

Flow Sensitivity without Control Flow Graph: An Efficient Andersen-Style Flow-Sensitive Pointer Analysis

Jiahao Zhang^{a,1}, Xiao Cheng^{a,1} and Yuxiang Lei^{a,*}

^a*School of Computer Science and Engineering, University of New South Wales, Sydney, Australia*

ARTICLE INFO

Keywords:

program analysis
static analysis
pointer analysis
flow-sensitivity
constraint graph
performance

ABSTRACT

Flow-sensitive pointer analysis constitutes an essential component of precise program analysis for accurately modeling pointer behaviors by incorporating control flows. Flow-sensitive pointer analysis is extensively used in alias analysis, taint analysis, program understanding, compiler optimization, etc. Existing flow-sensitive pointer analysis approaches, which are conducted based on control flow graphs, have significantly advanced the precision of pointer analysis via sophisticated techniques to leverage control flow information. However, they inevitably suffer from computational inefficiencies when resolving points-to information due to the inherent complex structures of control flow graphs.

We present CG-FsPTA, a *Flow-Sensitive Constraint Graph* (FSConsG) based flow-sensitive pointer analysis to overcome the inefficiency of control-flow-graph-based analysis. CG-FsPTA uses a flow-sensitive variant to leverage the structural advantages of set-constraint graphs (which are commonly used in flow-insensitive pointer analysis) while keeping the flow sensitivity of variable definitions and uses, allowing the incorporation of sophisticated graph optimization and dynamic solving techniques. In this way, CG-FsPTA achieves significant efficiency improvements while keeping the precision of flow-sensitive analysis. Experimental evaluations on benchmark programs demonstrate that CG-FsPTA, which leverages the FSConsG to simplify graph structure and significantly reduces both memory usage and execution time while maintaining precision. In particular, by solving in the FSConsG, CG-FsPTA achieves an average memory reduction of 33.05% and accelerates flow-sensitive pointer analysis by 7.27× compared to the state-of-art flow-sensitive pointer analysis method. These experimental results underscore the efficacy of CG-FsPTA as a scalable solution to analyze large-scale software systems, thus establishing a robust foundation for future advancements in efficient program analysis frameworks.

1. Introduction

Pointer analysis determines the potential memory locations a pointer may reference during runtime, serving as a foundational component of static program analysis. This analysis enables numerous applications across software security Sui, Ye and Xue (2014); Fink, Yahav, Dor, Ramalingam and Geay (2008); Yan, Sui, Chen and Xue (2018); Shi, Yao, Wu and Zhang (2021); Sui, Ye and Xue (2012); Machiry, Spensky, Corina, Stephens, Kruegel and Vigna (2017); Li, Bai, Sui and Hu (2022); Guo, Yao and Zhang (2024), program verification Lee, Yang and Yi (2005); Das, Lerner and Seigle (2002); Cheng, Ren and Sui (2024), and compiler optimization Rountev and Chandra (2000); Hardekopf and Lin (2007b). Enhancing the efficiency and precision of pointer analysis Andersen (1994); Hardekopf and Lin (2007a); Pearce, Kelly and Hankin (2007a); Sui et al. (2014); Tan, Li, Ma, Xu and Smaragdakis (2021); Zuo, Zhang, Pan, Lu, Li, Wang, Li and Xu (2021) represents a long-standing research challenge, as its effectiveness directly influences the quality of downstream static analysis tasks.

The performance of pointer analysis depends significantly on the analysis approach and the *underlying graph*

representation. Flow-insensitive pointer analysis, which disregards control flow and execution order, achieves exceptional performance due to its simplicity and reduced computational demands Hardekopf and Lin (2007a); Pereira and Berlin (2009). It typically operates on a *constraint graph* (ConsG), where each node represents a single variable, and each edge encodes a constraint between two variables. This representation is efficiently solved using Andersen-style algorithms Andersen (1994); Hardekopf and Lin (2007a), which have been the subject of extensive optimization research for decades Pereira and Berlin (2009); Lei and Sui (2019a); Liu, Li, Swain and Huang (2022). However, this computational efficiency comes at the cost of precision, as flow-insensitive analysis produces overly conservative points-to sets by aggregating all possible program states without considering execution order, frequently resulting in spurious points-to relationships Yu, Xue, Huo, Feng and Zhang (2010); Sui et al. (2014). These imprecisions, inherent to the lack of flow sensitivity, render such analyses inadequate for applications requiring detailed modeling of program execution sequences. Conversely, flow-sensitive pointer analysis provides substantially higher precision by accounting for the sequential execution order of program statements. By tracking value changes at specific program points, flow-sensitive pointer analysis enables precise updates of points-to information. This enhanced precision makes flow-sensitive analysis particularly valuable in modern program analysis scenarios, especially for tasks

*Corresponding author

✉ jiahao.zhang6@unswalumni.com (J. Zhang); xiao.cheng@unsw.edu.au (X. Cheng); youthyounggray@gmail.com (Y. Lei)

ORCID(s):

¹These authors contributed equally to this work.

where correctness and detailed behavioral modeling are critical Sui, Di and Xue (2016); Hardekopf and Lin (2011).

In general, flow-sensitive pointer analysis is conducted based on a *control flow graph* (CFG), techniques including semi-sparse approach Hardekopf and Lin (2009), level-by-level analysis Yu et al. (2010), and the sparse value-flow graph (SVFG) Sui et al. (2016). These approaches enhance pointer analysis precision by selectively propagating points-to information or carefully partitioning the program into segments. Although semi-sparse techniques Hardekopf and Lin (2009) reduce redundant computations through data-dependency management and selective updates, they frequently suffer from scalability issues. Similarly, the SVFG achieves flow-sensitivity by maintaining sparse def-use chains generated from the integration of a CFG and the result of a flow-insensitive pointer analysis, which is essentially a simplified CFG. Consequently, solving the SVFG involves both processing statements in the nodes and propagating values along the edges. The overlapped variables in adjacent nodes lead to repeated computation and thus cause inefficiency. Furthermore, since the graphs of the current flow-sensitive pointer analysis techniques bundle program statements into the nodes, existing state-of-the-art graph simplification Hardekopf and Lin (2007a); Rountev and Chandra (2000); Lei, Sui, Tan and Zhang (2023) and dynamic solving techniques Pereira and Berlin (2009); Liu et al. (2022) are not applicable.

This observation motivates a fundamental research question: *can we synthesize the precision of flow-sensitive pointer analysis with the computational efficiency of Andersen-style flow-insensitive analysis?* To address this challenge, we introduce CG-FSPTA, a novel flow-sensitive pointer analysis approach based on the *Flow-Sensitive Constraint Graph* (FSConsG), which effectively bridges the gap between efficient Andersen-style constraint-based analysis and precise flow-sensitive pointer analysis. Unlike existing CFG-based techniques such as SVFG Sui et al. (2016), where nodes encapsulate statements potentially containing multiple variables, thereby preventing the application of powerful optimization techniques developed for Andersen-style pointer analysis, our FSConsG maintains the elegant simplicity of ConsG structures: each node represents exactly one variable, with no embedded control flow information. This critical insight—separating variable representation from control flow—yields a significantly cleaner and more computationally efficient graph representation compared to traditional flow-sensitive approaches. Our FSConsG is inherently compatible with the state-of-the-art graph optimization techniques, including variable substitution Rountev and Chandra (2000) and cycle elimination Tarjan (1972), as well as efficient constraint solvers like WAVE Pereira and Berlin (2009) and SFR Lei and Sui (2019b). Consequently, CG-FSPTA substantially reduces computational overhead while maintaining the precision characteristics of flow-sensitive analysis, demonstrating its potential as a scalable and effective solution for current program analysis challenges.

The construction of an FSConsG requires the information of a CFG and the result of a flow-insensitive pointer analysis, similar to the SVFG used in existing techniques VSFS Barbar, Sui and Chen (2021); Hardekopf and Lin (2009); Sui and Xue (2017). The essential difference is that the graph we used is constructed in the form of ConsG, i.e., each variable is kept as a node, and each edge represents an instruction. In particular, in an FSConsG, we separate address-taken variables (abstract memory objects) into versions according to the execution points (CFG nodes) and use def-use relations (calculated by the approach proposed in Hardekopf and Lin (2009)) to construct a series of def-use chains of the address-taken variables, maintaining the flow-sensitivity of address-taken variables. In this form, we are able to maintain a points-to set for each node and solve points-to relations by processing only the edges. Since our pointer analysis is conducted on LLVM's intermediate representation (LLVM-IR), where top-level variables (pointers) are already field-sensitive (in an SSA form), our FSConsG is sufficient to perform flow-sensitive pointer analysis. Based on the FSConsG, we propose a flow-sensitive constraint solver. With specific mechanisms for solving address-taken variables, the solver guarantees its soundness and precision by outputting points-to-set results identical to that of existing techniques, e.g., SFS and the semi-sparse approach Hardekopf and Lin (2009).

Our CG-FSPTA integrates flow-insensitive and flow-sensitive pointer analyses into the same abstract program model, simplifying the graph representation of flow-sensitive pointer analysis, and uniquely boosting the speed of flow-sensitive pointer analysis by minimizing computational overhead, with the incorporation of powerful graph simplification and dynamic solving techniques.

We implemented our CG-FSPTA and evaluated its scalability and efficiency by comparing it to the state-of-the-art VSFS. Integrated into the LLVM-16 compiler infrastructure and evaluated using the SPEC CPU 2017 benchmark suite, CG-FSPTA consistently outperforms VSFS in the number of constraints within the graph, memory usage, and execution time. These improvements stem from effectively reducing the number of nodes and edges in the FSConsG without sacrificing analytical precision. Moreover, the constraint-solving algorithm optimizes memory usage and accelerates execution, enabling CG-FSPTA to achieve significantly faster average execution compared to VSFS.

Our key contributions are as follows:

- We integrate flow-insensitive and flow-sensitive pointer analyses into the same program model via flow-sensitive constraint graph (FSConsG), reducing the graph complexity of flow-sensitive pointer analyses and making flow-sensitive pointer analyses well compatible with the state-of-the-art graph simplification and constraint solving techniques.
- Based on the FSConsG, we propose a constraint-solving algorithm with specific mechanisms to handle address-taken variables. By incorporating with

the state-of-the-art techniques, the solver significant boosts the efficiency of flow-sensitive pointer analysis.

- We integrated CG-FSPtA—including both the construction of FSConsGs and the constraint solver—into the LLVM-16 infrastructure and conducted a comprehensive evaluation based on benchmark programs. The experimental results indicate that, compared to VSFS, CG-FSPtA significantly enhances efficiency by reducing memory consumption, achieving an average reduction of 33.05%, and demonstrating an average execution time that is 7.27× faster. These results highlight its effectiveness and scalability in analyzing modern C/C++ programs.

The remainder of this paper is organized as follows: Section 2 provides the necessary background on pointer analysis, detailing the methodologies of set constraint-based analysis, flow-insensitive analysis, and flow-sensitive analysis. Section 3 explains the motivation for developing the CG-FSPtA, addressing the limitations of existing methods. Sections 4 and 5 present our approach, describing the construction of FSConsG and the solver of CG-FSPtA, as well as the application of graph optimization techniques such as graph folding. Section 6 evaluates our approach using experiments, comparing CG-FSPtA with the state-of-the-art method in terms of graph structure, memory usage, and execution time. Section 7 discusses related work, situating our contributions in the context of prior research. Finally, Section 8 concludes with a summary of findings and potential directions for future work.

2. Background and Problem Formulation

This section presents the foundational concepts underpinning our approach. We begin by describing the target language and introducing set-constraint based pointer analysis. We then compare flow-insensitive (constraint graph-based) and flow-sensitive (control-flow graph-based) pointer analysis methodologies, highlighting their respective advantages and limitations. Finally, we formulate the central problem addressed in this work: how to leverage the computational efficiency of Andersen’s pointer analysis to resolve flow-sensitive analysis challenges without sacrificing the precision benefits inherent to flow-sensitivity.

2.1. Analysis Domains, LLVM Instructions and Constraints

Following previous works Hardekopf and Lin (2007a); Sui and Xue (2016a); Lei and Sui (2019a), we conduct our pointer analysis on an LLVM-like intermediate representation Lattner and Adve (2004). Table 1 presents the analysis domains and LLVM instructions relevant to flow-sensitive pointer analysis. This intermediate representation employs a partial static single assignment (SSA) form, wherein the set of all program variables $\mathcal{V}$ is partitioned into two disjoint subsets: *address-taken variables* $\mathcal{O}$, comprising all abstract memory objects and their fields, and *top-level variables* $\mathcal{P}$,

consisting of stack virtual registers ($\mathcal{S}$, denoted by symbols prefixed with “%”) and global variables ($\mathcal{G}$, denoted by symbols prefixed with “@”). Top-level variables are represented in explicit SSA form, allowing direct access and manipulation. In contrast, address-taken variables in $\mathcal{O}$ are maintained in non-SSA form and can only be accessed indirectly through top-level variables via Load or Store instructions.

A key consequence of this design is that there are no direct assignments between address-taken variables in LLVM-IR. Any source-level assignment to an address-taken variable is compiled into a store instruction through a top-level pointer, and any read from an address-taken variable is compiled into a load instruction. Furthermore, multi-level pointer operations in the source language (e.g., `**p`) are decomposed by the compiler into sequences of single-level load/store instructions, each operating through a top-level temporary in SSA form. This two-level design ensures that every pointer dereference in LLVM-IR involves exactly one level of indirection.

The C-like representation of LLVM instructions and their corresponding constraint edges are presented in Table 1 (right side). The Addr constraint establishes the fundamental mapping between pointers and their referenced memory objects, initializing the basis for all pointer relationships. The Copy constraint represents direct assignment between top-level variables, enabling value propagation through the program. For memory access operations, Load and Store constraints capture the bidirectional semantics of retrieving values from and writing values to memory locations via pointers, respectively. To support structured data types, the Field constraint enables field-sensitive analysis by facilitating precise differentiation between distinct fields of compound objects, thus enhancing analysis precision.

2.2. Set-Constraint Based Pointer Analysis

Set-constraint based pointer analysis is a widely-used formulation of inclusion-based pointer analysis, exemplified by Andersen’s algorithm Andersen (1994). In this approach, each pointer variable is associated with a points-to set that conservatively represents all possible memory locations it may reference throughout program execution. The pointer operations in the program are systematically encoded as set constraints, as illustrated in Table 2, which are then solved iteratively to derive a sound approximation of the program’s pointer relationships.

The five constraint types presented in Table 2 formalize the core semantics of pointer operations, establishing the foundation for solving points-to analysis problems. The Addr constraint establishes the fundamental mapping between pointer variables and their referenced memory objects. The Copy constraint enables the propagation of points-to relationships through direct assignments between variables. For memory operations, the Load and Store constraints operate by dereferencing the pointer operand and establishing a Copy constraint between the referenced object and the corresponding value operand. Additionally, the GEP (GetElementPtr)

Table 1

Analysis domains, LLVM instructions, and constraint edges.

Analysis Domains			Instruction	Constraint	Type
ℓ	$\in \mathcal{L}$	Statements	$p = \&o$	$p \xleftarrow{\text{Addr}} o$	Addr
i, j, w	$\in \mathcal{C}$	Integer constants	$p = q$	$p \xleftarrow{\text{Copy}} q$	Copy
s	$\in \mathcal{S}$	Stack virtual registers	$p = \&q \rightarrow \text{fld}$	$p \xleftarrow{\text{Gep, fld}} q$	Gep
g	$\in \mathcal{G}$	Global variables	$p = *q$	$p \xleftarrow{\text{Load}} q$	Load
f	$\in \mathcal{F} \subseteq \mathcal{G}$	Program functions	$*p = q$	$p \xleftarrow{\text{Store}} q$	Store
p, q, r, x, y	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	Top-level variables			
$o, a, b, c, o, f_i \in \mathcal{O}$		Address-taken variables			
u, v	$\in \mathcal{V} = \mathcal{O} \cup \mathcal{P}$	Program variables			

Table 2

Set constraint-based points-to analysis rules.

Instruction	Constraint Type	Set Constraints
$p = \&o$	$p \xleftarrow{\text{Addr}} o$	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
$p = q$	$p \xleftarrow{\text{Copy}} q$	$\text{pts}(p) = \text{pts}(q) \cup \text{pts}(p)$
$p = \&q \rightarrow \text{fld}$	$p \xleftarrow{\text{Gep, fld}} q$	for each $o \in \text{pts}(q) : \text{pts}(p) = \text{pts}(p) \cup \{o.\text{fld}\}$
$p = *q$	$p \xleftarrow{\text{Load}} q$	for each $o \in \text{pts}(q) : \text{add } p \xleftarrow{\text{Copy}} o$
$*p = q$	$p \xleftarrow{\text{Store}} q$	for each $o \in \text{pts}(p) : \text{add } o \xleftarrow{\text{Copy}} q$

constraint facilitates field-sensitive analysis by precisely distinguishing between different fields of structured objects. When these constraints are applied iteratively within a set-constraint framework, they collectively construct a comprehensive model of the program's pointer relationships, which is essential for conducting precise and effective pointer analysis.

2.3. Flow-Insensitive vs. Flow-Sensitive Pointer Analysis

Flow-insensitive pointer analysis (FI-PTA) computes a single points-to solution that remains invariant across all program points. This approach offers computational efficiency by abstracting away control flow considerations, enabling significant optimizations such as efficient propagation techniques Pereira and Berlin (2009). However, this efficiency comes at the cost of precision, as the analysis frequently produces over-approximated points-to sets that can lead to false positives in downstream analyses Hardekopf and Lin (2007a). The implementation of FI-PTA typically employs a *set-constraint graph* representation, where program variables form the nodes of the graph, and the different constraint types enumerated in Table 2 manifest as edges between these nodes. This graph-based formulation transforms pointer analysis into a constraint-solving problem that can be efficiently resolved through iterative techniques.

Example 1 (FI-PTA). Figure 1(a) presents a simple C program where top-level pointers p , q , and x point to a , b , and o respectively through Addr operations at program points ℓ_3 , ℓ_4 , and ℓ_5 . Subsequently, the program stores p to the memory location referenced by x at ℓ_7 and loads from this location into y at ℓ_8 . This sequence is followed by storing

q to the same memory location at ℓ_9 and loading the value into z at ℓ_{10} .

The flow-insensitive set-constraint graph corresponding to this code is illustrated in Figure 1(b). The analysis proceeds by iteratively applying the inference rules defined in Table 2 to propagate points-to information throughout the graph without considering different program points. During constraint resolution, the black copy edges are dynamically added to reflect the propagation paths until a fixed point is reached—where no further changes to any points-to set occur. This approach, while computationally efficient, results in an over-approximation of the points-to sets for variables o , y , and z , which are conservatively computed to contain both a and b without regard to the temporal ordering of assignments at different program points.

Flow-sensitive pointer analysis (FS-PTA) addresses the limitations of flow-insensitive approaches by considering the control flow and execution order of the program. Unlike FI-PTA, which computes a single points-to set for each variable across the entire program, FS-PTA maintains distinct points-to information at each program point, thereby respecting the sequence of instructions and enabling more precise modeling of pointer behaviors.

Hardekopf and Lin Hardekopf and Lin (2011) introduce an efficient staged FS-PTA that leverages a preliminary conservative pointer analysis to construct a *sparse value-flow graph* (SVFG). This graph, derived from the control-flow graph (CFG) of LLVM-IR, captures def-use relations between variable definitions through building interprocedural memory SSA Chow, Chan, Liu, Lo and Streich (1996); Hardekopf and Lin (2011); Sui and Xue (2016b), serving as a foundation for subsequent, more precise flow-sensitive analysis by representing data dependencies that respect CFG

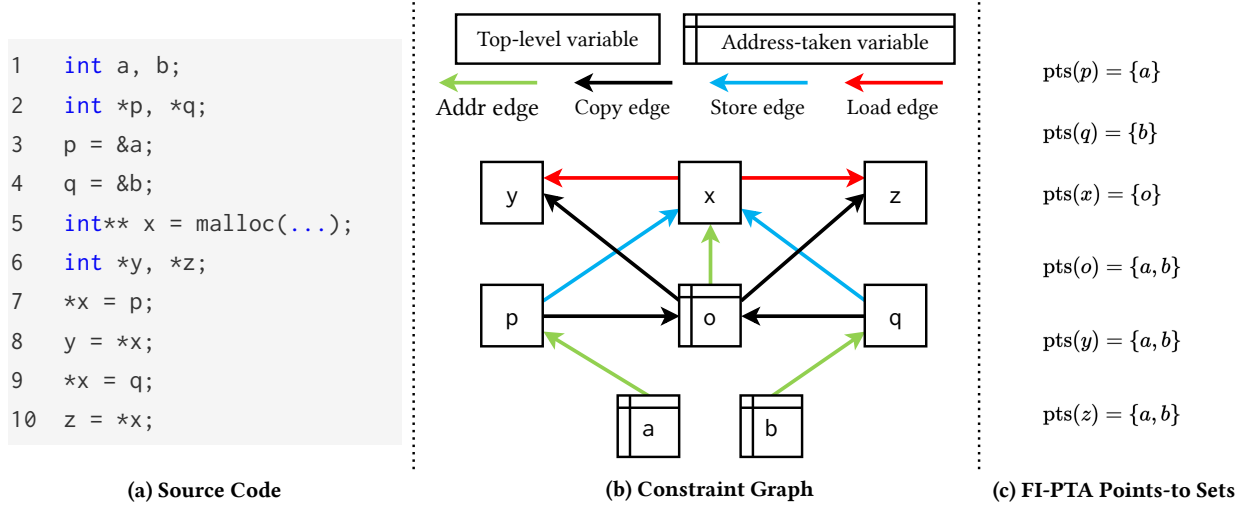
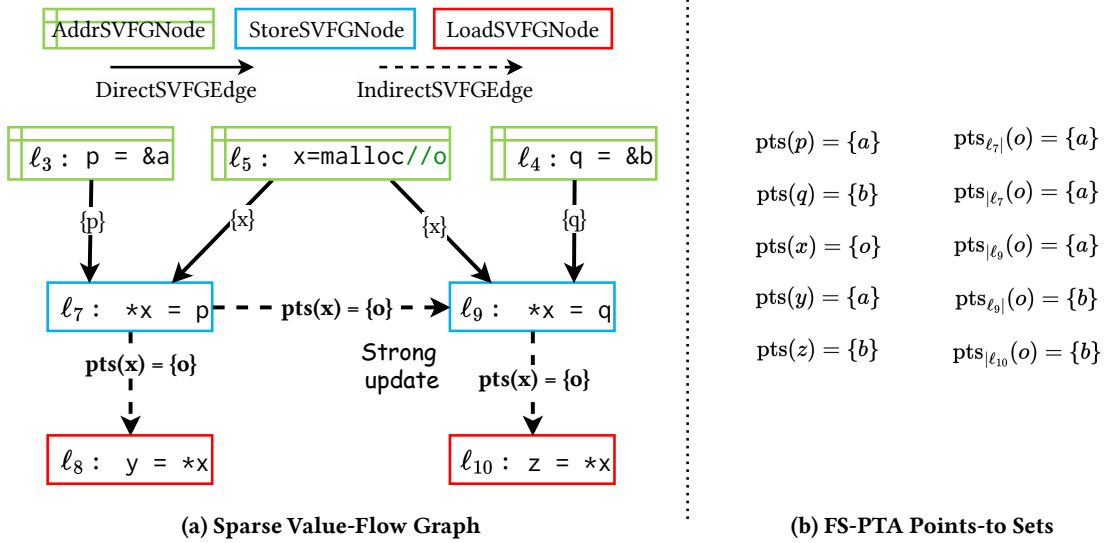


Figure 1: Flow-insensitive (constraint graph-based) pointer analysis.


 Figure 2: Flow-sensitive (sparse value-flow graph-based) pointer analysis. $\text{pts}_{|\ell_i|}(p)$ and $\text{pts}_{\ell_i}(p)$ represent the points-to set of p immediately before and after ℓ_i , respectively.

execution order. During the flow-sensitive analysis phase, the approach maintains program-point-specific points-to sets for address-taken variables in non-SSA form, while top-level variables, already in SSA form, can be tracked globally without sacrificing precision.

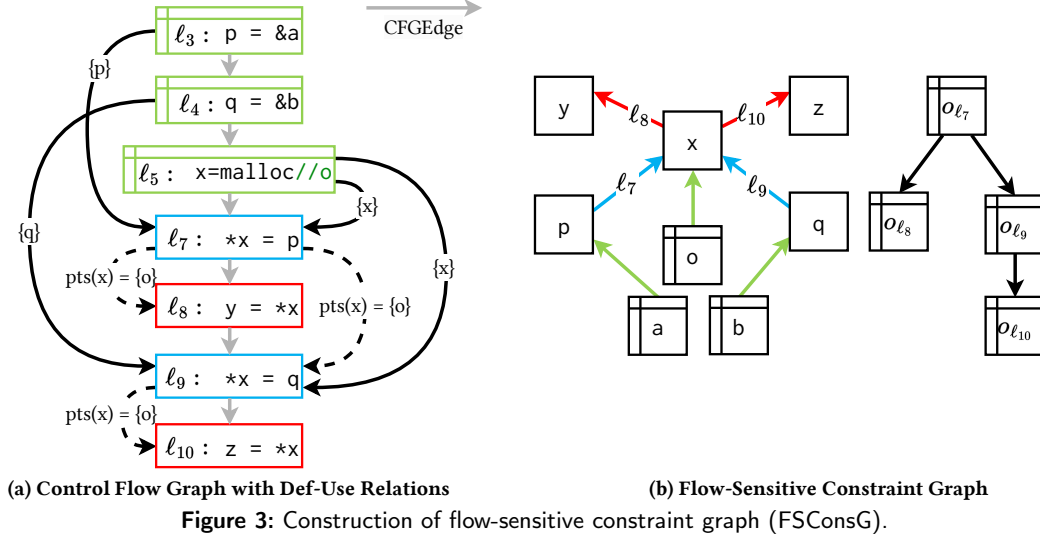
Example 2 (FS-PTA). Revisiting the code in Figure 1, the SVFG in Figure 2 precisely captures value flows reflecting execution order. The value flow of x originates from $\ell_5 : x = \text{malloc}$, establishing $x \rightarrow o$, and propagates to statements $\ell_7 : *x = p$ and $\ell_9 : *x = q$. Similarly, p and q have value flows originating at $\ell_3 : p = \&a$ and $\ell_4 : q = \&b$, respectively, which propagate to the corresponding store operations.

For the address-taken variable o , three critical indirect value flows are captured: (1) from $\ell_7 : *x = p$ to $\ell_8 : y = *x$, reflecting y 's inheritance of p 's value; (2) from $\ell_9 : *x = q$ to $\ell_{10} : z = *x$, indicating z 's acquisition of q 's value; and (3) from $\ell_7 : *x = p$ to $\ell_9 : *x = q$, capturing the execution

sequence between these statements. These def-use chains accurately model data flow at each program point.

By respecting execution order, FS-PTA precisely determines the different o 's points-to set during program execution. We use the notation $\text{pts}_{|\ell_i|}$ and pts_{ℓ_i} to denote the points-to sets flowing into and out of label ℓ_i , respectively. The analysis computes $\text{pts}_{|\ell_7|}(o) = \{a\}$, $\text{pts}_{\ell_7}(o) = \{a\}$ and $\text{pts}_{|\ell_9|}(o) = \{a\}$. After the strong update at ℓ_9 , we have $\text{pts}_{\ell_9}(o) = \{b\}$ and $\text{pts}_{|\ell_{10}|}(o) = \{b\}$. This precision is beyond the capability of FI-PTA, which conservatively merges all points-to information into a single points-to set $\{a, b\}$ for o across all program points. Note that the value $\{a\}$ does not persist to ℓ_{10} due to the strong update at ℓ_9 that overwrites the previous value.

SVFG-based FS-PTA achieves superior precision compared to set-constraint graph-based FI-PTA by maintaining distinct points-to sets at each program point. However, this precision improvement incurs computational overhead.



Moreover, FS-PTA cannot fully exploit the algorithmic optimizations that have made FI-PTA computationally efficient, which are specifically tailored for set-constraint graph representations. This limitation creates a tension between precision and performance, motivating the development of an approach that effectively combines the precision advantages of FS-PTA with the computational efficiency of FI-PTA. Such an approach would enable scalable yet precise pointer analysis.

2.4. Problem Formulation

The core problem addressed in this work is to develop a novel approach to flow-sensitive pointer analysis that leverages the computational efficiency of set-constraint graph-based flow-insensitive analysis. Specifically, we aim to:

1. Design a new set-constraint graph representation, called *FSConsG*, that encodes flow-sensitivity within a set-constraint graph framework, preserving program point-specific pointer information without requiring an explicit control flow graph structure.
2. Formulate a constraint resolution algorithm that maintains the precision benefits of flow-sensitive analysis — particularly the ability to distinguish points-to sets of address-taken variables at different program points — while utilizing efficient propagation techniques traditionally associated with flow-insensitive analysis.

3. Motivating Example

This section illustrates our CG-FSPTA by revisiting the example in Figure 1, together with elucidating our approach and discussing its benefits compared to traditional SVFG-based FS-PTA techniques Hardekopf and Lin (2011); Sui and Xue (2018).

Construction of FSConsG. Figure 3 illustrates how CG-FSPTA constructs the flow-sensitive constraint graph (FSConsG), depicted in Figure 3(b), based on the program's control flow graph (CFG) and pre-computed direct and indirect def-use relations, as shown in Figure 3(a). The direct

def-use relations between top-level variables are already explicitly defined in the program, thus it is unnecessary to encode this information in the FSConsG. The indirect def-use relations are derived from memory SSA forms constructed upon a preliminary pointer analysis. For each indirect relation, we augment the corresponding constraint edges (loads and stores) in the FSConsG with control flow information, generate versioned address-taken variables that correspond to dereferenced objects at these loads and stores, and establish copy edges between these versioned variables according to the pre-computed indirect def-use chains.

For instance, when processing the indirect def-use chain $\ell_7 \xrightarrow{\{o\}} \ell_8$, we consider the Store statement $\ell_7 : *x = p$ and the Load statement $\ell_8 : y = *x$. We annotate the constraint edge $p \xrightarrow{\text{Store}, \ell_7} x$ with the control flow information from ℓ_7 and the constraint edge $x \xrightarrow{\text{Load}, \ell_8} y$ with ℓ_8 . Based on the pre-computed points-to information $\text{pts}(x) = \{o\}$, we introduce a versioned address-taken variable o_{ℓ_7} to represent the memory location dereferenced at ℓ_7 and another versioned object o_{ℓ_8} to represent ℓ_8 . Subsequently, to capture the indirect def-use relation $\ell_7 \xrightarrow{\{o\}} \ell_8$, we incorporate a Copy constraint $o_{\ell_7} \xrightarrow{\text{Copy}} o_{\ell_8}$ into the FSConsG.

Analogously, for the indirect value-flow from ℓ_9 to ℓ_{10} (i.e., $\ell_9 \xrightarrow{\{o\}} \ell_{10}$), we establish constraint edges $q \xrightarrow{\text{Store}, \ell_9} x$ and $x \xrightarrow{\text{Load}, \ell_{10}} z$, and add a Copy constraint $o_{\ell_9} \xrightarrow{\text{Copy}} o_{\ell_{10}}$ to the FSConsG. Furthermore, to capture the indirect def-use chain $\ell_7 \xrightarrow{\{o\}} \ell_9$, we incorporate the copy constraint $o_{\ell_7} \xrightarrow{\text{Copy}} o_{\ell_9}$ in the FSConsG.

By incorporating these versioned address-taken variables and establishing copy constraints between them according to indirect def-use chains, CG-FSPTA effectively encapsulates the flow-sensitive semantics of address-taken variables at critical program points. This approach enables flow-sensitive pointer analysis to be performed using traditional constraint-graph-based techniques while preserving

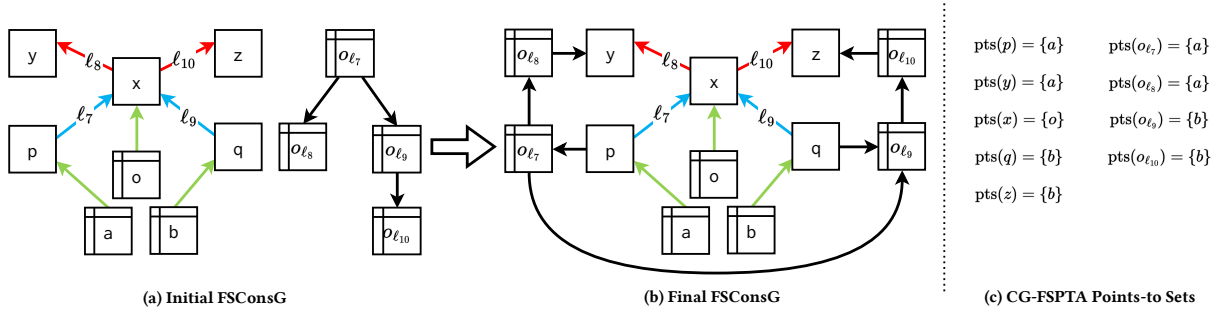


Figure 4: Flow-sensitive pointer analysis on FSConsG.

the precision benefits that flow-sensitivity offers. The resulting FSConsG thus serves as a unified representation that elegantly captures both points-to relations and their flow-sensitive properties in a single constraint framework.

FS-PTA on FSConsG. Figure 4 illustrates the process of flow-sensitive pointer analysis on FSConsG, which employs constraint solving according to the rules defined in Table 2. The figure depicts the updated FSConsG with newly added Copy edges and the points-to sets associated with each node. The analysis demonstrates that the resolved points-to sets of y and z yield the more precise flow-sensitive values $\{a\}$ and $\{b\}$ respectively, in contrast to the flow-insensitive result $\{a, b\}$ obtained via FI-PTA. This enhanced precision stems from CG-FSPTA’s ability to differentiate the indirect value flows from o_{l_7} to o_{l_8} and from o_{l_9} to $o_{l_{10}}$, thereby resolving the points-to sets of y and z with greater precision. Note that the points-to set of o_{l_7} does not propagate to o_{l_9} due to the strong update semantics Lhoták and Chung (2011) applied to the Copy constraint.

Benefits. The FSConsG representation offers efficient structural simplification compared to a traditional SVFG. For instance, while the variable x appears in multiple program statements ($\ell_5, \ell_7, \ell_8, \ell_9, \ell_{10}$) in the SVFG, it corresponds to exactly one node in the FSConsG. More importantly, FSConsG represents constraints solely through edges, whereas SVFG requires maintaining constraints on both nodes and edges. This design choice reduces the redundant number of constraints to be maintained during analysis, enabling our approach to achieve the precision of flow-sensitive pointer analysis with the computational efficiency of constraint graph-based techniques Pereira and Berlin (2009); Lei and Sui (2019a).

4. Flow-Sensitive Constraint Graph (FSConsG)

This section presents the definition and construction method of our Flow-Sensitive Constraint Graph (FSConsG).

4.1. Formal Definition of FSConsG

A Flow-Sensitive Constraint Graph, FSConsG, is formally defined as a directed graph $G_{FS} = (\mathcal{V}_{FS}, E_{FS})$. We next detail the nodes and edges of FSConsG.

Set of Nodes ($\mathcal{V}_{FS}$) The node set $\mathcal{V}_{FS}$ consists of top-level variables and program-point-versioned address-taken variables: $\mathcal{V}_{FS} = \mathcal{P} \cup \mathcal{O}_{\mathcal{L}}$.

Here, $\mathcal{P}$ denotes the set of top-level pointer variables represented as SSA values. Let $\mathcal{O}$ be the set of abstract memory objects and $\mathcal{L}$ the set of program points (CFG node identifiers). We define a partial versioning function

$$v : \mathcal{O} \times \mathcal{L} \rightarrow \mathcal{O}_{\mathcal{L}},$$

with a domain characterized as

$$v(o, \ell) \text{ is defined } \iff \ell \in \text{IndDef}(o) \cup \text{IndUse}(o),$$

where $\text{IndDef}(o)$ and $\text{IndUse}(o)$ denote the sets of program points at which object o is indirectly defined and indirectly used, respectively. When defined, $v(o, \ell) = o_{\ell}$ denotes the flow-sensitive version of object o at program point ℓ .

The set of flow-sensitive object nodes is

$$\mathcal{O}_{\mathcal{L}} \triangleq \{ o_{\ell} \mid v(o, \ell) \text{ is defined} \}.$$

Specifically, for each indirect def–use edge

$$\ell_i \xrightarrow{\text{pts}(o)} \ell_j,$$

where $\ell_i \in \text{IndDef}(o)$ and $\ell_j \in \text{IndUse}(o)$, we create two FSConsG nodes o_{ℓ_i} and o_{ℓ_j} . Here, o_{ℓ_i} represents the definition-site version of o , whereas o_{ℓ_j} represents the corresponding use-site version.

Set of Edges (E_{FS}) Similar to flow-insensitive constraint graphs, an FSConsG also contains the five basic types of constraint edges, i.e., Addr, Copy, Store, Load, and Gep. The difference is that each edge is annotated with an index ℓ , denoting the execution points of the represented statement in the program. In particular, there are exemptions for Addr, Copy and Gep edges, which are detailed in Section 4.2.1.

Remark (Handling arbitrary pointer levels). Although the above definition focuses on single-level dereferences in Load and Store edges, FSConsG naturally supports double and higher-level pointers. As described in Section 2.1, LLVM-IR decomposes multi-level pointer operations into sequences of single-level load/store

instructions, each involving a top-level variable (represented as a single SSA node in $\mathcal{P}$) and an address-taken variable (versioned in $\mathcal{O}_L$ at the corresponding program points). Each such instruction maps to exactly one FSConsG edge. Consequently, the FSConsG framework uniformly handles arbitrary levels of pointer indirection without any special treatment.

4.2. Construction of FSConsG

Our flow-sensitive constraint graph is constructed on top of the result of the conservative flow-insensitive pointer analysis Lei and Sui (2019a) and a control flow graph (CFG) Hardekopf and Lin (2011), constructed in preprocessing. In particular, in the CFG, we follow the technique of Sui et al. (2016) to compute the def-use relations of variables.

4.2.1. Converting Program Statements

The first step of constructing an FSConsG is to convert the statements in the CFG nodes into FSConsG edges. This establishes the basic structure of our FSConsG, a digraph describing the operations of top-level variables via edges. Since the flow-insensitive analysis and the construction of CFG in the preprocessing stage are both conducted based on the LLVM-IR, top-level variables are already in the SSA form. Moreover, the statements of CFG nodes involve only top-level variables, except for the Addr statements, which allocate an abstract memory object to a top-level variable. The rules of converting the five statements, involving top-level operations, into edges are displayed in Figure 5. The following part details each conversion rule.

Address The Addr CFG nodes capture LLVM-IR's `alloca` instructions from abstract memory objects to top-level variables. We use the rule BuildAddr in Figure 5 to facilitate the transformation from Addr CFG nodes to Addr edges in FSConsG. Specifically, a Addr statement $\ell_i : p = \&o$, which captures an LLVM-IR's `alloca` instruction, is converted into two nodes p and o , and an edge $o \xrightarrow{\text{Addr}} p$ in the FSConsG.

Copy The Copy CFG nodes capture LLVM-IR's `copy` instructions. We use the rule BuildCopy in Figure 5 to facilitate the transformation from Copy CFG nodes to Copy edges in FSConsG. Specifically, a CFG node containing a Copy statement $\ell_i : q = p$, which captures an LLVM-IR's `copy` instruction, is converted into two nodes p and q , and an edge $p \xrightarrow{\text{Copy}} q$ in the FSConsG.

Gep For field-sensitive analysis, the Gep CFG nodes capture LLVM-IR's `getelementptr` instructions about field access operations. We use the rule BuildGep in Figure 5 to facilitate the transformation from Gep CFG nodes to Gep edges in FSConsG. Specifically, a Gep statement $\ell_i : q = \&p.\text{fld}$, which captures an LLVM-IR's `getelementptr` instruction, is converted into two nodes p and q , and an edge $p \xrightarrow{\text{Gep.fld}} q$ in the FSConsG.

Store LLVM-IR uses indirect instructions `Store` and `Load` to module the accesses and operations of address-taken

variables. The Store nodes in a CFG capture LLVM-IR's store instructions. We use the rule BuildStore in Figure 5 to facilitate the transformation from Store CFG nodes to Store edges in FSConsG. Specifically, a Store statement $\ell_i : *q = p$, which captures an LLVM-IR's store instruction, is converted into two nodes p and q , and an edge $p \xrightarrow{\text{Store}, \ell_i} q$ in the FSConsG.

Load The Load nodes in a CFG capture LLVM-IR's load instructions. We use the rule BuildLoad in Figure 5 to facilitate the transformation from Load CFG nodes to Load edges in FSConsG. Specifically, a Load statement $\ell_i : q = *p$, which captures an LLVM-IR's load instruction, is converted into two nodes p and q , and an edge $p \xrightarrow{\text{Load}, \ell_i} q$ in the FSConsG.

Maintaining control flow information Note that in the rules of Figure 5, each FSConsG edge is also annotated with ℓ_i , the index of the converted CFG node (e.g., $p \xrightarrow{\text{Store}, \ell_i} q$). We use this annotation to capture the execution order manifested in CFG nodes and preserve the flow sensitivity of top-level operations. This is imperative in flow-sensitive constraint solving on FSConsG, which is detailed in Section 5. Notably, the top-level variables are already in the SSA form in LLVM-IR, which means that the statements involving only the points-to sets of top-level variables (i.e., Addr and Copy) in the CFG are inherently flow-sensitive. Thus, we do not need to maintain the corresponding CFG node indices for Addr and Copy edges in an FSConsG.

4.2.2. Maintaining Def-Use Relations

The methods of Section 4.2.1 convert a CFG into a set-constraint graph consisting of discrete edges. To make a complete FSConsG, we further correlate the discrete edges via def-use relations, producing a connected graph while maintaining flow-sensitivity.

In a CFG, a def-use edge between two nodes denotes that some common variable(s) are defined in a program point while redefined/used in another program point. Def-use edges include direct ones (the def-use relations of top-level variables) and indirect ones (the def-use relations of address-taken variables), and can be calculated by integrating the result of flow-insensitive preanalysis into the CFG Sui et al. (2016).

For example, Figure 3(a) is a CFG with def-use, the gray edges denote control flows, and the solid and dashed black edges denote direct and indirect def-use edges, respectively. The direct def-use edge from node ℓ_3 to node ℓ_7 , annotated with $\{p\}$, indicates that p is defined at ℓ_3 (via $p = \&a$) and used at ℓ_7 (via $*x = p$). Relatively, the indirect def-use edge from ℓ_7 to ℓ_8 , annotated with $pts(x) = \{o\}$, indicates that o is (re)defined at ℓ_7 (via $*x = p$) and used in ℓ_8 (via $y = *x$). In particular, $pts(x) = \{o\}$ is calculated in the flow-insensitive preanalysis.

Different from an SVFG that is constructed via connecting CFG nodes with def-use edges and removing the

$$\begin{array}{c}
\text{[BUILDADDR]} \frac{\text{CFGNode } \ell_i : p = \&o}{o \xrightarrow{\text{Addr}} p \in E_{FS}} \\
\text{[BUILDCOPY]} \frac{\text{CFGNode } \ell_i : q = p}{p \xrightarrow{\text{Copy}} q \in E_{FS}} \quad \text{[BUILDGEP]} \frac{\text{CFGNode } \ell_i : q = \&p.\text{fld}}{p \xrightarrow{\text{Gep.fld}} q \in E_{FS}} \\
\text{[BUILDSTORE]} \frac{\text{CFGNode } \ell_i : *q = p}{p \xrightarrow{\text{Store}, \ell_i} q \in E_{FS}} \quad \text{[BUILDLLOAD]} \frac{\text{CFGNode } \ell_i : q = *p}{p \xrightarrow{\text{Load}, \ell_i} q \in E_{FS}}
\end{array}$$

Figure 5: Rules for converting top-level operations. Here, $e \in E_{FS}$ denotes adding an edge e to the FSConsG.

original CFG edges, our FSConsG directly embeds the def-use relations into a set-constraint graph. The following details how we handle top-level and address-taken variables, respectively.

Avoiding top-level collisions Collisions of FSConsG nodes will emerge when several CFG nodes involve the same top-level variable. A brute-force approach is to separate the common variable into several versions, corresponding to the CFG nodes containing the variable, then connect the versions with Copy edges based on the control flow. For example, in Figure 3(a), consider the two CFG nodes $\ell_3 : p = \&a$ and $\ell_7 : *x = p$ that are connected by a direct def-use edge annotated with $\{p\}$. The brute-force approach separates p into two versions, i.e., p_1 and p_2 , constructing two edges $a \xrightarrow{\text{Addr}} p_1$ and $p_2 \xrightarrow{\text{Store}, \ell_7} x$, then using a Copy edge to connect p_1 and p_2 , forming a path $a \xrightarrow{\text{Addr}} p_1 \xrightarrow{\text{Copy}} p_2 \xrightarrow{\text{Store}, \ell_7} x$.

We can easily find the redundancy caused by the brute-force method, i.e., $p_1 \xrightarrow{\text{Copy}} p_2$. In fact, in a CFG construct from LLVM-IR, all top-levels are already in the SSA form, hence, they do not need to be separated into versions. Different from the brute-force method, our technique builds a *symbol table* along with the construction process of FSConsG. Specifically, when converting a CFG node ℓ , if a variable v involved in ℓ is not in the symbol table, then record it into the symbol table; otherwise, we directly use v to connect the newly transformed FSConsG edge. Thus, the top-level variables do not need to be separated into versions. Consequently, in the above example, the connected ℓ_3 and ℓ_7 will be transformed into $a \xrightarrow{\text{Addr}} p \xrightarrow{\text{Store}, \ell_7} x$.

Address-taken versioning. Based on the flow-insensitive preanalysis, we create versioned address-taken FSConsG nodes according to the information on the indirect def-use edges. Specifically, for an indirect def-use edge $\ell_i \xrightarrow{\text{pts}(o)} \ell_j$, we create two FSConsG nodes o_{ℓ_i} and o_{ℓ_j} of o , and connect a Copy edge from o_{ℓ_i} to o_{ℓ_j} , indicating that the points-to information of o_i defined in ℓ_i will directly flow to o_j defined in ℓ_j . By transforming the indirect def-use edges one by one, we create a series of *versioned* address-taken nodes that are connected with Copy edges in FSConsG. In this way, preserving the flow sensitivity of address-taken variables via chains of address-taken nodes, and, hence, converting a flow-insensitive constraint graph into a flow-sensitive one.

Example 3 (Address-Taken Versioning). Consider the indirect def-use edge $\ell_7 \xrightarrow{\{o\}} \ell_8$ in Figure 3(a). Two address-taken FSConsG nodes o_{ℓ_7} and o_{ℓ_8} , and a Copy edge $o_{\ell_7} \xrightarrow{\text{Copy}} o_{\ell_8}$ are created and added to the FSConsG. Similarly, $o_{\ell_7} \xrightarrow{\text{Copy}} o_{\ell_9}$ and $o_{\ell_9} \xrightarrow{\text{Copy}} o_{\ell_{10}}$ are created and added to the FSConsG, according to $\ell_7 \xrightarrow{\{o\}} \ell_9$ and $\ell_9 \xrightarrow{\{o\}} \ell_{10}$, respectively, as seen in Figure 3(b).

5. FSConsG-Based Solver

This section introduces an FSConsG-based constraint solver for solving flow-sensitive pointer analysis. The solver is presented in Algorithm 1, which accepts an LLVM-IR and returns the flow-sensitive pointer analysis result, in the form of points-to sets.

5.1. Solver Framework

The solving procedure is comprised of three stages - preprocessing (lines 1), FSConsG construction (lines 2, and points-to-set solving (lines 3–39). The following goes into the details.

The preprocessing stage focuses on three issues: (1) constructing a CFG and a flow-insensitive constraint graph according to the input LLVM-IR; (2) running flow-insensitive pointer analysis on the constraint graph; (3) determining def-use edges on the CFG based on the result of flow-insensitive preanalysis. In particular, we use the Wave solver Pereira and Berlin (2009) to perform the flow-insensitive pointer analysis and Sui et al.'s approach Sui et al. (2016) to determine def-use edges.

The process of constructing FSConsG follows the instructions demonstrated in Section 4.2.

We adopt the staged solving strategy of Pereria et al. Pereira and Berlin (2009); Lei and Sui (2019b) to solve FSConsG. Specifically, the FSConsG solver is initialized with points-to sets calculated from Addr edges (line 3), and an empty worklist (line 4). The solving process (lines 6–5) is conducted under iterations, with points-to-set propagations (lines 8–17) and edge insertions (lines 18–32) categorized into two different stages. The state-of-the-art graph simplification techniques, cycle elimination Nuutila and Soisalon-Soininen (1994); Tarjan (1972) and graph folding Cook and Evans (1979); Lei et al. (2023), are applied to the graph at the beginning of each iteration, as seen in line 7.

By utilizing Nuutila's approach Nuutila and Soisalon-Soininen (1994), the SCC() method can return a topological order of the nodes in the graph, which is used for the following constraint-solving. At the end of each iteration, to handle indirect call relations, our solver updates the call graph (line 33) and decides whether to start a new iteration according to whether the set of edges E is changed (line 35).

Algorithm 1 CG-FsPTA Solver

Require: LLVM-IR

Ensure: Result of flow-sensitive analysis as points-to sets

```

1: Preanalysis: construct CFG, flow-insensitive analysis,
   and def-use edges
2: Construct  $FSConsG\langle V, E \rangle$  based on the CFG with ex-
   plicit def-use edges
3:  $\forall o \xrightarrow{Addr} p \in E_{Addr}: pts(p) \leftarrow \{o\}$ 
4: let  $W$  be a worklist, initially empty
5: repeat
6:    $E_{old} \leftarrow E$ 
7:    $topologicalOrder \leftarrow SCC()$ 
8:   for each  $v \in topologicalOrder$  do
9:      $\forall v \xrightarrow{Copy} w: pts(w) \cup = pts(v);$ 
10:    if  $pts(w)$  changed then
11:       $W \leftarrow W \cup \{w\}$ 
12:    end if
13:     $\forall v \xrightarrow{Gep.fld} w: pts(w) \cup = \{o_{fld} \mid o \in pts(v)\};$ 
14:    if  $pts(w)$  changed then
15:       $W \leftarrow W \cup \{w\}$ 
16:    end if
17:  end for
18:  while  $W \neq \emptyset$  do
19:    select and remove a node  $v$  from  $W$ 
20:    for each  $o \in pts(v)$  do
21:       $\forall u \xrightarrow{Store, \ell_i} v \in E:$ 
22:      if  $IsStrongUpdate()$  then
23:         $pts(o_{\ell_i}) \leftarrow \emptyset; E \cup = \{u \xrightarrow{Copy} o_{\ell_i}\}; W \leftarrow$ 
         $W \cup \{u\}$ 
24:      else if  $u \xrightarrow{Copy} o_{\ell_i} \notin E$  then
25:         $E \cup = \{u \xrightarrow{Copy} o_{\ell_i}\}; W \leftarrow W \cup \{u\}$ 
26:      end if
27:       $\forall v \xrightarrow{Load, \ell_i} w \in E:$ 
28:      if  $o_{\ell_i} \xrightarrow{Copy} w \notin E$  then
29:         $E \cup = \{o_{\ell_i} \xrightarrow{Copy} w\}; W \leftarrow W \cup \{o_{\ell_i}\}$ 
30:      end if
31:    end for
32:  end while
33:  UpdateCallgraph()
34:  if  $E \neq E_{old}$  then
35:     $reanalyze = \text{true}$ 
36:  else
37:     $reanalyze = \text{false}$ 
38:  end if
39: until  $reanalyze = \text{false}$ 

```

5.2. Constraint Solving

This part details how our technique solves flow-sensitive set constraints on the FSConsG. Addr, Copy and Gep edges are solved by the same schemes of flow-insensitive analysis. Namely, each Addr edge adds its source to its target (line 3), each Copy edge adds its points-to objects to its target's points-to set (line 9), and each Gep edge adds the fields of its points-to objects to its target's points-to set (line 13). Store and Load edges involve indirect def-use relations, hence they are solved differently to maintain the flow-sensitivity of address-taken variables.

Processing Store edges Different from flow-insensitive analysis, the Store edges in an FSConsG contain the information of CFG node index, denoting the position of the Store instruction in the CFG. As illustrated in Section 4.2.2, we use CFG node indices to relate versions of address-taken variables (nodes) and the instructions defining (Store edges) or using them (Load edges). Therefore, when processing an Store edge, e.g., $u \xrightarrow{Store, \ell_i} v$ where the points-to object of v is (re)defined, we traverse $pts(v)$. And, for each $o \in pts(v)$, we use the CFG node index, i.e., ℓ_i , to locate the corresponding version of o , i.e., o_{ℓ_i} , and connect the corresponding Copy edges to o_{ℓ_i} , as seen in lines 23 and 25. In particular, our technique also checks strong updates Hardekopf and Lin (2011) when processing a Store edge. If the Store edge is a strong update, then we empty $pts(o_{\ell_i})$, as seen in line 23.

Processing Load edges Similar to processing Store edges, when processing a Load edge, e.g., $v \xrightarrow{Load, \ell_i} w$ where the points-to object of v is used, we traverse $pts(v)$. And, for each $o \in pts(v)$, we use the CFG node index, i.e., ℓ_i , to locate the corresponding version of o , i.e., o_{ℓ_i} , and connect the corresponding Copy edges from o_{ℓ_i} to the target w , as seen in line 29.

Soundness and Precision Guarantee CG-FsPTA guarantees its soundness and precision by producing points-to-set results identical to VSFS, which is embedded in the graph construction and constraint solving processes. In the graph construction process, since both techniques are based on LLVM-IR, where the top-level variables are already in the SSA form, the top-level variables in SVFGs and FSConsGs are naturally flow-sensitive. For address-taken variables, SVFG uses def-use edges connecting SVFG nodes to facilitate different points-to sets of an address-taken variable in different SVFG nodes. Correspondingly, CG-FsPTA separates address-taken variables into different versions, represented as FSConsG nodes, and uses Copy edges to connect them, strictly according to the def-use edges (Section 4.2.2). Thus, the different versions of an address-taken node in an FSConsG correspond to different points-to sets of an address-taken variable in an SVFG. In the solving process, when processing an indirect SVFG node (Store and Load),

VSFS uses indirect SVFG edges to distinguish different versions of the involved address-taken variables. Correspondingly, when processing an indirect FSConsG edge, CG-FSPTA uses the CFG node number on the edge to locate and process the correct version of the corresponding address-taken nodes. Therefore, CG-FSPTA and VSFS produce the same points-to-set results.

6. Evaluation

This section evaluates CG-FSPTA's performance in flow-sensitive pointer analysis through a comparative analysis with VSFS [Barbar et al. \(2021\)](#), a state-of-the-art flow-sensitive pointer analysis framework based on CFG. Our experimental evaluation demonstrates that CG-FSPTA achieves significant performance improvements: CG-FSPTA achieves a **1.93×** average execution time speedup (up to **2.91×**) and **23.99%** memory reduction (up to **54.31%**) when utilizing Andersen's WAVE propagation algorithm [Pereira and Berlin \(2009\)](#), and a **7.27×** average speedup (up to **20.07×**) and **33.05%** memory reduction (up to **61.07%**) when implementing the SFR technique [Lei and Sui \(2019a\)](#). Importantly, these substantial performance gains are achieved while maintaining precision equivalence with VSFS in terms of points-to set computation.

6.1. Experimental Setup

Datasets. Our evaluation employs the SPEC CPU 2017 Benchmark Suite Standard Performance Evaluation Corporation (2017), which comprises a diverse collection of real-world C/C++ applications spanning multiple domains and encompassing over 1 million lines of code. We compiled these programs into LLVM bitcode files (version 16) for analysis. Table 3 presents detailed statistics for each benchmark. The suite includes: *lbm* (fluid dynamics), *mcf* (combinatorial optimization for route planning), *namd* (molecular dynamics simulation), *deepsjeng* (artificial intelligence: alpha-beta tree search for chess), *leela* (artificial intelligence: Monte Carlo tree search for Go), *nab* (nucleic acid simulation), *xz* (general data compression), *x264* (H.264/AVC video encoding), *omnetpp* (discrete event simulation for computer networks), *povray* (ray tracing renderer), *cactus* (physics: numerical relativity), and *imagick* (image manipulation). These benchmarks were selected to represent a diverse range of program sizes and complexities, enabling a comprehensive evaluation of our approach across various scenarios.

Implementation. We conducted our experiments on a Rocky Linux 8.10 server equipped with 28-core Intel Xeon E5-2690v4 2.6GHz CPUs and 252 GB of memory. Our implementation builds upon LLVM version 16.0.0 [Lattner and Adve \(2004\)](#), constructing both the interprocedural control flow graph [Schiebel, Sattler, Schubert, Apel and Bodden \(2024\)](#) and ConsG based on LLVM's intermediate representation (LLVM-IR). In LLVM-IR, program functions are represented as global variables, which we model as address-taken variables during analysis. The analysis proceeds in

several phases. First, we perform a flow-insensitive pointer analysis [Andersen \(1994\)](#) as a preprocessing step. This initial analysis enables the construction of interprocedural memory SSA form [Chow et al. \(1996\)](#); [Hardekopf and Lin \(2011\)](#), which provides the foundation for subsequently building our FSConsG representation. For comparative evaluation, we utilize VSFS's [Barbar et al. \(2021\)](#) original open-source implementation for SVFG-based flow-sensitive pointer analysis as our baseline [Sui and Xue \(2016b\)](#). We then evaluate our CG-FSPTA implementation, which integrates the FSConsG representation with two established inclusion-based pointer analysis techniques: Andersen's WAVE propagation algorithm [Pereira and Berlin \(2009\)](#) and the more recent SFR technique [Lei and Sui \(2019a\)](#).

Evaluation Metrics. Our comparative assessment focuses on two fundamental metrics: the number of constraints within the respective graph representations and the total execution time. The constraints of a graph constitute the structural foundation upon which flow-sensitive pointer analysis depends. Specifically, SVFG constraints encompass both node-level elements (instructions) and edge-level connections (def-use relations), whereas FSConsG exclusively contains edge-level constraints (as specified in Table 2). The constraint count metric directly quantifies the structural efficiency of FSConsG versus SVFG, revealing the extent to which our approach eliminates redundant constraints during graph construction. Total execution time measures the end-to-end duration required to complete the analysis for each benchmark program, serving as the primary indicator of computational efficiency. These complementary metrics enable us to systematically evaluate both the theoretical advantages of our constraint representation and its practical performance implications. By correlating reductions in constraint count with corresponding improvements in execution time, we establish a comprehensive assessment of CG-FSPTA's effectiveness relative to VSFS.

6.2. Research Questions

Our experimental evaluation aims to address the following research questions:

- RQ1 **Comparison of FSConsG and SVFG:** How does our FSConsG representation compare to the SVFG used in VSFS with respect to the number of constraints required for flow-sensitive pointer analysis?
- RQ2 **Speedup of CG-FSPTA:** To what extent does CG-FSPTA improve analysis execution time compared to VSFS when integrated with state-of-the-art inclusion-based pointer analysis algorithms?
- RQ3 **Memory Consumption of CG-FSPTA:** What reduction in memory footprint does CG-FSPTA achieve relative to VSFS during flow-sensitive pointer analysis?

6.3. Comparison of FSConsG and SVFG (RQ1)

Experimental Results. Table 4 presents a comparative analysis between FSConsG and SVFG in terms of constraint

Table 3

Statistics of benchmarks from SPEC CPU 2017 Standard Performance Evaluation Corporation (2017). KLOC represents thousands of lines of code. #Pointers, #Gep, #Load, and #Store denote the number of pointer variables, getelementptr instructions (field accesses), load instructions, and store instructions in the LLVM IR representation, respectively.

No.	Project	KLOC	#Pointers	#Gep	#Load	#Store
1	lbm	1	4,131	437	256	313
2	mcf	3	7,979	1,201	858	709
3	namd	8	332,018	42,700	33,159	15,949
4	deepsjeng	10	28,517	3,392	2,983	2,188
5	leela	21	68,078	8,098	5,263	4,063
6	nab	24	55,925	6,203	6,617	4,657
7	xz	33	46,880	5,348	3,941	3,959
8	x264	96	258,037	67,934	41,617	34,728
9	omnetpp	134	494,762	51,385	47,757	27,090
10	povray	170	279,494	35,867	31,161	25,729
11	cactus	257	209,738	39,117	32,039	14,731
12	imagick	259	507,367	55,404	51,145	32,558
Total		1,016	2,288,926	312,086	260,796	166,673

Table 4

Comparison of constraint numbers between SVFG (used in VSFS Barbar et al. (2021)) and our proposed FSConsG.

Project	SVFG	FSConsG	Reduction
lbm	1,977	663	66.46%
mcf	12,713	8,004	37.04%
namd	279,903	174,211	37.76%
deepsjeng	12,722	4,415	65.30%
leela	159,448	114,079	28.45%
nab	129,820	80,975	37.63%
xz	136,171	105,411	22.59%
x264	6,781,972	6,597,148	2.73%
omnetpp	185,970,609	151,739,481	18.41%
povray	12,021,871	7,798,511	35.21%
cactus	4,922,453	4,368,472	11.25%
imagick	26,823,216	14,985,392	44.13%
Average			33.91%

quantities across our benchmark programs. The data consistently demonstrates that FSConsG requires significantly fewer constraints than SVFG across all benchmarks, with an average reduction of 33.91%. Notably, lbm exhibits the most substantial reduction at 66.46%, where FSConsG eliminates the vast majority of constraints present in SVFG. These findings empirically validate that FSConsG provides a substantially more compact representation than SVFG by systematically eliminating redundant constraints and consolidating necessary ones. This significant reduction in constraint count has profound implications for subsequent analysis stages, as the streamlined graph structure directly contributes to improved computational efficiency and enhanced scalability when performing flow-sensitive pointer analysis.

Results Analysis. The substantial reduction in constraint count can be attributed to two key architectural decisions in our FSConsG design. First, FSConsG implements selective versioning that *targets only address-taken variables*. Unlike

SVFG, which introduces versions across a broader spectrum of program elements, FSConsG restricts versioning exclusively to address-taken variables—precisely those not already in SSA form that require flow-sensitive tracking. This targeted approach significantly reduces the proliferation of nodes that characterizes SVFG-based methods, yielding a more compact representation without sacrificing analytical precision. Second, FSConsG employs a fundamentally different constraint encoding strategy by *storing all constraint information exclusively on graph edges*. In contrast, SVFG-based approaches necessarily maintain constraints at both the edge and node levels to facilitate points-to set propagation. This dual-layer constraint representation in SVFG introduces substantial redundancy, whereas FSConsG’s edge-centric design consolidates all constraint logic into a unified framework. The resulting streamlined architecture not only reduces the absolute constraint count but also simplifies subsequent propagation operations during analysis.

Table 5

Comparison of execution time (in seconds) and speedup ratios between VSFS and CG-FsPTA variants with wave propagation Pereira and Berlin (2009) (CG-FsPTA-WAVE) and SFR Lei and Sui (2019a) (CG-FsPTA-SFR).

Project	VSFS	Cg-FsPta-Wave (Speedup)	Cg-FsPta-SFR (Speedup)
lbm	0.012	0.007 (1.71×)	0.004 (3.00×)
mcf	0.070	0.040 (1.75×)	0.018 (3.89×)
namd	4.026	2.180 (1.85×)	1.163 (3.46×)
deepsjeng	0.138	0.064 (2.16×)	0.043 (3.21×)
leela	2.474	0.982 (2.52×)	0.375 (6.60×)
nab	1.162	0.711 (1.63×)	0.240 (4.84×)
xz	1.552	1.085 (1.43×)	0.258 (6.02×)
x264	158.741	87.487 (1.81×)	16.617 (9.55×)
omnetpp	10067.5	6933.065 (1.45×)	501.629 (20.07×)
povray	302.244	111.240 (2.72×)	24.020 (12.58×)
ldecod	159.376	54.737 (2.91×)	14.151 (11.26×)
imagemick	136.985	116.377 (1.18×)	49.283 (2.78×)
Average		(1.93×)	(7.27×)

FSConsG's structural similarity to traditional constraint graphs enables the application of established graph simplification techniques. Because each edge in FSConsG directly encodes either a pointer operation or def-use relation, techniques such as cycle elimination and constraint folding can be applied systematically before each iteration of the flow-sensitive analysis. These optimizations further reduce the effective graph size and complexity during the analysis process, contributing to the observed efficiency gains.

Answer to RQ1: FSConsG results in significantly fewer constraints because it avoids the extensive node duplication inherent in SVFG-based approaches by confining versioning only to address-taken variables. Moreover, FSConsG exclusively stores constraint information on its edges, unlike SVFG, which must maintain constraints on both nodes and edges to support points-to set propagation. FSConsG enables effective graph simplification techniques such as cycle elimination and folding, further reducing the overall graph complexity.

6.4. Speedup of CG-FsPTA (RQ2)

Table 5 presents a comparative analysis of execution times between VSFS and CG-FsPTA when integrated with two state-of-the-art constraint resolution algorithms: WAVE Propagation (WAVE) Pereira and Berlin (2009) and Stride-based Feld Representation (SFR) Lei and Sui (2019a). CG-FsPTA exhibits significant speedup over VSFS in both scenarios, with an average 1.93× improvement when using WAVE and a remarkable 7.27× speedup when employing SFR. A key factor behind the superior performance of FSConsG over the SVFG-based approach lies in how each graph represents and propagates constraints. In FSConsG, constraint information is maintained exclusively on edges rather than being duplicated in both nodes and edges, as is the case with SVFG. This design choice reduces overhead and streamlines the process of points-to set propagation, particularly in iterative solvers such as WAVE Propagation Pereira and Berlin (2009) and SFR Lei and Sui (2019a).

6.4.1. CG-FsPTA-WAVE

The WAVE Propagation algorithm Pereira and Berlin (2009) is an inclusion-based pointer analysis technique that enhances computational efficiency through three key mechanisms. First, it collapses strongly connected components to form an acyclic constraint graph, enabling a topologically ordered traversal. Second, it propagates pointer information incrementally by transmitting only the differences between current and previously propagated points-to sets, thereby minimizing redundant computations. Third, it dynamically updates the constraint graph by inserting new edges to represent complex pointer relationships, iterating until a fixed point is reached.

Experimental Results. Experimental results demonstrate that CG-FsPTA-WAVE consistently outperforms VSFS across all benchmarks, achieving an average speedup of 1.93×. The performance improvements range from 1.18× for *imagemick* to 2.91× for *ldecod*. Even for computationally lightweight benchmarks such as *lbm* and *mcf*, CG-FsPTA-WAVE exhibits measurable efficiency gains, reducing execution times from 0.012s to 0.007s and from 0.07s to 0.04s, respectively. For computationally intensive benchmarks, the performance advantage becomes more pronounced: analysis time for *povray* decreases from 302.244s to 111.24s (2.72× speedup), while *omnetpp* improves from 10067.5s to 6933.065s (1.45× speedup). These results demonstrate that the structural advantages of FSConsG translate directly into tangible performance benefits when leveraging the WAVE Propagation algorithm.

Results Analysis. In the WAVE Propagation algorithm, collapsing strongly connected components and performing topologically ordered traversals become more efficient when constraints are kept strictly on edges. Because FSConsG forgoes node-level constraints, fewer structures require maintenance or synchronization, thereby accelerating the propagation of pointer information and minimizing redundant work. As evidenced by the 1.93× average speedup reported

in Table 5, even smaller benchmarks (e.g., *lbm* and *mcf*) exhibit notable performance gains, while larger benchmarks like *omnetpp* and *povray* reap substantial time reductions.

6.4.2. CG-FsPTA-SFR

The Stride-based Feld Representation (SFR) algorithm Lei and Sui (2019a) enhances inclusion-based pointer analysis efficiency through strategic constraint graph decomposition. It identifies a minimal causality subgraph containing only nodes affected by recent updates, thereby localizing computation to relevant graph regions. SFR systematically detects and collapses strongly connected components, transforming cyclic structures into an acyclic representation that facilitates topological ordering. This refined architecture significantly reduces computational redundancy during points-to set propagation by isolating updates to affected subgraphs, thus accelerating convergence while preserving the analysis's correctness, predictability, and scalability.

Experimental Results. The integration of SFR with CG-FsPTA yields substantial performance improvements, achieving an average speedup of 7.27 $\times$ over VSFS. Performance gains are consistent across the entire benchmark suite, ranging from 2.78 $\times$ for *imagemagick* to an exceptional 20.06 $\times$ for *omnetpp*. Even smaller benchmarks demonstrate marked improvements, with *lbm* and *mcf* achieving 3.00 $\times$ and 3.89 $\times$ speedups, respectively. Medium-sized programs like *x264* and *povray* show 9.55 $\times$ and 12.58 $\times$ gains. The impact on large-scale benchmarks is particularly noteworthy—*omnetpp*'s execution time decreases from 10067.5s to 501.629s, underscoring the exceptional scalability of our SFR-based implementation when combined with the FSConsG representation.

Results Analysis. These results underscore the advantages of integrating FSConsG with the advanced SFR optimization technique. By maintaining constraint information exclusively on graph edges and applying more aggressive simplifications, CG-FsPTA-SFR reduces redundant constraints and accelerates subsequent pointer analysis phases. SFR refines the constraint graph by identifying and collapsing strongly connected components into a minimal subgraph, focusing updates on the specific regions affected by new information. Because FSConsG already maintains a more compact, edge-centric graph, the decomposition and selective update steps in SFR proceed with significantly less overhead compared to an SVFG-based structure. The result is an average 7.27 $\times$ speedup, including a remarkable 20.07 $\times$ acceleration for *omnetpp*, as shown in Table 5.

Performance Variation Across Benchmarks. The performance variation of CG-FsPTA across benchmarks is primarily influenced by the program scale and the relative weight of the solving phase within the total execution time. Our approach demonstrates the highest speedups (e.g., 20.07 $\times$ for *omnetpp* with SFR) in large-scale benchmarks where the solving phase is the primary bottleneck. In these cases, the streamlined FSConsG structure allows advanced

solvers to localize updates and collapse cycles much more efficiently than on traditional SVFG structures.

Conversely, the reported speedup appears more modest on smaller benchmarks, such as *lbm* and *deepsjeng*, despite achieving a constraint reduction of over 65% (Table 4). This is because the total execution time includes fixed overheads for preprocessing (pre-analysis and FSConsG construction). For smaller codebases, these setup costs represent a substantial portion of the end-to-end duration, which masks the significant performance gains achieved during the actual points-to solving phase. As the program scale increases, these fixed costs become negligible, allowing the architectural efficiency of FSConsG to translate into higher total speedup ratios. This observation is consistent across both the WAVE and SFR integration results reported in Tables 5.

Answer to RQ2: Our experimental results indicate that CG-FsPTA substantially reduces execution time compared to VSFS when using inclusion-based pointer analysis optimizations. In particular, when combined with the WAVE Propagation algorithm, CG-FsPTA achieves an average speedup of 1.93 $\times$. More strikingly, when using SFR—which further refines the graph by collapsing strongly connected components and restricting updates to the minimal subgraph—CG-FsPTA's average speedup rises to 7.27 $\times$. In all cases, CG-FsPTA preserves the same level of precision as VSFS, demonstrating that these speedups do not come at the expense of accuracy.

6.5. Memory Consumption of CG-FsPTA (RQ3)

Experimental Results. Table 6 reveals that integrating FSConsG with the WAVE Propagation algorithm (CG-FsPTA-WAVE) reduces memory consumption by an average of 23.99% compared to VSFS. Notably, this improvement extends across both small and large benchmarks. For instance, *lbm* exhibits reductions exceeding 50%, indicating that even simpler workloads can benefit substantially from CG-FsPTA-WAVE's leaner constraint representation. Medium-scale applications such as *namd* and *deepsjeng* also show considerable savings, while larger, memory-intensive programs like *omnetpp* and *x264* realize more modest, yet still meaningful, reductions in peak memory usage.

When combined with the Stride-based Feld Representation (SFR) solver, CG-FsPTA-SFR demonstrates an even greater reduction in memory usage, achieving an average decrease of 33.05% relative to VSFS. The gains are especially pronounced for complex or large-scale benchmarks. For example, *lbm* again shows over 60% memory savings, and *omnetpp* and *x264* also see more substantial reductions under SFR than under WAVE. These results illustrate that SFR's strategy of localizing updates to a minimal subgraph—together with FSConsG's inherently streamlined structure—can significantly lower the memory footprint of inclusion-based pointer analysis.

Results Analysis. The superior memory efficiency of CG-FsPTA compared to VSFS stems primarily from fundamental architectural differences in constraint representation.

Table 6

Comparison of memory consumption (in MB) and reduction ratios between VSFS and CG-FsP_{TA} variants with wave propagation Pereira and Berlin (2009) (CG-FsP_{TA}-Wave) and SFR Lei and Sui (2019a) (CG-FsP_{TA}-SFR).

Project	VSFS	Cg-FsPta-Wave (Reduction)	Cg-FsPta-SFR (Reduction)
lbm	8.81	4.20 (52.33%)	3.43 (61.07%)
mcf	25.53	18.00 (29.52%)	13.20 (48.32%)
namd	674.48	362.29 (46.29%)	333.91 (50.49%)
deepsjeng	53.10	24.26 (54.31%)	22.97 (56.74%)
leela	295.18	253.08 (14.26%)	186.12 (36.95%)
nab	217.22	181.54 (16.42%)	136.09 (37.35%)
xz	235.35	205.29 (12.77%)	160.91 (31.63%)
x264	10531.30	9565.82 (9.17%)	8643.88 (17.92%)
omnetpp	35184.50	22733.10 (35.39%)	25601.00 (27.24%)
povray	14419.50	13529.04 (6.18%)	12524.40 (13.14%)
cactus	7343.16	6762.89 (7.90%)	6607.95 (10.01%)
imagick	27003.70	26106.74 (3.32%)	25440.70 (5.79%)
Average		23.99%	33.05%

Table 7

Comparison of the number of resolved points-to sets of address-taken variables when using SVFG (as in VSFS Barbar et al. (2021)) versus our proposed FSConsG.

Project	SVFG	FSConsG	Reduction
lbm	159	140	11.95%
mcf	1,523	1,095	28.10%
namd	31,803	29,772	6.39%
deepsjeng	638	577	9.56%
leela	51,582	39,766	22.91%
nab	27,005	24,749	8.35%
xz	52,138	38,950	25.29%
x264	9,040,845	6,094,054	32.59%
omnetpp	51,825,737	47,848,627	7.67%
povray	2,521,484	2,074,473	17.73%
cactus	3,706,226	3,167,779	14.53%
imagick	2,575,964	2,016,879	21.70%
Average			17.23%

While VSFS maintains points-to information for both input and output states of address-taken variables at store instructions Barbar et al. (2021); Sui and Xue (2016a); Hardekopf and Lin (2011), FSConsG employs a more streamlined approach that stores points-to set information exclusively on nodes. Table 7 quantifies this difference in points-to sets of address-taken variables (while points-to sets for top-level variables remain identical across both approaches), revealing that FSConsG consistently requires fewer points-to sets across all benchmarks—averaging a 17.23% reduction. This significant reduction in required points-to sets directly contributes to FSConsG’s smaller memory footprint during analysis.

For CG-FsP_{TA}-WAVE specifically, memory efficiency derives from two synergistic factors. First, the inherently compact FSConsG structure eliminates redundant constraint storage. Second, WAVE’s algorithm—which collapses strongly connected components and propagates only incremental pointer information—operates more efficiently on FSConsG’s

edge-centric representation. With fewer redundant constraints and simplified graph traversal paths, update operations consume substantially less memory during analysis. This architectural synergy between FSConsG and WAVE’s topological processing model explains the observed 23.99% average reduction in memory consumption compared to VSFS.

CG-FsP_{TA}-SFR achieves even greater memory efficiency by combining FSConsG’s streamlined representation with SFR’s advanced graph processing techniques. The SFR algorithm strategically decomposes the constraint graph and collapses cycles, confining points-to set propagation exclusively to graph regions affected by new pointer information. This localized processing approach, when applied to FSConsG’s already-optimized structure, minimizes the memory footprint of in-process data structures during analysis. By restricting both the scope of active computation and the underlying graph representation, CG-FsP_{TA}-SFR achieves a

33.05% reduction in memory consumption while simultaneously delivering substantial performance improvements, demonstrating that FSConsG provides an ideal foundation for advanced inclusion-based pointer analysis algorithms.

Answer to RQ3: CG-FSPTA achieves significant memory consumption reductions compared to VSFS, averaging 23.99% with the WAVE algorithm and 33.05% with SFR. The improvements are primarily due to the FSConsG design, which confines versioning to address-taken variables and maintains constraint information exclusively on edges, thereby eliminating redundant node-level data structures.

7. Related Work

This section examines prior research in two domains central to our work: Andersen-style pointer analysis, which formulates points-to relations as set-constraint problems, and flow-sensitive pointer analysis, which enhances precision by distinguishing points-to relationships at different program points. We discuss how our approach builds upon and differs from these established techniques.

Andersen-Style Pointer Analysis. Andersen's style pointer analysis Andersen (1994) is a widely-adopted and extensively-studied approach that formulates points-to relation resolution as a set-constraint or set-union problem by computing a dynamic transitive closure on the program's constraint graph. Numerous works have focused on improving this analysis at both algorithmic Hardekopf and Lin (2007a); Blackshear, Chang, Sankaranarayanan and Sridharan (2011); Fähndrich, Foster, Su and Aiken (1998); Heintze and Tardieu (2001); Pearce et al. (2007a); Pereira and Berlin (2009); Rountev and Chandra (2000); Lei and Sui (2019a); Sridharan and Fink (2009); Liu et al. (2022); Lei, Bossut, Sui and Zhang (2024) and data-structural Barbar and Sui (2021a,b); Bravenboer and Smaragdakis (2009); Berndt, Lhoták, Qian, Hendren and Umanee (2003); Lhoták and Hendren (2003) levels to reduce the overhead of Andersen's style pointer analysis. These optimizations include employing efficient constraint solvers Pearce, Kelly and Hankin (2007b); Pereira and Berlin (2009); Liu et al. (2022); Lei and Sui (2019a), simplifying constraint graphs through cycle elimination Fähndrich et al. (1998); Hardekopf and Lin (2007a), applying variable substitution Rountev and Chandra (2000); Hardekopf and Lin (2007c), hash consing Barbar and Sui (2021b), and points-to set compaction Barbar and Sui (2021a). Our FSConsG framework inherently benefits from these optimizations, as it employs a constraint graph architecture that naturally accommodates various Andersen's style pointer analyses.

Flow-Sensitive Pointer Analysis. Flow-sensitive pointer analysis Zuo et al. (2021); Hardekopf and Lin (2009, 2011); Barbar et al. (2021); Sui et al. (2016); Sui and Xue (2018); Choi, Cytron and Ferrante (1991); Hind and Pioli (1998) enhances precision by distinguishing points-to relationships

at different program control points. These approaches necessarily interact with program control flow during analysis. Given the computational complexity associated with analyzing multiple program points, researchers have proposed various efficiency-enhancing techniques. Early research Choi et al. (1991); Hind and Pioli (1998); Lhoták and Chung (2011) focused on optimizing the interprocedural control flow graph (ICFG) through the elimination of analysis-irrelevant nodes. Subsequently, Hardekopf and Lin Hardekopf and Lin (2009) introduced semi-sparse analysis, which applies sparse techniques to top-level pointers while retaining traditional analysis methods for address-taken objects. Recent advances in sparse analysis Yao, Zhou, Xiao, Shi, Wu and Zhang (2024); Oh, Heo, Lee, Lee and Yi (2012); Fink et al. (2008); Hardekopf and Lin (2011); Sui and Xue (2018, 2016a); Barbar et al. (2021) leverage def-use analysis to construct sparse value-flow graphs (SVFGs) and perform flow-sensitive analysis on the SVFG. These SVFGs encode def-use relationships between program variables while preserving program point information, with each SVFG node corresponding to a control flow graph node. In contrast, our approach eschews dependence on control flow or value flow graphs during the core flow-sensitive pointer analysis phase. Instead, we develop a novel constraint graph architecture that encodes requisite flow-sensitivity information offline as versioned address-taken variables, enabling the underlying pointer analysis algorithm to operate in a purely flow-insensitive Andersen's style without interaction with the control flow graph during analysis.

8. Conclusion

We present CG-FSPTA, a technique performing flow-sensitive pointer analysis based on a flow-sensitive set-constraint graph FSConsG, to address key limitations of traditional control-flow-based flow-sensitive pointer analysis. CG-FSPTA integrates flow-insensitive and flow-sensitive pointer analyses to the same model, streamlining the analysis process by reducing graph complexity, memory usage, and execution time. These innovations enable significant improvements in computational efficiency and scalability while maintaining the analytical precision required for complex program analysis. In particular, CG-FSPTA achieves an average memory reduction of 33.05% and accelerates flow-sensitive pointer analysis by 7.27× compared to the state-of-art flow-sensitive pointer analysis method. Such results show the efficiency of CG-FSPTA. Our study also suggests the potential for using set-constraint mind to solve flow-sensitive pointer analysis. This promising avenue represents an intriguing direction for future research in further enhancing analysis efficiency.

References

- Andersen, L.O., 1994. Program analysis and specialization for the C programming language. Ph.D. thesis. University of Copenhagen.
- Barbar, M., Sui, Y., 2021a. Compacting points-to sets through object clustering. Proceedings of the ACM on Programming Languages 5, 1–27.

- Barbar, M., Sui, Y., 2021b. Hash consed points-to sets, in: International Static Analysis Symposium, Springer. pp. 25–48.
- Barbar, M., Sui, Y., Chen, S., 2021. Object Versioning for Flow-Sensitive Pointer Analysis, in: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, Seoul, Korea (South). pp. 222–235. URL: <https://ieeexplore.ieee.org/document/9370334/>, doi:10.1109/CGO51591.2021.9370334.
- Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N., 2003. Points-to analysis using BDDs, in: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, ACM, USA. pp. 103–114. doi:10.1145/781131.781144.
- Blackshear, S., Chang, B.Y.E., Sankaranarayanan, S., Sridharan, M., 2011. The flow-insensitive precision of Andersen’s analysis in practice, in: Proceedings of the 18th International Conference on Static Analysis, Springer, Germany. pp. 60–76. doi:10.1007/978-3-642-23702-7_9.
- Bravenboer, M., Smaragdakis, Y., 2009. Strictly declarative specification of sophisticated points-to analyses, in: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, ACM, USA. pp. 243–262. doi:10.1145/1640089.1640108.
- Cheng, X., Ren, J., Sui, Y., 2024. Fast graph simplification for path-sensitive typestate analysis through tempo-spatial multi-point slicing. *Proc. ACM Softw. Eng.*
- Choi, J.D., Cytron, R., Ferrante, J., 1991. Automatic construction of sparse data flow evaluation graphs, in: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 55–66.
- Chow, F., Chan, S., Liu, S.M., Lo, R., Streich, M., 1996. Effective representation of aliases and indirect memory operations in ssa form, in: Compiler Construction: 6th International Conference, CC’96 Linköping, Sweden, April 24–26, 1996 Proceedings 6, Springer. pp. 253–267.
- Cook, C.R., Evans, A.B., 1979. Graph folding. *Congressus Numerantium* XXIII, 305–314.
- Das, M., Lerner, S., Seigle, M., 2002. Esp: Path-sensitive program verification in polynomial time, in: Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation, ACM. URL: <https://doi.org/10.1145/512529.512538>, doi:10.1145/512529.512538.
- Fähndrich, M., Foster, J.S., Su, Z., Aiken, A., 1998. Partial online cycle elimination in inclusion constraint graphs, in: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, ACM, USA. pp. 85–96. doi:10.1145/277650.277667.
- Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E., 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 1–34.
- Guo, Y., Yao, P., Zhang, C., 2024. Precise compositional buffer overflow detection via heap disjointness, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 63–75.
- Hardekopf, B., Lin, C., 2007a. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’07)*, 290–299.
- Hardekopf, B., Lin, C., 2007b. Exploiting pointer and location equivalence to optimize pointer analysis, in: International Static Analysis Symposium, Springer. pp. 265–280.
- Hardekopf, B., Lin, C., 2007c. Exploiting pointer and location equivalence to optimize pointer analysis, in: International Static Analysis Symposium, Springer, Germany. pp. 265–280. doi:10.1007/978-3-540-74061-2_17.
- Hardekopf, B., Lin, C., 2009. Semi-sparse flow-sensitive pointer analysis, in: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, New York, NY, USA. pp. 226–238. doi:10.1145/1480881.1480911.
- Hardekopf, B., Lin, C., 2011. Flow-sensitive pointer analysis for millions of lines of code, in: International Symposium on Code Generation and Optimization (CGO 2011), IEEE, Chamonix, France. pp. 289–298. URL: <http://ieeexplore.ieee.org/document/5764696/>, doi:10.1109/CGO.2011.5764696.
- Heintze, N., Tardieu, O., 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second, in: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, ACM, USA. pp. 254–263. doi:10.1145/378795.378855.
- Hind, M., Pioli, A., 1998. Assessing the effects of flow-sensitivity on pointer alias analyses, in: International Static Analysis Symposium, Springer. pp. 57–81.
- Lattner, C., Adve, V., 2004. Llvm: A compilation framework for lifelong program analysis & transformation, in: International symposium on code generation and optimization, 2004. CGO 2004., IEEE. pp. 75–86.
- Lee, O., Yang, H., Yi, K., 2005. Automatic verification of pointer programs using grammar-based shape analysis, in: Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4–8, 2005. Proceedings 14, Springer. pp. 124–140.
- Lei, Y., Bossut, C., Sui, Y., Zhang, Q., 2024. Context-free language reachability via skewed tabulation. *Proceedings of the ACM on Programming Languages* 8, 1830–1853.
- Lei, Y., Sui, Y., 2019a. Fast and Precise Handling of Positive Weight Cycles for Field-Sensitive Pointer Analysis, in: Chang, B.Y.E. (Ed.), *Static Analysis*. Springer International Publishing. volume 11822, pp. 27–47. URL: http://link.springer.com/10.1007/978-3-030-32304-2_3, doi:10.1007/978-3-030-32304-2_3, series Title: *Lecture Notes in Computer Science*.
- Lei, Y., Sui, Y., 2019b. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis, in: *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings 26*, Springer. pp. 27–47.
- Lei, Y., Sui, Y., Tan, S.H., Zhang, Q., 2023. Recursive state machine guided graph folding for context-free language reachability. *Proceedings of the ACM on Programming Languages* 7, 318–342.
- Lhoták, O., Chung, K.C.A., 2011. Points-to analysis with efficient strong updates, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, NY, USA. p. 3–16.
- Lhoták, O., Hendren, L., 2003. Scaling Java points-to analysis using SPARK, in: Proceedings of the 12th International Conference on Compiler Construction, Springer, Germany. pp. 153–169.
- Li, T., Bai, J.J., Sui, Y., Hu, S.M., 2022. Path-sensitive and alias-aware typestate analysis for detecting os bugs, in: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 859–872.
- Liu, P., Li, Y., Swain, B., Huang, J., 2022. Pus: A fast and highly efficient solver for inclusion-based pointer analysis, in: Proceedings of the 44th International Conference on Software Engineering, pp. 1781–1792.
- Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G., 2017. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers, in: 26th USENIX Security Symposium (USENIX Security 17), pp. 1007–1024.
- Nuutila, E., Soisalon-Soininen, E., 1994. On finding the strongly connected components in a directed graph. *Information Processing Letters* 49, 9–14. URL: <https://linkinghub.elsevier.com/retrieve/pii/0020019094900477>, doi:10.1016/0020-0190(94)90047-7.
- Oh, H., Heo, K., Lee, W., Lee, W., Yi, K., 2012. Design and implementation of sparse global analyses for c-like languages, in: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 229–238.
- Pearce, D.J., Kelly, P.H., Hankin, C., 2007a. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems* 30, 4. URL: <https://dl.acm.org/doi/10.1145/1290520.1290524>, doi:10.1145/1290520.1290524.
- Pearce, D.J., Kelly, P.H., Hankin, C., 2007b. Efficient field-sensitive pointer analysis for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 4.
- Pereira, F.M.Q., Berlin, D., 2009. Wave Propagation and Deep Propagation for Pointer Analysis, in: 2009 International Symposium on Code

- Generation and Optimization, IEEE, Seattle, WA, USA. pp. 126–135. URL: <http://ieeexplore.ieee.org/document/4907657/>, doi:10.1109/CGO.2009.9.
- Rountev, A., Chandra, S., 2000. Off-line variable substitution for scaling points-to analysis, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM, USA. pp. 47–56. doi:10.1145/349299.349310.
- Schiebel, F., Sattler, F., Schubert, P.D., Apel, S., Bodden, E., 2024. Scaling interprocedural static data-flow analysis to large c/c++ applications: An experience report, in: 38th European Conference on Object-Oriented Programming (ECOOP 2024), Schloss Dagstuhl–Leibniz-Zentrum für Informatik. pp. 36–1.
- Shi, Q., Yao, P., Wu, R., Zhang, C., 2021. Path-sensitive sparse analysis without path conditions, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, ACM. doi:10.1145/3453483.3454086.
- Sridharan, M., Fink, S.J., 2009. The complexity of Andersen’s analysis in practice, in: Proceedings of the 16th International Symposium on Static Analysis, Springer, Germany. pp. 205–221. doi:10.1007/978-3-642-03237-0_15.
- Standard Performance Evaluation Corporation, 2017. SPEC CPU 2017 Benchmark Suite. URL: <https://www.spec.org/cpu2017/>. accessed: 2025-03-25.
- Sui, Y., Di, P., Xue, J., 2016. Sparse flow-sensitive pointer analysis for multithreaded programs, in: Proceedings of the 2016 International Symposium on Code Generation and Optimization, ACM, Barcelona Spain. pp. 160–170. URL: <https://dl.acm.org/doi/10.1145/2854038.2854043>, doi:10.1145/2854038.2854043.
- Sui, Y., Xue, J., 2016a. On-demand strong update analysis via value-flow refinement, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, Seattle WA USA. pp. 460–473. URL: <https://dl.acm.org/doi/10.1145/2950290.2950296>, doi:10.1145/2950290.2950296.
- Sui, Y., Xue, J., 2016b. SVF: Interprocedural static value-flow analysis in LLVM, in: CC ’16, pp. 265–266.
- Sui, Y., Xue, J., 2017. Demand-Driven Pointer Analysis with Strong Updates via Value-Flow Refinement. URL: <http://arxiv.org/abs/1701.05650>. arXiv:1701.05650 [cs].
- Sui, Y., Xue, J., 2018. Value-Flow-Based Demand-Driven Pointer Analysis for C and C++. IEEE Transactions on Software Engineering 14.
- Sui, Y., Ye, D., Xue, J., 2012. Static memory leak detection using full-sparse value-flow analysis, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM. doi:10.1145/2338965.2336784.
- Sui, Y., Ye, D., Xue, J., 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. IEEE Transactions on Software Engineering 40, 107–122. URL: <http://ieeexplore.ieee.org/document/6720116/>, doi:10.1109/TSE.2014.2302311.
- Tan, T., Li, Y., Ma, X., Xu, C., Smaragdakis, Y., 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. Proceedings of the ACM on Programming Languages 5, 1–27.
- Tarjan, R., 1972. Depth-first search and linear graph algorithms. SIAM journal on computing 1, 146–160.
- Yan, H., Sui, Y., Chen, S., Xue, J., 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities, in: Proceedings of the 40th International Conference on Software Engineering, pp. 327–337.
- Yao, P., Zhou, J., Xiao, X., Shi, Q., Wu, R., Zhang, C., 2024. Falcon: A fused approach to path-sensitive sparse data dependence analysis. Proceedings of the ACM on Programming Languages 8, 567–592.
- Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z., 2010. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code, in: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, ACM, Toronto Ontario Canada. pp. 218–229. URL: <https://dl.acm.org/doi/10.1145/1772954.1772985>, doi:10.1145/1772954.1772985.
- Zuo, Z., Zhang, Y., Pan, Q., Lu, S., Li, Y., Wang, L., Li, X., Xu, G.H., 2021. Chianina: An evolving graph system for flow-and context-sensitive analyses of million lines of c code, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 914–929.