identified through our initial search terms, we conducted a lightweight backward snowballing process, performed only once. The objective was to identify additional papers cited by our initially selected primary publications that align with the scope of our study. Manual snowballing process it tedious and time consuming, so we developed a series of python scripts to automate and streamline the process. First, We used pdfx [1] to extract the text from the pdf and then retrieved the references from the extracted text. Subsequently, additional scripts were used to filter out papers which fell outside the defined year boundary of our study scope. Furthermore, we used python scripts to extract keywords from the paper, and eliminated those that did not align with the scope of our review. After the automatic filtering stage, we manually reviewed the titles and abstracts of the remaining references. Papers found relevant to the scope of study, and not already included in our primary publications set were added to the final list of primary publications. The relatively high number of additional papers identified through backward snowballing can be attributed to the fine-grained terms used by the original studies, which were not fully covered by our broader search strategy. For instance, many papers contains keywords such as context-sensitivity, context, parallel processing, call graph, call site sensitivity, which, although highly relevant, were not explicitly included in our original search queries. Our initial search term aimed for broader coverage, while snowballing allowed us to capture these more granular studies.

3.2.6 Primary publication Selection. In this section, we present the final selection results of the primary publications, as summarized in Table 2. The first row (search results) shows statistics on the papers retrieved using the keywords defined in the previous section. Through this repositories search, we collect metadata such as paper title, abstracts and additional relevant information. We introduced a second filtering step, due to the inconsistent and often flawed results

[1] https://github.com/metachris/pdfx

Source	IEEE	ACM	Springer	Google Scholar	Total
Search Results	5561	3195	3980	1000	13736
Script verification (keywords)	225	450	45	97	817
After Reviewing Title	125	136	13	43	317
After Reading Abstracts	117	30	22	2	171
After Skimming	72	22	3	20	117
After Final Discussion					96
Backward snowballing					28
Total					124

Table 2. Summary of the Primary Publications Selection Process

of the "advanced" search feature of the search repositories, where the search results are oftern inaccurate resulting in the set of irrelevant and noisy results. After performing the second step (represented in row 2), the number of potential relevant papers are significantly reduced.

4 Graphs

Understanding how control flows through the program, how data moves between the variables, and how different program components depend on each other is a very crucial thing to know before any analysis.

There are 4 different widely used graph structures for them namely Control Flow Graph (CFG), Value Flow Graph (VFG), Program Dependence Graph (PDG), and System Dependence Graph (SDG)

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[3] Eric Bodden. 2009. *Verifying finite-state properties of large-scale programs.* Ph. D. Dissertation. McGill University. Available in print through ProQuest.

[4] Eric Bodden, Patrick Lam, and Laurie Hendren. 2012. Partially evaluating finite-state runtime monitors ahead of time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 2 (June 2012), 7:1–7:52.

[5] Rastislav Bodik and Rajiv Gupta. 1997. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*. 159–170.

[6] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 55–66.

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *Acm Sigplan Notices* 34, 10 (1999), 1–19.

[8] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 324–328.

[9] Pär Emanuelsson and Ulf Nilsson. 2008. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science* 217 (2008), 5–21.

[10] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review* 37, 5 (2003), 237–252.

[11] David Evans and David Larochelle. 2002. Improving security using extensible lightweight static analysis. *IEEE software* 19, 1 (2002), 42–51.

[12] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 106–117.

[13] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (October 2017).

[14] James C King. 1975. A new approach to program testing. *ACM Sigplan Notices* 10, 6 (1975), 228–233.

[15] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.

[16] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. *ACM Sigplan Notices* 29, 6 (1994), 147–158.

[17] Wei Le and Mary Lou Soffa. 2008. Marple: a demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 272–282.

[18] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 98–108.

[19] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *International conference on compiler construction*. Springer, 153–169.

[20] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (October 2018).

[21] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking incremental and parallel pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2019), 1–31.

[22] Jingbo Lu. 2020. *Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with CFL-Reachability*. Ph. D. Dissertation. UNSW Sydney.

[23] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–46.

[24] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (October 2019).

[25] Magnus Madsen and Anders Møller. 2014. Sparse dataflow analysis with pointers and reachability. In *International Static Analysis Symposium*. Springer, 201–218.

[26] Steven Muchnick. 1997. *Advanced compiler design implementation*. Morgan kaufmann.

[27] Ganesan Ramalingam. 2002. On sparse evaluation representations. *Theoretical Computer Science* 277, 1-2 (2002), 119–147.

[28] Nishant Sinha and Chao Wang. 2010. Staged concurrent program analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. Association for Computing Machinery, New York, NY, USA, 47–56.

[29] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: Context-sensitivity, across the board. *SIGPLAN Notices* 49, 6 (June 2014), 485–495.

[30] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDEal: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (October 2017).

[31] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[32] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel pointer analysis with CFL-reachability. In *2014 43rd International Conference on Parallel Processing*. IEEE, 451–460.

[33] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

[34] Westley Weimer and George C Necula. 2004. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. 419–431.

[35] Yichen Xie, Andy Chou, and Dawson Engler. 2003. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 327–336.