Static Analysis Taxonomies

PALANIAPPAN MUTHURAMAN*, University of Paderborn, Germany
LEEN JUBARAH, University of Paderborn, Germany
AASHISH PRAJAPATI, University of Paderborn, Germany
ERIC BODDEN, 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, Leen Jubarah, Aashish Prajapati, and Eric Bodden. 2025. Static Analysis Taxonomies. 1, 1 (September 2025), 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

With the introduction of the planned EU Cyber Resilience Act [11], 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

Authors' Contact Information: Palaniappan Muthuraman, palaniappan.muthuraman@upb.de, University of Paderborn, Paderborn, Germany; Leen Jubarah, leen.jubarah@upb.de, University of Paderborn, Paderborn, Germany; Aashish Prajapati, aashish.prajapati@upb.de, University of Paderborn, Paderborn, Germany; Eric Bodden, eric.bodden@upb.de, University of Paderborn, Paderborn, Germany.

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 - while reducing false positives - typically comes at the cost of longer runtimes and higher resource computation [18]. 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 stages[7, 8, 29, 43, 47, 60, 62]. 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 [12, 49, 55, 63]. 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. 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 challenging. This is where **Points-to-analysis** [38] 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** [5, 27, 36, 68], 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** [20, 35, 70] ensures that the program doesn't write past the allocated memory - a common cause of runtime exceptions and potential security vulnerabilities. **Resource Leak analysis** [69] 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** [15] over-approximates the program behavior ensures soundness while striving to maintain scalability albeit precision. **Symbolic Execution** [31] 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 [6].

2.4 Performance-Hungry: Trimming the Fat

This set of techniques targets performance optimizations. **Loop Optimizations** [52] and redundant load elimination aim to identify and eliminate inefficiencies in how loops are constructed and how values are accessed from memory. **Escape analysis** [13] 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** [9, 33] 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** [19], 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 [44, 67]. Moreover, advanced models like **Pushdown Systems** [64] 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 [32]. 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?
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 identifying the research gaps in the current knowledge base, and aims to understand the emerging trends and shifts in priorities within the field.

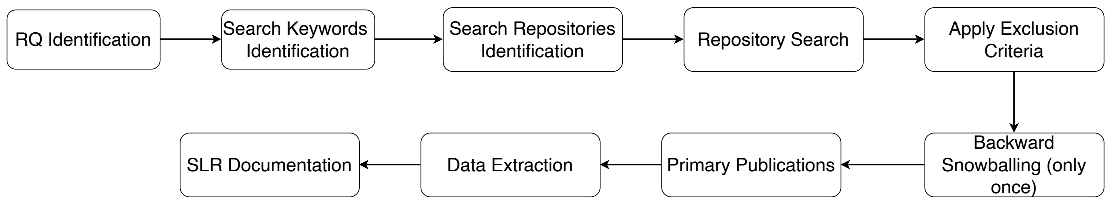

Fig. 1. An overview of the systematic literature review process.

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.

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]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

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 [46] was selected over its shorter conference version [45].
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 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.

[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

4 Pointer Analysis

Pointer analysis is a fundamental and widely applied static analysis technique that aims to determine the possible objects or memory locations that the pointer variables in a program may reference during execution. Among the earliest and most influential contributions in this area is the work of Andersen in 1994, known as Andersen's Analysis, which is flow-, context-, and field-insensitive [2]. This analysis also referred to as a subset based analysis, introduces subset constraints to model the relationships between pointer variables and the memory locations they can potentially point to. Following Andersen's work, Steensgaard proposed in 1996 a unification-based pointer analysis that operates with almost linear time complexity [65]. Unlike Andersen's approach, Steensgard's analysis merges points-to-set wherever any aliasing between two pointer variables, leading to a coarser approximation of the points-to information. As a result, Steensgaard analysis sacrifices precision for scalability, making it significanlty faster and less precise than Andersen's analysis [59]. TODO: Should write the difference between context, object, field, flow sensitivity here??. TODO: Should write about k-site sensitivity??. Since both Andersen's and Steensgaard's analyses are field-, context-, and flow-insensitive, many subsequent researchers sought to improve precision by introducing different forms of sensitivity. In 1994, Emami et al., [17] defined a pointer analysis that, while still flow-insensitive, was extended to be context-sensitive. Their formulation distinguished between "may" and "must" points-to relations, thereby enabling strong updates. Later in 2002, Milanova et al., [51] introduced object-sensitive pointer analysis as a novel form of context sensitivity tailored for flow-insensitive points-to-analysis for Java. Through empirical evaluation, they demonstrated that object-sensitivity provides greater precision than the traditional call-site sensitivity in the context of object-oriented programs. In 2011, Smaragdakis et al., [61] proposed type sensitivity, a different sensitivity approach that leverages the type information as the distinguishing context. Subsequently, in 2013, Kastrinis et al., [30] proposes a hybrid form of context sensitivity that selectively combines both object-sensitivity and call-site sensitivity. Their experimental research showed that this selection hybrid approach yields higher precision than a naive, non-selective combination of the two sensitivities.

Following the foundational works in pointer analysis, researchers have continually sought to enhance both the precision and efficiency of these analyses. Starting from 2009, significant strides have been made in optimizing pointer analysis techniques.

CFL reachability formulations gives very precise points to results but are computationally expensive [71]. In 2009, Xu et al., [71] introduced a novel staging approach which does cheap, conservative, context-sensitive pre analysis, that helps in identifying pairs of variables that cannot alias in the given calling context. Using this pre-analysis information, the main CFL-reachability based pointer analysis is able to prune infeasible CFL paths and thus improve both efficiency and precision.

In 2009, Hardekopf et al., [23] introduces a flow-sensitive, context-insensitive pointer analysis that combines two key innovations: semi-sparse analysis and novel use of BDD's (Binary Decision Diagrams) to efficiently represent large points-to sets. Their approach employs a partial static single assignment form, which allows for a sparse analysis on top-level variables (those whose addresses are not taken), while performing a standard iterative dataflow algorithm for the remaining variables. Additionally, the use of BDDs enables compact representation of points-to information [10] significantly reducing both memory usage and runtime compared to traditional flow-sensitive analyses. Empirical results reported in their work demonstrate that this analysis scales to large programs while maintaining high precision.

In 2009, Lundberg et al., [48] introduced a context- and flow-sensitive pointer analysis approach that significanlty improves upon traditional methods in terms of both speed and precision. Their analysis employed a graph-based Static Single Assignment (SSA) form, which extends the conventional SSA form [16] to effectively represent points-to information. In this representation, each method is modeled as a graph where nodes correspond to operations within the method, and edges denote data dependencies between these operations. By leveraging SSA form, their analysis maintains strict flow-sensitivity. They also introduced "this-sensitivity". This technique analyses methods seperately for each abstract object reaching the implicit this variable, thereby distinguishing between different calling contexts. Empirical evaluations demonstrated that their 1-this-sensitivity analysis is considerably faster than the traditional context-sensitive and object-sensitivity analyses.

In 2010, Mendez-Lojo et al., [50] presented the first parallel implementation of Andersen's inclusion based points-to analysis, developed in Java. They showed that the inclusion-based points-to analysis can be expressed entirely in terms of graph rewriting rules. Their central insight of their approach is that, during the rewriting phase, two active nodes can be processed parallel if do not interfere with each other. Thus, identifying non-interfering active nodes becomes the key to unlock parallelism. They leveraged the Galois system [28] to implement their parallel points-to analysis. The implementation employed two main data structures (i) BDD's with hash table for storing points-to edges, and (ii) sparse bit vector based on linked list for storing other edges. The authors compared their experimental results with those reported by Hardekopf's [23]. However, the results are not directly comparable, since their implementation was in Java, while Hardekopf and Lin's was written in C++.

TODO: Should we write about using parallel client analyses alongside with parallel points to analyses?? Parallel Reachability and Escape Analyses Paper

In 2011, Li et al., [42] introduced a novel approach that reformulates flow-sensitive pointer analysis as a general graph reachability problem over a Value Flow Graph (VFG). In a VFG, nodes represents both memory objects and pointer variables, while directed edges captures value flows between them. Unlike static representations, the VFG is iteratively and dynamically refined by introducing indirect value flows, until a fixed point is reached. This method simplifies the traditionally complex process of tracking precise points-to-relations. As a result, the analysis not only computes precise points-to information but also dervies inverse points-to information (which pointers may reference a given memory object).

In 2011, Lhotak et al., [37] proposed a strong update analysis combining best of both worlds: the efficiency of flow-insensitive analysis and the precision of flow-sensitive analysis through strong updates. The key insight is the use of strong updates which can be applied when the dereferenced points-to set is a singleton. In their approach, singleton points-to sets are tracked using a flow-sensitive analysis, while non-singleton points-to sets are handled using a flow-insensitive analysis. This hybrid strategy allows flow-sensitive analysis to refine the precision of flow-insensitive points-to information, while the flow-insensitive analysis provides a robust fallback in cases where flow-sensitive tracking would be expensive. The analysis leverages SSA form which is effective to perform strong updates. Empirical results demonstrated that this hybrid method produces more precise points-to sets compared to purely flow-insensitive or purely flow-sensitive approaches.

Despite the advancements, flow-sensitive pointer analysis has struggled to scale to large, real-world programs with millions of lines of code [24]. To address these challenge, Hardekopf et al., [24] introduced a new flow-sensitive analysis that leverages a sparse program representation. This sparse representation is not built directly; instead, it is derived from a staged process involving a pre-analysis followed by the main flow-sensitive analysis. The Pre-analysis computes the conservative def-use chains, which helps the primary flow-sensitive analysis to perform sparsely.

Points-to analysis is typically performed by constructing a constraint graph of pointer variables and dynamically updating it, propagating more and more points-to information until a fixed point is reached. [53] Traditionally, the inherent structure of this constraint graph has not been exploited so far. In 2012, Nasre et al., [53] introduced a new approach that exploits the structure of the constraint graph to optimize inclusion-based points to analysis. Their work introduced two key : dynamic pointer equivalence and dominating pointers. They demonstrated that their algorithm is both sound and precise while achieving significant performance improvement over the inclusion-based points-to analysis.

Most existing pointer analysis techniques perform whole program points-to analysis which can improve precision but often comes at the cost of efficiency. In 2012, Shang et al., [58] introduced Dynsum, a novel approach that performs fully on-demand, context-sensitive points-to analysis leveraging CFL-reachability summaries. The key innovation of this technique lies in performing field-sensitive, context-insensitve partial points-to analysis for individual methods, summarizing their local points-to relations and reusing these summaries in the same or different contexts.

Traditional flow-sensitive pointer analysis suffer from the ability to not do strong updates, which is a sever restriction for Java programs. And they also use points-to graphs to store the points-to information.

5 Incremental

Traditional data-flow analyses often require recomputation of the entire analysis for every modification in the codebase, regardless of the size or scope of the change. This approach becomes highly time-consuming when dealing with large codebases. To address this limitation, researchers have proposed various incremental analysis techniques that efficiently update analysis results in response to code changes, thus avoiding unnecessary recomputation. TODO: Should talk about reviser updating incrementally for both phases of IDE? In 2014, Arzt et al., introduced Reviser, a tool designed to incrementally update IDE-based data-flow analyses [4]. Reviser follows a clean-and propagate approach: for each affected node it clears the computed information and recompute the information using all of the node's predecessors. The algorithm identifies changes in the code and their affected predecessors or successors by computing structural differences between Control Flow Graphs (CFG's). TODO: Should have this -> Reviser also replaces the standard IDE Solver with one capable of handling incremental updates? Experimental results demonstrated that Reviser produces the same results as a full analysis while saving 80% of time required for a full recomputation. Notably, Reviser also recomputes the complete call graph.

While Reviser focuses on incremental data-flow analysis in IDE's, In 2017, Sathyanathan et al., [57] proposed a framework for incremental whole program optimization. Their framework uses a simple and fast checksum technique to detect changes in the codebase. Unl;ike Reviser, which targets IDE-based data-flow analyses, this framework is applied directly within the C/C++ compiler. The primary goal here is to reduce recompilation time rather than update data-flow information

TODO: Add An incremental points-to analysis with CFL-reachability here

Another important area for incremental pointer analysis is pointer analysis which is central to most interprocedural analyses. While there has been significant work on parallelizing pointer analysis, relatively little work has focused on making it incremental. TODO: cite In 2019, Liu et al., [44] proposed the first efficient and precise incremental and parallel pointer analysis for Java programs. This approach detects code modifications by comparing the SSA-based IR of the old and new program. This algorithm constructs call graphs on the fly and supports efficient parallelelization within each fixed-point iteration. It leverages a fundamental transitivity property of Andersen's analysis and it is context sensitive, path sensitive, and flow insensitive. Experimental results shows that this achieves more than 200X speedups over other existing approaches and also 2-5 times faster than the whole program pointer analysis.

Existing techniques typically support either incremental or demand-driven analysis, but not both, and they often impose restrictions such as requiring finite abstract domains. In 2021, Stein et al., [66] introduced a framework that unifies incremental and demand-driven analysis for any arbitraty abstract domain, even for those with infinite domains and widening operators. Their approach is based on a data structure called Demanded Abstract Interpretation Graph (DAIG), which treats program edits, client queries uniformly. By capturing dependencies between statements, abstract states, and intermediate computations, DAIG enables efficient reuse of previously computed results while preserving soundness.

Most recently, In 2024, Krishna et al., [34] developed a dynamic algorithm for handling the addition and deletion of lines in the source code with a guarantee that each such modification takes linear time in the worst case. Every edge insertion can be handled by reusing the already constructed component graph, while deletions- which typically requires global re-evaluation of the entire graph - are optimized using techniques from dynamic undirected connectivity and sparsification. Although, construction of the initial component graph requires $O(n^2)$ time due to dense graph, subsequent updates can start from a preliminary component graph.

These analyses optimize different aspects of incremental computation, and many of them can be complementary when combined - for example, pairing parallel fixed-point computation with dynamic graph updates can further improve efficiency. At the same time, paralelization must be integrated with care, as it can interfere with determinstic guarantees required by incremental solvers.

6 Taint Analysis

The Interprocedural Finite Distributive Subset (IFDS) problem is a fixed-point, flow- and context-sensitive data flow analysis problem. In IFDS, data flow is reduced to a reachability problem over the exploded supergraph. The IFDS algorithm is both memory- and compute-intensive, with a worst-case complexity of $O(|E|.|D|^2)$ in space and $O(|E|.|D|^3)$ in time where E denotes the number of edges in the supergraph and D denotes the set of data flow facts in the program [56].

A prominent application of the IFDS framework is taint analysis, which tracks the flow of sensitive data through programs.

In 2014, Arzt et al. introduced FlowDroid [5], a precise, context-, flow-, field-, and object-sensitive static taint analysis tool for Android applications. FlowDroid constructs an interprocedural control-flow graph and leverages the IFDS framework to propagate taints efficiently. While FlowDroid remains a state-of-the-art tool widely used in research and security analysis, it has been reported to suffer from poor scalability: analyzing large Android applications consumes excessive memory and time due to its precise sensitivities. [40]

In 2017, Grech et al., proposed PTaint [21], showing that existing points-to analysis implementations can be reused—almost without modification—to compute information flow, thereby unifying points-to and information-flow analysis. However, PTaint is flow-insensitive, which may introduces false positives.

Since IFDS solvers typically require substantial memory—sometimes exceeding 100 GB of RAM—several works have sought to improve scalability [40]:

In 2019, Choi et al., [14] introduced STAR, a taint analysis tool that is both context- and flow-sensitive and supports multi-source taint analysis. STAR employs a novel summarization technique called as Symbolic Summarization, which replaces concrete taint sources with symbolic taint sources. The authors argur that many temporary objects are local to their allocation sites, do not escape, and therefore need not be propagated beyond their allocation sites. To improve scalability in the IFDS analysis, STAR applies three state-pruning techniques such as escape-based pruning [13], access-based localization [54], and bypassing [54].

In 2019, He et al., [26] proposed SparseIFDS, a sparse alternative to traditional IFDS. Instead of pre-constructing full control-flow graphs, SparseDroid constructs sparse control-flow graphs on demand for each data flow fact, allowing direct propagation to relevant use sites. This approach reduces overhead and achieves an average 22× speedup over FlowDroid while also lowering memory usage when evaluating with the FossDroid dataset [1].

In 2021, Li et al., [40] introduced DiskDroid, A disk-assisted IFDS solver that reduces memory requirements. Instead of storing all path edges, DiskDroid memorizes only frequently accessed ("hot") edges using a Hot Edge Selector. Non-hot edges are recomputed on demand, and inactive edges are swapped to disk once memory usage crosses a threshold. This optimization allows DiskDroid to analyze Android apps with only 10 GB of memory, compared to the 128 GB required by FlowDroid, while incurring only a modest overhead, when evaluated with F-Droid [1].

In 2021, Arzt et al., [3] introduced CleanDroid, An efficient, method-level garbage collector for IFDS solvers. CleanDroid reclaims memory by discarding intermediate data flow facts (edges and taint abstractions) that are no longer needed. Integrated into FlowDroid, CleanDroid reduces memory consumption by 63% on average without sacrificing precision. However, its method-level granularity is conservative, leaving many non-live edges uncollected and occasionally requiring reprocessing of previously collected edges.

In 2023, He et al., [25] developed Fine-Grained Path Edge Collection, to address CleanDroid's limitations, They proposed a data-fact-level garbage collection algorithm (FPC). Unlike CleanDroid, which operates at the method level, FPC selectively collects non-live path edges at a finer granularity, reducing both memory consumption and analysis time. On average, FPC outperforms CleanDroid by a factor of 1.7× in runtime.

In 2023, Gui et al., [22] introduced MergeDroid which targeted at reducing redundant computation in IFDS analyses. Traditional IFDS repeatedly propagates equivalent value flows, and its context-insensitive activation statements lead to false positives. MergeDroid merges equivalent value flows linked to the same abstraction, eliminating redundancy and improving precision. It demonstrates an average 9× performance improvement compared to traditional approaches.

In 2024, Li et al., [41] proposed IDEDroid, a novel approach for field-sensitive data flow analysis within the IFDS/IDE framework. Traditional field sensitivity requires the use of access paths, which generate a large number of data flow facts and hinder scalability. IDEDroid reinterprets access-path generation as a context-free language (CFL) problem, encoded as an IDE problem, thereby allowing propagation only of the base variable. This substantially reduces the number of facts generated and improves scalability.

In 2024, Li et al., [39] introduced SADroid, a tool with flow-sensitivity optimizations. Flowdroid relies on the concept of an activation unit, which marks the source location where the data-flow becomes active. However, each data-flow fact is cloned multiple times at different activation points leading to substantial increase in the number of data-flow facts. While activation units enable flow-sensitivity, they also causes significant overhead in both runtime and memory usage, with number of data-flow facts growing by up to 9.55x. To address this, SADroid simplifies the data-flow fact by disregarding activation points and uses a flow-sensitive path builder, which searches for taint propagation paths in a flow- and context-sensitive manner.

TODO: This text talks about the tool as a whole When considering which optimizations can be combined, pairing SparseDroid with garbage collection approaches (CleanDroid or FPC) is unnecessary. SparseDroid's sparse control-flow graph already ensures that only live edges are present, so adding GC would introduce computational overhead without yielding additional efficiency gains. Similarly, combining SparseDroid with DiskDroid is redundant, since SparseDroid already incorporates a caching mechanism for sparse CFGs. On the other hand, SparseDroid and MergeDroid complement each other well: SparseDroid reduces the number of propagation paths, while MergeDroid eliminates redundancy within the propagated facts.

TODO: This talks just about the optimizations that can be combined TODO: Which one can be added in the final draft? While each optimization addresses specific limitations of IFDS-based taint analysis, these techniques are not mutually exclusive. For example, disk-assisted storage could be combined with method-level garbage collection to simultaneously lower memory usage and avoid costly recomputation of discarded states. Similarly, sparse graph construction could be integrated with flow-sensitive path building to minimize number of edges explored. However, not all optimizations are naturally compatible, and some may undermine each other's effectiveness. For example, fine-grained garbage collection and disk-assisted storage both manage memory reduction, but combining them could introduce execessive overhead. Likewise, symbolic summarization, which abstract taint sources, may conflict with flow-sensitive path building, as the loss of concrete allocation details, can diminish the precision needed for fine-grained flow tracking.

7 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)

Graphs

- **Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs (Hammer et al., 2009):** The Information Flow Control (IFC) is used to analyze a program for security leaks. Existing approaches are not precise enough, that means they report too many false alarms. Therefore in this paper they use program dependence graphs (PDG) for Java bytecode to implement IFC. Their approach is more precise and it is flow-, context and object-sensitive. One limit of their approach is that they can only deal with medium-sized programs (up to 100kLOC).

- **Speeding up context-, object- and field-sensitive SDG generation (Graf, 2010):** In their presented approach they generate SDGs (System dependence graph) efficiently for Java. It is context-, field- and object-sensitive and is based on the WALA framework. They showed that their approach reduces time and memory and can improve precision. The technqical background is that the size of object trees can become large if the points-to analysis is less precise, so they can't be used in SDGs. In their paper they introduce object graphs, an extension of object tress. As a result they merge duplicate information to save space.

- **Quantitative Interprocedural Analysis (Chatterjee et al., 2015):** They present an efficient algorithm which can answer several static analysis questions, e.g. estimating the average energy consumption (cite). For that they need the

ICFG and a positive weight function which determines for every transition a positive number which says how good, bad or neutral this transition (event) is. After that their algorithm compares the good and bad events and determine whether a given threshold is reached in a run. As a result they can quantify the ICFG. Their algorithm is implemented in the Java soot framework and can deal with recursion and is sound. But one limit is that they have not consider mulitple quantitative objectives yet but are aiming that in the future. (TODO: have they already done that and can we find improvements?)

- **A Precise Framework for Source-Level Control-Flow Analysis (Riouak et al., 2021):** Their presented framework is called INTRACFG which constructs precise intraprocedrual CFGs. It is based on reference attribute grammers and enables the construction of CFGs which are independent of the shape and structure of the AST because other frameworks for constructing CFGs are usually depending from the structure of ASTs. INTRACFG only connects AST nodes of interest in the CFG. Attribute grammers denote AST nodes with attributes, reference attribute grammers extend them: their values are references to other AST nodes (cite). In their work they achieved to reduce the number of nodes and edges in the CFGs by over 30

- **Efficient Path-Sensitive Data-Dependence Analysis (Yao et al., 2021):** They present a path- and context-sensitive data-dependence analysis with the goal to solve the **aliasing-path-explosion problem** via a sparse and demand-driven approach. The exisiting challenge for data-dependence analysis is the aliasing-path-explosion problem which means to tracks a large amount of aliases. Their approach summarizes data dependence as symbolic and storeless value-flow graphs and they perform a demand-driven phase that resolves transitive data dependence over the graphs. The presented framework is called FALCON, a fused and sparse approach for path-sensitive data-dependence analysis. First their approach computes storeless value-flow graphs, as a result no duplicated edges. Then this graph is used for data-dependencies in a demand-driven way. By that a path- and context-sensitive def-use information is tracked on demand. It is more efficient because the data dependency can be figured out without points-to information

- **Path-Sensitive Sparse Analysis without Path Conditions (Shi et al., 2021) :** The sparse program analysis considers only necessary control flows but is still expensive for path-sensitive analysis (cite). In this work the authors present Fusion, a fused approach to inter-procedurally path-sensitive sparse analysis (cite). They can determine the path feasiblity without knowing the path conditions, this saves cost and time. In their approach the SMT solver works on the program data dependece graph. That's how they can determine path feasibility directly. They showed that they were able to detect a lot of bugs in mature open-source software with their approach,faster than two other state-of-art approaches. Fusion can analyze milliones of lines of code within a short time and less memory compared to other approaches. They achieved the precision of inter-procedural path-sensitivity.

8 Call Graphs

In 2016, Petrashko et al., introduced context-sensitive call graph construction algorithms that leverage generic type information in object-oriented languages. They proposed two extensions to Scala: one that uses actual type arguments for method contexts and another that refines contexts using more precise subtypes from static types of actual arguments. Their approach significantly improved call graph precision and reduced analysis time, demonstrating the value of incorporating generic type information into call graph analyses.

In 2020, Santos et al., introduced Salsa targets Java programs with serialization and deserialization, enhancing existing points-to and call-graph analyses. It employs an on-the-fly, iterative framework that refines call graphs under explicit assumptions about serialization-related behavior. It addresses the challenges of performing static analysis on programs with dynamic features, particularly focusing on serialization and deserialization processes. To regain soundness, Salsa injects synthetic nodes and edges modeling serialization effects into previously computed call graphs and re-analyzes until convergence. Initial results on the Java Call Graph Test Suite (JCG) indicate that Salsa improves call-graph soundness for programs exercising serialization features.

In 2022, Le-Cong et al., introduced AutoPruner, a novel technique for call graph pruning that utilizes both statistical semantic and structural analysis to eliminate false positives in call graphs. Unlike previous machine learning approaches that primarily relied on structural features, AutoPruner employs a Transformer-based model to capture semantic relationships between caller and callee functions. Unlike cgPruner and pure program-analysis methods that consider only the caller, AutoPruner analyzes the source code of both the caller and callee for each edge. It derives semantic embeddings using a pre-trained code model (e.g., CodeBERT), which can learn relationships from paired inputs, and combines them with per-edge structural features. Given a call graph from a static analyzer, AutoPruner preprocesses the graph, extracts these features, and feeds them to a neural classifier that predicts false-positive edges; predicted false positives are pruned to yield a more precise call graph with fewer false alarms. Empirical evaluations demonstrate that AutoPruner significantly outperforms state-of-the-art methods, achieving up to 13% improvement in F-measure and reducing false alarm rates in client analyses.

In 2024, Santos et al., introduced Seneca, a call-graph construction technique for Java that makes serialization and deserialization callbacks explicit. By combining taint analysis with API modeling, it resolves callback targets arising during (de-)serialization to produce sound call graphs with minimal overhead. Seneca performs a novel taint-based call graph construction, which relies on the taint state of variables when computing possible dispatches for callback methods. In evaluation, Seneca identifies vulnerable paths related to untrusted deserialization, passes all serialization-callback tests, and yields fewer spurious edges than Soot and OPAL, while remaining practical in performance.

In 2024, Helm et al., introduced Unimocg, a modular call-graph framework that separates type-information computation from call resolution, enabling interchangeable algorithms and language-feature modules. It supports reuse of type information in downstream analyses, preserves soundness across ten algorithms without precision or performance loss, and outperforms Soot and WALA's emulated RTA.

TODO: Future Directions and Combinations

Future work should investigate composing modular construction and pruning techniques to further improve efficiency and precision. A practical pipeline could construct call graphs with Unimocg and subsequently prunes edges with AutoPruner. Seneca can also be integrated as the serialization/callback module within Unimocg, followed by AutoPruner to remove residual false positives. As an alternative, Salsa's serialization modeling can precede AutoPruner. For Scala codebases, Petrashko et al.'s context-sensitive construction can be paired with AutoPruner. Seneca and Salsa should not be combined in the same pipeline because they target the same (de-)serialization callbacks; one should be selected based on requirements.

References

[1] 2019. *Fossdroid*. https://fossdroid.com/

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Citeseer.

[3] Steven Arzt. 2021. Sustainable solving: Reducing the memory footprint of IFDS-based data flow analyses using intelligent garbage collection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1098–1110.

[4] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering.* 288–298.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] Randal E Bryant. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on* 100, 8 (1986), 677–691.

[11] Polona Car and Stefano De Luca. 2022. EU Cyber resilience act. *EPRS, European Parliament* (2022).

[12] 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.

[13] 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.

[14] Wontae Choi, Jayanthkumar Kannan, and Domagoj Babic. 2019. A scalable, flow-and-context-sensitive taint analysis of android applications. *Journal of Computer Languages* 51 (2019), 1–14.

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

[16] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[17] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices* 29, 6 (1994), 242–256.

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

[19] 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.

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

[21] Neville Grech and Yannis Smaragdakis. 2017. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[22] Yujiang Gui, Dongjie He, and Jingling Xue. 2023. Merge-replay: Efficient ifds-based taint analysis by consolidating equivalent value flows. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–331.

[23] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. *ACM SIGPLAN Notices* 44, 1 (2009), 226–238.

[24] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 289–298.

[25] Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. 2023. Reducing the memory footprint of IFDS-based data-flow analyses using fine-grained garbage collection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 101–113.

[26] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 267–279.

[27] 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.

[28] ISS Group, University of Texas at Austin. [n. d.]. *Galois Project.* https://iss.ices.utexas.edu/projects/galois/

[29] 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).

[30] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices* 48, 6 (2013), 423–434.

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

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

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

[34] Shankaranarayanan Krishna, Aniket Lal, Andreas Pavlogiannis, and Omkar Tuppe. 2024. On-The-Fly Static Analysis via Dynamic Bidirected Dyck Reachability. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1239–1268.

[35] 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.

[36] 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.

[37] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 3–16.

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

[39] Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Lian Li, and Lin Gao. 2024. Boosting the Performance of Multi-solver IFDS Algorithms with Flow-Sensitivity Optimizations. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 296–307.

[40] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling up the IFDS algorithm with efficient disk-assisted computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 236–247.

[41] Haofeng Li, Chenghang Shi, Jie Lu, Lian Li, and Jingling Xue. 2024. Boosting the performance of alias-aware IFDS analysis with cfl-based environment transformers. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2633–2661.

[42] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 343–353.

[43] 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).

[44] 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.

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

[46] 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.

[47] 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).

[48] Jonas Lundberg, Tobias Gutzmann, Marcus Edvinsson, and Welf Löwe. 2009. Fast and precise points-to analysis. *Information and Software Technology* 51, 10 (2009), 1428–1439.

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

[50] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 428–443.

[51] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 1–11.

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

[53] Rupesh Nasre. 2012. Exploiting the structure of the constraint graph for efficient points-to analysis. In *Proceedings of the 2012 international symposium on Memory Management*. 121–132.

[54] Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. 2011. Access analysis-based tight localization of abstract memories. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 356–370.

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

[56] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[57] Patrick W Sathyanathan, Wenlei He, and Ten H Tzen. 2017. Incremental whole program optimization and compilation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 221–232.

[58] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 264–274.

[59] Marc Shapiro and Susan Horwitz. 1997. The effects of the precision of pointer analysis. In *International Static Analysis Symposium*. Springer, 16–34.

[60] 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.

[61] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 17–30.

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

[63] 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).

[64] 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.

[65] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.

[66] Benno Stein. 2022. *Demanded abstract interpretation.* University of Colorado at Boulder.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*. Springer, 98–122.