IncSFS: Incremental Full-Sparse Flow-Sensitive Pointer Analysis for C/C++

Kunlin Liu*
College of Computer Science and Technology,
National University of Defense Technology
Changsha, China
klliu18@nudt.edu.cn

Zhenbang Chen*†
College of Computer Science and Technology,
National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Piyi Zu*
College of Computer Science and Technology,
National University of Defense Technology
Changsha, China
piyizu@nudt.edu.cn

Yide Du*
College of Computer Science and Technology,
National University of Defense Technology
Changsha, China
dyd1024@nudt.edu.cn

Ji Wang*
College of Computer Science and Technology,
National University of Defense Technology
Changsha, China
wj@nudt.edu.cn

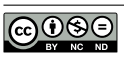
Abstract

Pointer analysis is the fundamental technique for compiler optimization and program analysis. Flow sensitive pointer analysis provides high precision but hard to scale to large-size projects. Tailored for rapid iteration scenarios where software evolves continuously, we introduce IncSFS, the first incremental full-sparse flow-sensitive pointer analysis algorithm for C/C++ programs. The algorithm contains two main steps. It first transforms the value-flow graph of a program into a constraint graph, on which a strongly-connected component detection is performed to ensure precision. It then propagates increases and decreases in points-to set in an interleaving manner to support both code deletion and insertion within a single analysis pass. IncSFS is guaranteed to terminate and compute the least fixed point when point-to relation during the analysis is object-acyclic. Experimental results on six large-scale real-world projects show that IncSFS is both precise and efficient, achieving average speedups of 9.60x over full flow-sensitive pointer analysis and of 5.84x over the traditional reset-recompute incremental approach. Furthermore, IncSFS improves efficiency by 15.8% compared with state-of-the-art incremental pointer analysis algorithms that also propagate changes in points-to sets.

CCS Concepts

• **Software and its engineering → Automated static analysis**.

*Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China.
†Zhenbang Chen is the corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837496

Keywords

Full-sparse Flow-sensitive Pointer Analysis, Incremental Analysis, Constraint Graph

ACM Reference Format:
Kunlin Liu, Zhenbang Chen, Piyi Zu, Yide Du, and Ji Wang. 2026. IncSFS: Incremental Full-Sparse Flow-Sensitive Pointer Analysis for C/C++. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837496

1 Introduction

Pointer analysis statically computes the memory address which pointer variables point to at runtime. Its precision is crucial to compiler optimization and program analysis. One of precision improvement dimension is flow-sensitive. It provides high precision and serves as a foundational technique in a wide array of upstream applications, ranging from compiler optimizations [3, 4, 7, 10, 19] to complex program analyses such as typestate analysis [8], multi-threaded program analysis [30, 36], bug detection [13, 26], and program slicing [22, 33].

Compared with flow-insensitive which treats statements as unordered, flow-sensitive pointer analysis follows the program's execution order in control flow graph (CFG). As program executes, the memory addresses pointed to by a pointer variable may change over time and differs at different statements. To capture this dynamic behavior, flow-sensitive pointer analysis propagates points-to information statement by statement and maintains a separate points-to information for each pointer variable at every program statement. It leverages *strong update* [6, 20] to update points-to information with a new one if we know the points-to information of this pointer variable must be updated. Since real-world programs often comprise hundreds of thousands of pointer variables and memory objects, and the program can have hundreds of thousands of statements, flow-sensitive pointer analysis struggles to scale to large-size software [15, 16, 31, 41]. Due to its practical importance,

```
1  p = x; // p and x points to {o1,o2}
2  - p = y; // y points to {o2,o3}
3  q = p; //
```

Figure 1: IPA performs inefficiently in this deletion scenario

lots of works have been proposed to improve its scalability [2, 14, 15, 20, 21, 41, 44]. The state-of-the-art and most widely used flow-sensitive pointer analysis is **full-sparse flow-sensitive pointer analysis (SFS)** [15]. The primary efficiency gain of SFS stems from its use of *value-flow graph* (VFG) rather than CFG, where a variable's points-to set is propagated directly from its definition sites to potential use sites, skipping irrelevant statements. Despite its significantly improved efficiency, *SFS faces the same scalability challenges as traditional flow-sensitive pointer analyses and still struggles to scale to large codebases—a long-standing open problem.*

This scalability bottleneck is particularly critical due to the widespread adoption of Continuous Integration and Continuous Delivery (CI/CD) in modern software engineering. To meet fast-feedback requirements, incremental pointer analysis becomes crucial by updating results based on code changes rather than recomputing them from scratch. In recent years, a growing body of approaches have been proposed for incremental pointer analysis [23, 24, 27, 38, 42, 43]. These approaches can be divided into three categories: 1) The *Reset–Recompute* approach [12, 38, 45], which resets the points-to results at all affected statements to a safe initial estimate ,*i.e.*, an empty set and recomputes them via a full re-analysis. In particular, the work by Yur *et al.* was the first to apply the Reset–Recompute approach to incremental traditional flow- and context-sensitive pointer analysis, rather than to full-sparse flow-sensitive pointer analysis. 2) The *Restart–Iteration* approach [5, 11, 42], which resets the points-to results at only the modified statements to an empty set and recomputes. 3) The *Incremental Pointer Analysis* (IPA) framework [23, 24, 43], which computes and propagates only the *deltas* (differences) in points-to information instead of resetting.

Limitation of existing approaches. The first category ensures the safety of results but faces performance problem. The process of resetting and recomputing often leads to significant redundant computations. The second category enjoys efficiency but suffers from imprecision when reusing old analysis results within strongly connected components (SCCs) of the program [27]. Both the incremental flow-sensitive approaches above operate directly on the control-flow graph rather than the full-sparse *value-flow graph*. The third category achieves high efficiency by leveraging delta propagation but is currently limited to flow-insensitive settings. As a result, *no prior researches on incremental analysis have been conducted on the full-sparse flow-sensitive pointer analysis.*

In this paper, we present the **first incremental algorithm for full-sparse flow-sensitive pointer analysis**. Inspired by IPA, we aim to propagate the deltas of points-to sets from code modification as it is particularly well suited for pointer analysis and has been shown to be more efficient than reset-based methods [24].

Challenges. IPA is inherently tailored to flow-insensitive analysis, where constraint edges encode propagation paths between variables. When one propagation path is removed, IPA inspects the remaining propagation path from other variables to validate whether the potentially removed memory objects should be removed. For an example, in Figure 1, q originally points-to {o1,o2,o3} which is propagated from x and y in a flow-insensitive way. When Line 2 is deleted, the propagation path from y to q is deleted and IPA find another propagation path from x still contain o2, then only *o*3 is removed from points-to set of q. In IPA, all propagation paths require to be found, otherwise resulting in incorrectness. However, flow-sensitive analysis is fundamentally more complex. To find propagation paths between variables in flow-sensitive pointer analysis, IPA further requires to consider the control flow reachability between variables and ensure points-to set of variables is not strongly updated in the propagation process. In this example, x cannot flow to q since p is strongly updated at Line 2. Since the strong update could happen for any variable at any statement, different variables may have different propagation path. How to encode propagation paths with control flow information for all variables is the **key challenge** for extending IPA to flow-sensitivity.

Another **key challenge** is how to characterize the evolution of the SFS-based value flow graph in response to code changes. As illustrated in Fig.1, when the second line is deleted, the value flow edge from `p = y` to `q = p` is deleted and `p = x` to `q = p` is added. The points-to set of `q` changes from $\{o2, o3\}$ to $\{o1, o2\}$, resulting in the removal of $o3$ and the addition of $o1$. However, traditional IPA typically assumes that code deletion leads solely to set reduction, thus lacking an efficient and unified algorithm to handle such complex, non-monotonic updates. Existing approaches struggle to identify the correlation between additions and deletions, thus resorting to a "delete-then-add" engineering strategy [23, 24, 43].

Our solution. To encode control flow information and align with IPA, our insight is to instantiate a unique renamed variable for the variable at the redefined statement to capture its local points-to set. We then construct a constraint graph used in flow-insensitive analysis to propagates points-to set among these renamed variables. Consequently, we perform IPA on the constraint graph in a flow-insensitive way. To address the second challenge, we compute increases and decreases in points-to sets in an interleaved manner in response to value flow graph changes. To guarantee termination and convergence to the least fixed point, we ensure that increases and decreases are mutually exclusive and require object-acyclicity throughout the analysis: the points-to relation remains acyclic. Another key scenario is code *replacement*. Although Git represents replacements as deletions followed by insertions, the statements are often semantically correlated; our method exploits this correlation to avoid redundant propagation in standard IPA.

Based on the above approaches, we have designed our incremental full-sparse flow-sensitive pointer analysis, *i.e.*, IncSFS, and implemented it in the SVF framework [37]. We have evaluated our approach in large scale real-world programs. We compare our approach with both full analysis and traditional reset-recompute approach, and it achieves 9.60x and 5.84x speedups respectively. We also compare our interleaved propagation with IPA's separate propagation and it achieves 15.8% efficient improvement.

In summary, we claim the following contributions:

Table 1: Program Syntax

Type	Statement
Alloc	$p = alloc_o$
Copy	$p = q$
Field	$p = \&q.f$
Phi	$p = \phi(p_1, \dots, p_n)$
Store	$*p = q$
Load	$p = *q$

Table 2: Abstract Domain

Notation	Definition
$\mathcal{L}$	Labels
$\mathcal{O}$	Address Taken Variables
$\mathcal{T}$	Top Level Variables
$\mathcal{V} := \mathcal{O} \cup \mathcal{T}$	Variables
$\mathcal{P} := \mathcal{V} \to 2^{\mathcal{O}}$	Points-to Set of Variables
$\texttt{IN} := \mathcal{L} \to \mathcal{V} \to 2^{\mathcal{O}}$	Points-to set at each label
$\texttt{OUT} := \mathcal{L} \to \mathcal{V} \to 2^{\mathcal{O}}$	Points-to set at each label

- We propose the first incremental algorithm for full-sparse flow-sensitive pointer analysis that efficiently updates the points-to information in CI/CD scenarios.
- We rename variables to enable safe propagation of points-to deltas in incremental settings by transforming the VFG into a variable-level constraint graph.
- We support both code insertions and deletions by jointly propagating positive and negative deltas; under the object-acyclicity condition, the algorithm terminates and reaches the least fixed point despite the non-monotonic effects of code modification.
- We evaluate our approach on large-scale real-world programs and demonstrate that it achieves 9.60x and 5.84x speedups than full analysis and reset-recompute approach respectively.

2 PRELIMINARIES And Motivation

2.1 Program Syntax

Table 1 defines our program syntax with six statement forms: **Alloc**, **Copy**, **Field**, **Phi**, **Store**, and **Load**. The input programs are in a static single assignment (SSA) form, where each variable is defined only once. The input program is in a static single assignment (SSA) form in which variable is defined only once.

2.2 Full-Sparse Flow-Sensitive Pointer Analysis

2.2.1 Abstract domains. The abstract domains are illustrated in Table 2. We assign a label $\ell \in \mathcal{L}$ for each statement, representing the location of statement within the program. All Variables are partitioned as $\mathcal{V} = \mathcal{T} \cup \mathcal{O}$, where $\mathcal{T}$ contains contains top-level SSA variables and $\mathcal{O}$ contains address-taken variables, such as abstract heap objects accessed through loads and stores. For example, as shown in Figure 2, we omit the definitions of variables a, b, d, p, q, m, n, x and assume the points-to sets of these variables are shown in Figure 2 on the right. Variables x, a , b, c, d, m, n, p, q are top level variables while o^p, o^q, o^x, o^a, o^b are address-taken variables. $\mathcal{P}$ is a mapping which stores the points-to set for each top-level variable. Flow-sensitive pointer analysis maintains IN and OUT at each statement which maps every variable to its points-to set before and after that statement.

2.2.2 Full-sparse flow-sensitive pointer analysis. Traditional flow-sensitive pointer analysis blindly computes and propagates points-to result of all variable statement by statement along the CFG regardless of it is used or not in the statement, resulting in inefficiency. Full-sparse flow-sensitive pointer analysis (SFS) [15] improve this by eliminating unnecessary points-to information propagation of a variable if this variable is not used at this statement. For example, since o^x is not used at Line 4, SFS does not maintain points-to result of o^x in $\texttt{IN}_4$ and $\texttt{OUT}_4$ and does not propagate the points-to set of o^x to Line 4. To realize it, SFS leverages a value-flow graph (VFG) (Definition 2.1) to propagate points-to sets directly from definition sites to use sites [14, 15, 28].

*Definition 2.1 (**Value-Flow Graph**).* Value-flow graph (VFG) is defined as a labeled directed graph $G = (N, E)$, where N is a set of nodes, each of which represents a program statement and $E \subseteq N \times N \times \mathcal{V}$ is a set of labeled directed edges. Each edge $n_1 \overset{o}{\rightsquigarrow} n_2 \in E$ indicates that the variable o defined at statement n_1 may be used at statement n_2.

Construction of the VFG. SFS relies on a flow-insensitive pre-analysis to identify address-taken variables potentially accessed by load and store statements. It then treats stores as potential definitions and loads as potential uses and connects their def-use relations to construct the VFG. As shown in Figure 2, $\mathcal{P}(m)$ here is over-approximated as $\{o^p, o^q\}$ under the pre-analysis, while the other points-to sets remain unchanged. Thus, `*m = a` may define both o^p and o^q, and `*n = c` may use and redefine them, producing a value-flow edge from `*m = a` to `*n = c` labeled with o^p and o^q.

Full-sparse flow-sensitive pointer analysis algorithm. SFS propagates points-to result along the VFG instead of CFG following Equation 1, $\texttt{IN}_n(o)$ is computed as the union of all $\texttt{OUT}_k(o)$ from predecessor nodes k that can reach n via a value-flow edge, *i.e.*, $k \overset{o}{\rightsquigarrow} n$. The $\texttt{OUT}_n(o)$ set is then strongly updated by combining newly generated pointers $\texttt{GEN}_n(o)$ with those from $\texttt{IN}_n(o)$ that are not killed by n i.e., $\texttt{IN}_n(o) \setminus \texttt{KILL}_n(o)$. Three behaviors are categorized based on $\mathcal{P}(p)$ for store statement $\ell : *p = q$: no operation (Definition 2.2), *strong update* (Definition 2.3), and *weak update* (Definition 2.4). Taking Line 2: `*x = b` as an example, since x only point to o^x, a strong update is performed: it kills all incoming points-to set from the predecessor ($\texttt{KILL}_2(o^x) = \texttt{IN}_2(o^x)$) and generate a new set for o^x: $\texttt{OUT}_2(o^x) = \mathcal{P}(b)$. Since `*n = c` where n points to two objects, a weak updates is performed and $\forall_{o \in \mathcal{P}(n)} \texttt{KILL}_8(o) = \emptyset$.

$$\begin{aligned} \texttt{IN}_n(o) &= \bigcup_{k \overset{o}{\rightsquigarrow} n} \texttt{OUT}_k(o) \\ \texttt{OUT}_n(o) &= \texttt{GEN}_n(o) \cup (\texttt{IN}_n(o) \setminus \texttt{KILL}_n(o)) \end{aligned} \tag{1}$$

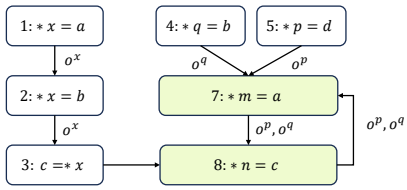
*Definition 2.2 (**No Operation**).* For store statement $\ell : *p = q$, if $|\mathcal{P}(p)| = 0$, SFS does not handle this statement to ensure monotonicity and $\forall_{o \in \mathcal{O}} \texttt{OUT}_\ell(o) = \emptyset$.

*Definition 2.3 (**Strong Update**).* For store statement $\ell : *p = q$, if $|\mathcal{P}(p)| = 1 \wedge o \in \mathcal{P}(p)$, SFS replaces $\texttt{OUT}_\ell(o)$ with a new value,*i.e.*, $\texttt{OUT}_\ell(o) = \mathcal{P}(q)$. And for other address-taken variables o' ($o \neq o'$) reaching this statement, the points-to result just bypass this node,*i.e.*,$\texttt{OUT}_\ell(o') = \texttt{IN}_\ell(o')$.

```
1  *x = a;
2  -*x = b;
3  c = *x;
4  *q = b;
5  *p = d;
6  while(...){
7    *m = a;
8    *n = c;
9  }
```

Var	Pts Set
x	$\{o^x\}$
p	$\{o^p\}$
q	$\{o^q\}$
m	$\{o^q\}$
n	$\{o^p, o^q\}$
a	$\{o^a\}$
b	$\{o^a, o^b\}$
d	$\{o^a\}$

Figure 2: Motivating example.



Node	Variable	IN	OUT
1	o^x	$\emptyset$	$\{o^a\}$
2	o^x	$\{o^a\}$	$\{o^a, o^b\}$
3	o^x	$\{o^a, o^b\}$	$\emptyset$
4	o^q	$\emptyset$	$\{o^a, o^b\}$
5	o^p	$\emptyset$	$\{o^a\}$
7	o^p	$\{o^a, o^b\}$	$\{o^a, o^b\}$
	o^q	$\{o^a, o^b\}$	$\{o^a\}$
8	o^p	$\{o^a, o^b\}$	$\{o^a, o^b\}$
	o^q	$\{o^a\}$	$\{o^a, o^b\}$

Figure 3: the VFG (left) and the corresponding points-to sets during the sparse flow-sensitive analysis (right).

*Definition 2.4 (**Weak Update**).* For store statement $\ell : *p = q$, if $|\mathcal{P}(p)| > 1 \wedge o \in \mathcal{P}(p)$, SFS merges $\mathtt{OUT}_\ell(o)$ with a new value ,*i.e.*, $\mathtt{OUT}_\ell(o) = \mathtt{IN}_\ell(o) \cup \mathcal{P}(q)$. And for other address-taken variables o' reaching this statement, the points-to result just bypass this node,*i.e.*,$\mathtt{OUT}_\ell(o') = \mathtt{IN}_\ell(o')$.

In the presence of loops or recursive calls, the analysis iteratively updates the points-to sets until a fixed point is reached [14]. For example in Figure 3, the nodes shaded in green form an SCC. We present a walkthrough of SFS starting with Node 8. Initially the IN and OUT at node 7 and node 8 are all $\emptyset$. We firstly focus on node 8: since $\mathcal{P}(c) = \mathtt{IN}_3(o^x) = \mathtt{OUT}_2(o^x) = \{o^a, o^b\}$, then $\mathtt{OUT}_8(o^p) = \mathtt{IN}_8(o^p) \cup \mathcal{P}(c) = \emptyset \cup \{o^a, o^b\} = \{o^a, o^b\}$,and $\mathtt{OUT}_8(o^q)$ is also $\{o^a, o^b\}$. Then we focus on node 7: $\mathtt{IN}_7(o^p) = \mathtt{OUT}_5(o^p) \cup \mathtt{OUT}_8(o^p) = \{o^a\} \cup \{o^a, o^b\} = \{o^a, o^b\}$ and $\mathtt{IN}_7(o^q) = \mathtt{OUT}_4(o^q) \cup \mathtt{OUT}_8(o^q) = \{o^a, o^b\} \cup \{o^a, o^b\} = \{o^a, o^b\}$. Since node 7 performs a strong update for o^q, $\mathtt{OUT}_7(o^q) = \mathcal{P}(a) = \{o^a\}$ while $\mathtt{OUT}_7(o^p) = \mathtt{IN}_7(o^p) = \{o^a, o^b\}$. And the $\mathtt{OUT}_7$ is propagated to node 8, resulting in $\mathtt{IN}_8(o^p) = \{o^a, o^b\}$ and $\mathtt{IN}_8(o^q) = \{o^a\}$. Restart analyzing node 8 again, the result is stable and reaches a fixed point.

2.3 Incremental Pointer Analysis

IPA is designed for flow-insensitive pointer analysis which translates program statements into a set of constraints in Table 3, constructing a constraint graph (Definition 2.5) [1]. Flow-insensitive pointer analysis iteratively propagates points-to sets along the constraint edges until a fixed point is reached.

*Definition 2.5 (**Constraint Graph**).* The constraint graph is a directed graph $G = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N} \subseteq \mathcal{V}$ represents the set of variables and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ represents subset constraints between point-to sets of connected variables.

Overall algorithm. The input to IPA is an acyclic constraint graph by identifying SCCs and collapsing each SCC into a single representative node. Importantly, IPA leverage the *change-locality property* [24] of constraint graph: upon deleting a copy constraint edge, it suffices to check the incoming neighbors of the target node in the constraint graph to determine whether the potential removed objects should be removed and continue to propagate. For example, in Figure 4, deleting $b \to o_2^x$ makes $\{o^a, o^b\}$ candidate removals. Since the remaining predecessor a still contains o^a, IPA preserves o^a and propagates only the removal of o^b from $\mathcal{P}(c)$.

Table 3: Constraints for flow-insensitive pointer analysis. The edge $p \leftarrow q \in \mathcal{E}$ means the constraint $\mathcal{P}(q) \subseteq \mathcal{P}(p)$. The edge $p \overset{f}{\leftarrow} q \in \mathcal{E}$ means the constraint $\forall o \in \mathcal{P}(q), o.f \in \mathcal{P}(p)$.

Type	Statement	Points-To Constraint
Alloc	$p = alloc_o$	$o \in \mathcal{P}(p)$
Copy	$p = q$	$p \leftarrow q \in \mathcal{E}$
Phi	$p = \phi(p_1, \ldots, p_n)$	$\forall p_i : p \leftarrow p_i \in \mathcal{E}$
Load	$p = *q$	$\forall o \in \mathcal{P}(q) : p \leftarrow o \in \mathcal{E}$
Store	$*p = q$	$\forall o \in \mathcal{P}(p) : o \leftarrow q \in \mathcal{E}$
Field	$p = \&q.f$	$p \overset{f}{\leftarrow} q \in \mathcal{E}$

2.4 Motivation

2.4.1 Transform the VFG to a constraint graph. **Identify the SCCs as IPA.** We aim to extend IPA's delta propagation [24] to flow-sensitive pointer analysis. IPA must identify SCCs and use the predecessors of the entire SCC to filter removed objects; otherwise, imprecision may be introduced. In Figure 3 as an example, after deleting Line 2 (`*x = b`), the points-to set of variable c changes from $\{o^a, o^b\}$ to $\{o^a\}$(replaced by a from b), and the reduced set $\{o^b\}$ is propagated to its successor node 8. A local check would retain o^b because it remains in $\mathtt{OUT}_7(o^p)$. However, since the occurrence of o^a in both $\mathtt{OUT}_7(o^p)$ and $\mathtt{OUT}_8(o^p)$ is solely due to the propagation from $\mathcal{P}(c)$, both $\mathtt{OUT}_8(o_p)$ and $\mathtt{OUT}_7(o^p)$ should decrease from $\{o^a, o^b\}$ to $\{o^a\}$. We cannot use $\mathtt{OUT}_7(o^p)$ to validate $\mathtt{OUT}_8(o^p)$, as they are in the same SCC, where their values are interdependent and potentially changing together.

Strong updates may break SCCs presented on the VFG. Considering the example in Figure 3, node 7 and node 8 form a single SCC. Since node 7:`*m = a` performs a strong update, $\mathtt{OUT}_8(o^q)$ does not contribute to $\mathtt{OUT}_7(o^q)$. Therefore, the propagation path between $\mathtt{OUT}_7(o^q)$ and $\mathtt{OUT}_8(o^q)$ cannot form a SCC. If we mistakenly treat these two as belonging to the same SCC, when $\mathtt{OUT}_8(o^q)$ is reduced, the predecessors of $\mathtt{OUT}_7(o^q)$ may be incorrectly used to filter the potentially removed object, resulting in imprecision.

This highlights a critical issue: in flow-sensitive pointer analysis, strong updates for different address-taken variable will make these variables have a different dependency structure and the SCC decomposition varies per variable. To efficiently reuse the *change-local property* like IPA, our key insight is that SCC identification

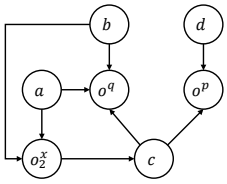


Figure 4: Example constraint graph in flow-insensitivity.

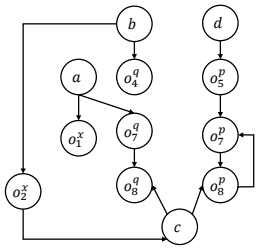


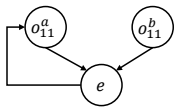
Figure 5: Constraint Graph with respect to the example VFG

must be object-specific. As a result, we transform the statement-level value-flow graph into a variable-level constraint graph as shown in Figure 5. To preserve flow-sensitivity—,*i.e.*, to distinguish the points-to set of a variable at different program statements—each potentially defined variable v at statement ℓ is renamed as v_ℓ, with $\mathcal{P}(v_\ell) = \texttt{OUT}_\ell(v)$. Specially, since a strong update happens at `*m = a`, $\texttt{OUT}_8(o^q)$ does not contribute to the result of $\texttt{OUT}_7(o^q)$ and there is no constraint edge from o_8^q to o_7^q, resulting in no SCC for o^q. When the points-to set of `c` changes, the constraint graph explicitly encodes the propagation paths to o^p and o^q as well as the SCCs which they belongs to, enabling precise identification of which predecessor's old results can be safely reused.

2.4.2 Propagation of Points-to Additions and Deletions. Considering the example shown in Figure 2, `*x = b` at Line 2 performs a strong update and the points-to set of variable c includes only b's points-to set $\{o^a, o^b\}$. When Line 2 is removed, `*x = a` → `c = *x` becomes active and the points-to set of c firstly is cleared and regains $\{o^a\}$. IPA handles this incrementally by first propagating the reduced set ($\{o^a, o^b\}$) forward, and then propagating the increased set ($\{o^a\}$). This two-phase process introduces unnecessary overhead by repeatedly processing the same constraint nodes and performing SCC detection to break and reconstruct affected SCCs. For example, we extend the running example by incorporating the loop shown in Figure 6. This extension reuses variable a and c from Figure 2. The load at Line 11 induces the object-specific constraints $o_{11}^a \to e$ and $o_{11}^b \to e$. The store at Line 12, together with the loop-carried value flow to the next iteration's load, induces the return constraint $e \to o_{11}^a$. Hence, $\{o_{11}^a, e\}$ forms an SCC. Under delete-then-insert propagation, the deletion phase temporarily changes $\mathcal{P}(c)$ from $\{o^a, o^b\}$ to $\emptyset$ and removes $o_{11}^a \to e$, breaking the SCC and leaving only $e \to o_{11}^a$. The insertion phase then restores o^a, re-adds $o_{11}^a \to e$, and reconstructs the same SCC. Since SCC detection is linear in the graph size, repeatedly destroying and rebuilding such SCCs can dominate incremental-analysis time.

```
10  while (...) {
11    e = *c;
12    *a = e;
13  }
```

(a) Code extension



(b) Local constraint graph

Figure 6: Downstream extension of the running example and its relevant local constraint graph.

Algorithm 1 Incremental Analysis Algorithm for full-sparse flow-sensitive pointer analysis

Input: old points-to set $\mathcal{P}$, code changes $\mathcal{C}$, old VFG G
Output: flow pointer analysis result $\mathcal{P}'$ after modified

1: $G' \leftarrow$ IncVFGbySILVA(G,$\mathcal{C}$) ▷ incrementally construct VFG by SILVA
2: $\mathcal{G} \leftarrow$TranVFGtoCG(G) ▷ transform VFG into constraint graph
3: $R, \mathcal{N}_T \leftarrow$SCCDetection($\mathcal{G}$) ▷ perform SCC detection initially and merge SCCs
4: $N^+, N^-, E^+, E^- \leftarrow$ GraphDiff(G,G')
5: $\mathcal{E}^-, \mathcal{E}^+$=InitCGchanges($N^+, N^-, E^+, E^-$) ▷ initialize deleted and inserted constraint edges
6: **while** $\mathcal{E}^- \neq \emptyset$ or $\mathcal{E}^+ \neq \emptyset$ **do**
7: **for** $e_C : p \to q \in \mathcal{E}^+$ **do**
8: P-InsCopy(e_C,R) ▷ handle inserted constraint edges
9: **end for**
10: $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}^+ \backslash \mathcal{E}^-$
11: $R, \mathcal{N}_T \leftarrow$SCCDetection($\mathcal{G}$) ▷ perform SCC detection and merge SCCs
12: **for** $e_C : p \to q \in \mathcal{E}^-$ **do**
13: P-DelCopy(e_C,R) ▷ handle deleted constraint edges
14: **end for**
15: $\mathcal{E}^-, \mathcal{E}^+ \leftarrow \emptyset$
16: $\mathcal{P}' \leftarrow$FilterAndPropagate($\mathcal{N}_T$, $\mathcal{P}$) ▷ propagate and filter following topological order
17: **end while**

To address these limitations, we propose a unified incremental algorithm that jointly handles both code deletions and insertions in a single pass. In the downstream-loop example, when the points-to set of `c` is potentially decreased by removing both o^a and o^b, our algorithm uses not only the remaining edges but also the newly added edges to filter removed objects. Since the newly activated definition still contributes o^a, only o^b is removed and edge $o_{11}^a \to e$ remains alive. Consequently, the SCC containing o_{11}^a and e is not destroyed, avoiding redundant SCC detection and reconstruction.

3 Incremental Full-Sparse Flow-Sensitive Pointer Analysis

3.1 Overall Incremental Algorithm

Our incremental full-sparse flow-sensitive pointer analysis algorithm proceeds in two phases. The first phase lies in Line 1-6 in Algorithm.1. The inputs of the algorithm are old flow- sensitive pointer analysis result $\mathcal{P}$, old VFG G, and code changes $\mathcal{C}$. Firstly, we use the existing tool SILVA [43] to incrementally compute the updated value-flow graph G' based on the code changes $\mathcal{C}$ and old VFG G. Then we transform the old VFG G into a constraint graph $\mathcal{G}$.[1] An SCC detection is performed on the constraint graph for subsequent use. Based on the detected SCCs, we collapse each SCC into a single representative node on the constraint graph, yielding the representative information R of each node, and obtain the topological order $\mathcal{N}_T$. Afterwards, we have implemented GraphDiff to

[1]The detailed transformation rules are presented in **Section 3.2**.

$$\frac{p = alloca_o}{\mathcal{P}(p) \leftarrow \mathcal{P}(p) \cup \{o_\ell\}}\ \textsc{Alloc} \quad \frac{\ell : p = q}{\mathcal{E} \leftarrow \mathcal{E} \cup \{p \leftarrow q\}}\ \textsc{Copy} \quad \frac{\ell : p = \phi(p_1, ...p_n)}{\mathcal{E} \leftarrow \mathcal{E} \cup \{p \leftarrow p_i\}}\ \textsc{Phi}$$

$$\frac{\ell : p = \&q.f}{\mathcal{E} \leftarrow \mathcal{E} \cup \{p \xleftarrow{f} q\}}\ \textsc{Field} \quad \frac{\ell : p = *q, o \in pts(q), \ell' \overset{o}{\rightsquigarrow} \ell}{\mathcal{E} \leftarrow \mathcal{E} \cup \{p \leftarrow o_{\ell'}\}}\ \textsc{Load}$$

$$\frac{\ell : *p = q \quad \ell' \overset{o'}{\rightsquigarrow} \ell}{\begin{cases} & |\mathcal{P}(p)| = 0 \\ \mathcal{E} \leftarrow \mathcal{E} \cup \{o_\ell \leftarrow q\}, o' \neq o \Rightarrow \mathcal{E} \leftarrow \mathcal{E} \cup \{o'_\ell \leftarrow o'_{\ell'}\} & |\mathcal{P}(p)| = 1 \wedge o \in \mathcal{P}(p) \\ \mathcal{E} \leftarrow \mathcal{E} \cup \{o'_\ell \leftarrow o'_{\ell'}\} \cup \{o_\ell \leftarrow q\} & |\mathcal{P}(p)| > 1 \wedge o \in \mathcal{P}(p) \end{cases}}\ \textsc{Store}$$

Figure 7: Rules of transforming VFG to constraint graph

compare the new value-flow graph G' with the old one G and obtain the inserted nodes N^+, deleted nodes N^-, inserted value-flow edges E^+, deleted value-flow edges E^-. [2]

In the second phase which lies in Line 6-17, we firstly handle the inserted constraint edges $\mathcal{E}^+$ according to the representative information. We then perform SCC detection on the modified constraint graph and obtain the new representative information for the handling of the removed constraint edges $\mathcal{E}^-$. Following the topological order $\mathcal{N}_T$ derived from SCC detection, we compute and propagate the increases and decreases for each constraint node on the constraint graph $\mathcal{G}'$. These changes in points-to set may further trigger more copy constraint edges being deleted or inserted due to load and store statements. We start the next iteration with these newly deleted edges $\mathcal{E}^-$ and inserted edges $\mathcal{E}^+$, and repeat this process until no further edge updates occur. [3]

3.2 Transform VFG to Constraint Graph

For each address-taken variable o potentially defined at a store statement ℓ, we create a renamed version o_ℓ with $\mathcal{P}(o_\ell) = \texttt{OUT}_\ell(o)$. After renaming, we build the constraint graph whose nodes are either top level variables or renamed address taken variables. Inclusion based constraint edges between variables are added according to the semantic transfer function associated with each type of statements in flow-sensitive pointer analysis.

Figure 7 illustrates the rules of transformation. For each **alloc** statement, we introduce a additional memory object $\mathcal{P}(p) \leftarrow \mathcal{P}(p) \cup \{o\}$ for p. For **copy** statements $p = q$ and **phi** instructions, we insert a copy constraint edge $p \leftarrow q$, enforcing the subset constraint $\mathcal{P}(q) \subseteq \mathcal{P}(p)$. For load and store statement ℓ, since we only represent $\texttt{OUT}_\ell(o)$ by $\mathcal{P}(o_\ell)$ and do not present $\texttt{IN}_\ell(o)$, we represent $\texttt{IN}_\ell(o)$ by union the $\mathcal{P}(o_{\ell'})$ is identical to $\texttt{OUT}_{\ell'}(o)$ if $\ell' \overset{o}{\rightsquigarrow} \ell$. Consequently, for **load** statements ($p = *q$), for every object $o \in \mathcal{P}(q)$ pointed to by the base variable q, we insert a copy constraint edge $p \leftarrow o_{\ell'}$, effectively propagating the points-to information from the address-taken o at the predecessor ℓ' to variable p.

For each **store** statement ($\ell : *p = q$), we follow the standard SFS paradigm. To ensure correctness, we design three specific constraint generation rules to align with these distinct semantic transfer functions:

- $|\mathcal{P}(p)| = 0$: we does not insert any constraint edges.

[2] The graph changes in VFG will result in the deletion $\mathcal{E}^-$ and insertion $\mathcal{E}^+$ of constraint edges which is elaborated in **Section 3.3**.

[3] The precise mechanism for this efficient delta propagation is detailed in **Section 3.4**.

$$\frac{\ell : p = alloc_o \in N^-}{\Delta_p^- \leftarrow \Delta_p^- \cup \{o\}}\ \textsc{DelAlloc} \quad \frac{\ell : p = q \in N^-}{\mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{p \leftarrow q\}}\ \textsc{DelCopy}$$

$$\frac{\ell : p = \&q.f \in N^-}{\mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{p \xleftarrow{f} q\}}\ \textsc{DelField} \quad \frac{\ell : p = \phi(p_1, ...p_n), i \in [1, n] \in N^-}{\mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{p \leftarrow p_i\}}\ \textsc{DelPhi}$$

$$\frac{\ell : p = *q \in G, o \in \mathcal{P}(p), \ell' \overset{o}{\rightsquigarrow} \ell \in E^-}{\mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{p \leftarrow o_{\ell'}\}}\ \textsc{DelLoad}$$

$$\frac{\ell : *p = q \in G, \ell' \overset{o}{\rightsquigarrow} \ell \in E^-}{\ell \in N^- \Rightarrow \mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{o_\ell \leftarrow q \mid o \in \mathcal{P}(p)\}}\ \textsc{DelStore}$$

$$\begin{cases} & |\mathcal{P}(p)| = 0 \\ \mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{o_\ell \leftarrow o_{\ell'}\} & |\mathcal{P}(p)| = 1 \wedge o \notin \mathcal{P}(p) \\ \mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{o_\ell \leftarrow o_{\ell'}\} & |\mathcal{P}(p)| > 1 \end{cases}$$

Figure 8: Remove VFG nodes and edges

- $|\mathcal{P}(p)| = 1 \wedge o \in \mathcal{P}(p)$: we insert a constraint edge $o_\ell \leftarrow q$, propagating the points-to set of q to o_ℓ. For all other variable, o' (where $o' \neq o$), we preserve their values propagated from predecessors by adding copy constraint edges $o'_\ell \leftarrow o'_{\ell'}$.
- $|\mathcal{P}(p)| > 1 \wedge o \in \mathcal{P}(p)$: we insert a copy constraint $o_\ell \leftarrow q$ and a copy constraint edge $o'_\ell \leftarrow o'_{\ell'}$ for all potential defined o' to preserve points-to information from the predecessor.

3.3 Initialize by Deleting and Inserting Edges According to VFG Changes

In this section, we initilize the removed and inserted constraint edges based on the VFG changes: E^+, E^-, N^-, N^+ and initialize both positive deltas Δ^+ and negative deltas Δ^- in the points-to set according to the constraint graph changes.

Delete VFG nodes and edges: as illustrated in Figure 8, For address, load, copy, phi, and field statements,the deletion process suffices to remove all incoming constraint edges whose target is the defined top-level variable For instance, deleting a copy statement $p = q$ simply involves removing the edge $p \leftarrow q$.

For **store** nodes ($\ell : *p = q$), when this node is deleted,*i.e.*,$\ell \in N^-$, the constraint edge $o_\ell \leftarrow q$ is removed if p points-to o. Afterwards, we remove the constraint edge which propagates points-to set of predecessors to this node based on p's points-to set:

- $|\mathcal{P}(p)| = 0$: there is no constraint edge propagating the points-to set from the predecessor. Upon deletion, we do nothing.
- $|\mathcal{P}(p)| = 1 \wedge o \in \mathcal{P}(p)$: for all $o \notin \mathcal{P}(p)$, since there exists a constraint edge $o_\ell \leftarrow o_{\ell'}$, upon the value flow edge $\ell' \overset{o}{\rightsquigarrow} \ell$ is deleted, we remove $o_\ell \leftarrow o_{\ell'}$.
- $|\mathcal{P}(p)| > 1 \wedge o \in \mathcal{P}(p)$: we remove the constraint edges $o_\ell \leftarrow o_{\ell'}$ upon the value flow edge $\ell' \overset{o}{\rightsquigarrow} \ell$ is deleted.

Insert VFG nodes and edges. As illustrated in Figure 9, for insertion of **load** statement ($\ell : *p = q$) and its inserted value flow edge $\ell' \overset{o}{\rightsquigarrow} \ell$, we insert the constraint edge $o_{\ell'} \leftarrow p$. For insertion of **store** statement ($\ell : *p = q$), we firstly insert the newly generated constraint edge $o_\ell \leftarrow q$ if p points-to o. Afterwards,we insert the constraint edge if a value flow edge $\ell' \overset{o}{\rightsquigarrow} \ell$ is inserted to this node based on p's points-to set:

- $|\mathcal{P}(p)| = 0$: no constraint edges are inserted.

$$\frac{\ell : p = alloc_o \in N^+}{\Delta_p^+ \leftarrow \Delta_p^+ \cup \{o\}}\ \textsc{InsAlloc} \qquad \frac{\ell : p = q \in N^+}{\mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{p \leftarrow q\}}\ \textsc{InsCopy}$$

$$\frac{\ell : p = \phi(p_1, ...p_n), i \in [1,n] \in N^+}{\mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{p \leftarrow p_i\}}\ \textsc{InsPhi} \qquad \frac{\ell : p = q \in N^+}{\mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{p \xleftarrow{f} q\}}\ \textsc{InsField}$$

$$\frac{\ell : p = *q \in G' \quad o \in \mathcal{P}(q) \quad \ell' \overset{o}{\rightsquigarrow} \ell \in E^+}{\mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{p \leftarrow o_{\ell'}\}}\ \textsc{InsLoad}$$

$$\frac{\ell : *p = q \in G', \ell' \overset{o}{\rightsquigarrow} \ell \in E^+}{\ell \in N^+ \Rightarrow \mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{o_\ell \leftarrow q \mid o \in \mathcal{P}(p)\}}\ \textsc{InsStore}$$

$$\begin{cases} & |\mathcal{P}(p)| = 0 \\ \mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{o_\ell \leftarrow o_{\ell'}\} & |\mathcal{P}(p)| = 1 \wedge o \notin \mathcal{P}(p) \\ \mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{o_\ell \leftarrow o_{\ell'}\} & |\mathcal{P}(p)| > 1 \end{cases}$$

Figure 9: Insert VFG nodes and edges

- $|\mathcal{P}(p)| = 1 \wedge o \in \mathcal{P}(p)$: upon a value flow edge $\ell' \overset{o}{\rightsquigarrow} \ell$ in inserted, constraint edge $o_\ell \leftarrow o_{\ell'}$ is added for $o \notin \mathcal{P}(p)$.
- $|\mathcal{P}(p)| > 1 \wedge o \in \mathcal{P}(p)$: upon a value flow edge $\ell' \overset{o}{\rightsquigarrow} \ell$ is inserted, we insert a constraint edge $o_\ell \leftarrow o_{\ell'}$ to preserve the incoming flow.

Example 3.1. Considering the example in Fig. 3, upon deleting node 2, we modify the value-flow graph by removing value flow edges $1 \overset{o^x}{\rightsquigarrow} 2$ and $2 \overset{o^x}{\rightsquigarrow} 3$, and inserting a new value flow edge $1 \overset{o^x}{\rightsquigarrow} 3$. Correspondingly, we update the constraint graph: (1) The direct constraint $o_2^x \leftarrow b$ is removed according to the Rule DelStore. (2) For the deleted incoming value flow $1 \overset{o^x}{\rightsquigarrow} 2$, since the strong update at node 2 previously killed the definition from Node 1, no predecessor constraint $o_2^x \leftarrow o_1^x$ existed to be removed according to the Rule DelStore. (3) For the deleted outgoing flow $2 \overset{o^x}{\rightsquigarrow} 3$, we remove the constraint $c \leftarrow o_2^x$ following the DelLoad rule. (4) Conversely, for the newly inserted flow $1 \overset{o^x}{\rightsquigarrow} 3$, we add the constraint $c \leftarrow o_1^x$ according to the InsLoad rule.

3.4 Filter and Propagate Difference of Points-to Set on Constraint Graph

After identifying the structural changes in the constraint graph, we first initialize the potential changes in variables' points-to sets based on the insertion and deletion of copy edges (Rules **P-InsCopy** and **P-DelCopy**). Given these initial deltas, we apply the **Filter** rule to eliminate spurious points-to updates using the *change-local property* of the constraint graph and the **Propagate** rule to propagate the actual changes along the graph. Such propagation may further trigger deletion or insertion of constraint edges (**P-Store** and **P-Load**). After each propagation round, the newly inserted or deleted constraint edges are treated as the input for the next iteration until a fixed point is reached. The inference rules governing this process are presented in Figure 10.

Initialization of Deltas in Points-to sets. When a copy constraint edge $p \leftarrow q$ is inserted, if p and q previously share the same representative node,*i.e.*,$R(p) = R(q)$ which means the points-to sets of both p and q are the same, thus q does not introduce any additional objects. Otherwise, we initialize the positive delta of p as $\Delta_p^+ \leftarrow \Delta_p^+ \cup \mathcal{P}(q)$, indicating that p may now point to new objects in q's points-to set. We then process the deleted constraint edges $p \leftarrow q$. If p and q are previously in the same SCC, then this SCC may be broken. We perform an SCC detection on the updated constraint graph. If p and q do not belong to the same SCC in the updated graph, we initialize the negative delta as $\Delta_p^- \leftarrow \Delta_p^- \cup \mathcal{P}(q)$, indicating that p may lose the objects in q's points-to set.

$$\frac{q \to p \in \mathcal{E}^+}{R(p) \neq R(q) \Rightarrow \Delta_p^+ \leftarrow \Delta_p^+ \cup \mathcal{P}(q)}\ \textsc{P-InsCopy}$$

$$\frac{q \to p \in \mathcal{E}^-}{R(p) \neq R(q) \Rightarrow \Delta_p^- \leftarrow \Delta_p^- \cup \mathcal{P}(q)}\ \textsc{P-DelCopy}$$

$$\frac{p, \Delta_p^-, \Delta_p^+}{\begin{array}{l}\forall p \leftarrow q \in \mathcal{E} \Rightarrow \Delta_p^- \leftarrow \Delta_p^- \setminus \mathcal{P}(q) \\ \Delta_p^+ \leftarrow \Delta_p^+ \setminus \mathcal{P}(p) \quad \Delta_p^+ \leftarrow \Delta_p^+ \setminus \Delta_p^- \quad \Delta_p^- \leftarrow \Delta_p^- \cap \mathcal{P}(p) \\ \mathcal{P}(p) \leftarrow \mathcal{P}(p) \backslash \Delta_p^- \cup \Delta_p^+ \quad \textsc{Propagate}(p, \Delta_p^-, \Delta_p^+)\end{array}}\ \textsc{Filter}$$

$$\frac{p, \Delta_p^-, \Delta_p^+}{\begin{array}{l}\forall s \leftarrow p \in \mathcal{E}, \Delta_s^+ \leftarrow \Delta_s^+ \cup \Delta_p^+, \Delta_s^- \leftarrow \Delta_s^- \cup \Delta_p^- \\ \forall \ell : *p = q \in G',\ \textsc{P-Store}(\ell : *p = q, \Delta_p^-, \Delta_p^+) \\ \forall \ell : q = *p \in G',\ \textsc{P-Load}(\ell : q = *p, \Delta_p^-, \Delta_p^+)\end{array}}\ \textsc{Propagate}$$

$$\frac{\ell : p = *q, o^+ \in \Delta_p^+, o^- \in \Delta_p^+, \ell' \overset{o^+}{\rightsquigarrow} \ell, \ell' \overset{o^-}{\rightsquigarrow} \ell}{\mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{p \leftarrow o_{\ell'}^-\} \quad \mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{p \leftarrow o_{\ell'}^+\}}\ \textsc{P-Load}$$

$$\frac{\ell : *p = q, \ell' \overset{o}{\rightsquigarrow} \ell, ,\ o^+ \in \Delta_p^+,\ o^- \in \Delta_p^+,}{\begin{array}{l}|\mathcal{P}(p)| = 0 \Rightarrow \mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{e_C | e_C : o_\ell \leftarrow o_{\ell'} \in \mathcal{E}\} \\ |\mathcal{P}(p)| = 1 \Rightarrow \begin{cases} \mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{e_C | e_C : o_\ell \leftarrow o_{\ell'} \in \mathcal{E}\} & o \in \mathcal{P}(p) \\ \mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{e_C | e_C : o_\ell \leftarrow o_{\ell'} \notin \mathcal{E}\} & o \notin \mathcal{P}(p) \end{cases} \\ |\mathcal{P}(p)| > 1 \Rightarrow \mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{e_C | e_C : o_\ell \leftarrow o_{\ell'} \notin \mathcal{E}\} \\ \mathcal{E}^- \leftarrow \mathcal{E}^- \cup \{o_\ell^- \leftarrow q\}, \mathcal{E}^+ \leftarrow \mathcal{E}^+ \cup \{o_\ell^+ \leftarrow q\}\end{array}}\ \textsc{P-Store}$$

Figure 10: Rules of filtering and propagating increase and decrease in the points-to set

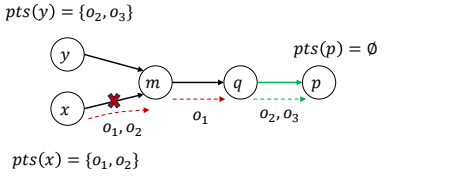


Figure 11: Simultaneous propagation of points-to additions and deletions.

Topological Propagation and Filtering. We process constraint nodes in topological order to handle dependencies correctly. The initial deltas Δ^+ and Δ^- derived from edge insertions/deletions or propagated from predecessors represent only *candidate* changes.

- **Filtering Decreases (Δ_p^-):** for each object o in the candidate Δ_p^-, we inspect all existing predecessors $\{q \mid p \leftarrow q \in \mathcal{G}\}$. If there exists any predecessor q such that $o \in \mathcal{P}(q)$, we remove o from Δ_p^- and $\mathcal{P}(p)$ retains o.
- **Filtering Increases (Δ_p^+):** if a candidate increased object o is already present in $\mathcal{P}(p)$, it constitutes a redundant update. Thus, we remove those objects from Δ_p^+ which is already in $\mathcal{P}(p)$.

A critical challenge in incremental analysis is handling "interleaved" updates, where a variable might receive both additions

and deletions in the same iteration. For instance, as shown in Figure 11, if a new edge $q \to p$ is inserted while q itself is losing a target o_1 due to an upstream deletion $x \to m$, eagerly adding $\mathcal{P}(q) = \{o_1, o_2, o_3\}$ to Δ_p^+ would incorrectly propagate o_1 to p. To address this, the **Filter** rule applies strict set difference operations: we remove any element found in Δ_p^- from Δ_p^+ ,*i.e.*, $\Delta_p^+ \leftarrow \Delta_p^+ \setminus \Delta_p^-$). This also guarantees that Δ_p^+ and Δ_p^- are mutually exclusive. We subtract $\Delta_p^- = \{o_1\}$ from $\Delta_p^+ = \{o_1, o_2, o_3\}$, resulting in the updated $\Delta_p^+ = \{o_2, o_3\}$. The **Filter** rule also constrains the negative delta to include only elements that are actually present in the current points-to set ,*i.e.*, $\Delta_p^- \leftarrow \Delta_p^- \cap \mathcal{P}(p)$). In this example, assuming p is initially empty or does not contain o_1, the operation yields $\Delta_p^- = \{o_1\} \cap \mathcal{P}(p) = \emptyset$.

Handling Side Effects on Load/Store Statements. Finally, we update the points-to set of p locally: $\mathcal{P}(p) \leftarrow (\mathcal{P}(p) \setminus \Delta_p^-) \cup \Delta_p^+$, and propagate the refined Δ_p^+ and Δ_p^- to p's successors. Changes in the points-to set of a pointer variable p (Δ_p^+ or Δ_p^-) can trigger more constraint edge updates.

Load Statements ($\ell : p = *q$). As defined in rule **P-Load**, for any object o added to q's points-to set,*e.g.*,$o \in \Delta_q^+$), we insert a copy constraint edge $p \leftarrow o_{\ell'}$. Conversely,constraint edge $p \leftarrow o_{\ell'}$ is removed for object $o \in \Delta_q^-$.

Store Statements ($\ell : *p = q$). As defined in rule **P-Store**, we add a constraint edge $o_\ell^+ \leftarrow q$ and remove $o_\ell^- \leftarrow q$ for $o^+ \in \Delta_p^+$ and $o^- \in \Delta_p^-$ respectively. Additionally, we also incrementally modify whether the points-to information of current variable should be propagated from the predecessor depending on p's points-to set:

- $|\mathcal{P}(p)| = 0$: the store becomes a no-op and any existing bypass edges $o_\ell \leftarrow o_{\ell'}$ is deleted.
- $|\mathcal{P}(p)| = 1$: for the unique target object $o \in \mathcal{P}(p)$, we delete the constraint edge $o_\ell \leftarrow o_{\ell'}$ to kill the points-to information propagated from predecessor ℓ' if it exists . For all other objects $o' \notin \mathcal{P}(p)$, the bypass edge $o'_\ell \leftarrow o'_{\ell'}$ is inserted if missing .
- $|\mathcal{P}(p)| > 1$: For any object o that reaches this statement, we must insert the bypass edge $o_\ell \leftarrow o_{\ell'}$ if it is missing, restoring the points-to information propagated from the predecessor.

Lemma 3.2 (Monotonicity). *Points-to reductions and expansions induce only edge deletions and insertions, respectively, which in turn can only shrink and enlarge points-to sets.*

Proof. Consider a store statement $\ell : *p = q$. whose generated constraint edges depend on $\mathcal{P}(p)$. If $\mathcal{P}(p)$ decreases from multiple objects to a singleton o, the update changes from weak to strong: all bypass edges for o and all edges from q to objects removed from $\mathcal{P}(p)$ are deleted, while bypass edges for the remaining objects are preserved. If the store becomes a no-op, all associated constraint edges are removed. Therefore, the resulting target edge set can only decrease. Conversely, if $\mathcal{P}(p)$ expands, causing a transition from a no-op to a strong or weak update, or from a strong update to a weak update, the target edge set can only increase. Edge deletions only trigger Δ^- and further edge deletions, whereas edge insertions only enlarge them and trigger further insertions. □

Definition 3.3 (Object points-to cycle). For object nodes o, o', write $o \rightsquigarrow_{\text{pt}} o'$ iff $o' \in \mathcal{P}(o)$. An object points-to cycle is a nonempty sequence $o_1, \ldots, o_n$ such that $o_i \rightsquigarrow_{\text{pt}} o_{i+1}$ for $1 \le i < n$ and $o_n \rightsquigarrow_{\text{pt}} o_1$. The points-to relation is *object-acyclic* iff it contains no such cycle.

Theorem 3.4. *If the points-to relation remains object-acyclic during the analysis, the interleaved analysis algorithm terminates.*

Proof. Let $S_k = \langle G_k, \mathcal{E}_k^-, \mathcal{E}_k^+ \rangle$ be the state after round k, where G_k is the constraint graph annotated with the current points-to sets and $\mathcal{E}_k^-$ and $\mathcal{E}_k^+$ are the load/store-derived edge deletions and insertions. If the algorithm did not terminate, then $S_i = S_j$ for some $i < j$. Since filtering makes $\mathcal{E}_m^- \cap \mathcal{E}_m^+ = \emptyset$, every edge deleted between these two equal states must be reinserted in a later round, and conversely. Thus, $D_{i,j}^- = \bigcup_{i \le m < j} \mathcal{E}_m^- = \bigcup_{i \le m < j} \mathcal{E}_m^+ = D_{i,j}^+ \neq \emptyset$. Let $e \Rightarrow e'$ mean that the points-to reduction caused by derived-edge deletion e triggers deletion e' through P-Load or P-Store according to Lemma 3.2. Since every update in $D_{i,j}^-$ must be reversed before S_j, following these causes yields a cycle $e_1 \Rightarrow e_2 \Rightarrow \cdots \Rightarrow e_n \Rightarrow e_1$. Let $e_r (1 \le r \le n)$ be induced when the store $*p_r = a_r$ loses target object o_r. If e_r causes e_{r+1}, the reduction propagated through o_r removes o_{r+1} at the next store; thus $o_{r+1} \in \mathcal{P}(o_r)$, with indices modulo n. Hence, $o_2 \in \mathcal{P}(o_1)$, $o_3 \in \mathcal{P}(o_2), \ldots,$ $o_1 \in \mathcal{P}(o_n)$, which gives $o_1 \rightsquigarrow_{\text{pt}} o_2 \rightsquigarrow_{\text{pt}} \cdots \rightsquigarrow_{\text{pt}} o_n \rightsquigarrow_{\text{pt}} o_1$. This contradicts the acyclicity assumption. Therefore, the algorithm terminates. □

Theorem 3.5 (Consistency). *Under the object-acyclicity condition , for each statement type in the updated program, the constraint graph and points-to sets produced by our incremental method are consistent with those derived from the data-flow functions of full-sparse flow-sensitive pointer analysis.*

Proof Sketch. Consider a copy statement $\ell : p = q$; the same argument applies to field and phi statements. We prove that $\mathcal{P}(q) \subseteq \mathcal{P}(p)$. Assume, for contradiction, that some object $o \in \mathcal{P}(q)$ but $o \notin \mathcal{P}(p)$. If $o \in \Delta_p^-$, the deletion-filtering rule $\Delta_p^- \leftarrow \Delta_p^- \setminus \mathcal{P}(q)$ removes o from Δ_p^-. Otherwise, o must be absent from Δ_p^+. However, an object already in Δ_p^+ can be filtered out only if it is already in $\mathcal{P}(p)$. Another case where $q \to p$ is newly inserted also results in $o \in \Delta_p^+$. Thus, the assumption is contradicted and $\mathcal{P}(q) \subseteq \mathcal{P}(p)$. For Store and load statements, rule **P-Store** and **P-Load** guarantee that constraint graph are consistent with those derived from data-flow functions.

We prove the minimality under object-acyclic condition. Assume the resulting fixed point is non-minimal. That is if o_1 is to be removed from $\mathcal{P}(p)$, $r \to p$ filters the removal of o_1 regardless of whether this edge is pre-existing or newly inserted during interleaved propagation. If removing o must eventually delete $r \to p$. There must exist a chain $e_1 \Rightarrow e_2 \Rightarrow \cdots \Rightarrow e_n$, where $e_n = (r \to p)$. Each step is induced by a store $*p_k = a_k$: losing target o_k deletes $a_k \to o_k$ and causes o_k to lose o_{k+1}, which implies $o_1 \in \mathcal{P}(p), o_2 \in \mathcal{P}(o_1), \ldots, p \in \mathcal{P}(o_{n-1})$, which contradicts object-acyclicity. Hence interleaved analysis will reach the least fixed point. □

Complete proofs of termination, constraint consistency, and least-fixed-point convergence are provided in the supplementary material archived with the artifact (Section 8).

Table 4: Subject programs and its characteristics. Prog, Loc,Nodes,Edges,Pointers,Objects,Funs,CS represents programs, the number of lines of code, nodes in VFG, number of edges in VFG, number of pointer variables , number of memory objects, number of functions, number of callsite.

Prog	Loc	Nodes	Edges	Pointer	Object	Funs	CS
janet	85274	134420	200093	176155	4300	1544	9676
zstd	95939	380797	558708	582876	49616	2348	24627
tmux	74322	231969	374530	207267	6215	2228	16524
astyle	100424	281375	467675	321716	4845	712	20658
nginx	170660	171730	284550	170058	2568	1424	9206
sqlite	525332	457787	756968	520887	8471	2566	28304

4 Evaluation

4.1 Implementation

We implemented our approach, named **IncSFS**, on top of the SVF framework (v2.7) [37], utilizing LLVM 14.0.1 as the underlying infrastructure. SVF is a widely-used, actively maintained open-source framework that supports full-sparse flow-sensitive pointer analysis. To handle the initial incremental updates of the Value-Flow Graph (VFG), we integrated the Silva [43] framework into IncSFS.

We designed our experiments to answer the following questions:

- **RQ1 (Performance Gain):** How much speedup does **IncSFS** achieve compared to whole-program analysis and traditional *Reset–Recompute* strategy?
- **RQ2 (Advantage over Two-Phase IPA):** How much performance benefit does IncSFS gain from its **unified propagation** strategy on the **constraint graph** compared to the traditional delete-then-insert propagation?
- **RQ3 (Scalability):** How does the analysis time of **IncSFS** scale with an increasing number of commits?
- **RQ4 (Correctness):** Are the points-to results produced incrementally by **IncSFS** identical to those obtained from the whole-program analysis?

4.2 Benchmark and Experimental Setup

Subject Programs. We evaluated IncSFS on a suite of six large-scale, real-world programs as shown in table 4 characterized by frequent updates, which were selected based on their widespread adoption in prior flow-sensitive pointer analysis literature [2, 43].

Dataset Construction. To assess the impact of code change magnitude, we constructed datasets based on four distinct commit intervals: 1, 10, 20, and 30. An interval of N represents the evolution between a baseline commit and a version N commits forward. For each program and interval, we randomly sampled 10 distinct pairs of old and new versions from the repository history. For every interval, we exclude pairs with no source-code differences between the old and new versions in the process of random selection. The reported performance for each interval is derived from the average analysis time of these 10 sampled pairs. Upon manual inspection, 81.67% of our randomly sampled pairs involve modifications such as product releases, bug fixes, and code refactoring.

Source-to-IR Mapping and IR Noise. Since minor source edits may cause noisy whole-IR differences, Our current incremental approach have to also handle IR deletions followed by insertions. we map source changes directly to the corresponding deleted IR fragment P_d in the old IR I_o and inserted IR fragment P_i in the new IR I_n. We measure the deletion time T_d by removing P_d from I_o. For insertion, we obtain an equivalent intermediate state by fully analyzing I_n, removing P_i, and then reinserting it to measure T_i. The reported incremental time is $T_d + T_i$, including only the transformation from the VFG to the constraint graph, SCC detection, and incremental propagation. The reported time of incremental analysis excludes the parts of LLVM IR generation, source-to-IR mapping, and incremental VFG maintenance which is a common prerequisite shared by all incremental algorithms evaluated.

Baselines. We implemented three incremental strategies: Reset-Recompute, the adapted SILVA two-phase delete-then-insert propagation, and IncSFS's unified single-phase propagation. To answer **RQ1**, we compare it against the classical Whole-Program Analysis (Full) [15] since it is the standard implementation provided by SVF; the Reset-Recompute strategy, which reset and recomputes all nodes reachable from the changed VFG nodes/edges. We explicitly exclude the *Restart-Iteration* strategy from our evaluation since it inherently suffers from precision loss. To answer **RQ2**, we compare IncSFS with SILVA [43], the state-of-the-art flow-insensitive incremental pointer analysis. For both approaches, we measure the runtime of delta computation and propagation on the constraint graph transformed from the VFG. In this graph, all top-level and address-taken variables are converted into Full SSA form, allowing flow-sensitive pointer analysis to be solved flow-insensitively and SILVA to be directly applied. To answer **RQ3**, we evaluate representative small (`janet`), medium (`tmux`), and large (`sqlite`) programs against their initial versions, increasing the commit interval by 100 for `janet` and `tmux` and by 500 for `sqlite`, until incremental analysis approaches or exceeds full re-analysis. To answer **RQ4**, we compare the points-to sets of all maintained top-level variable produced by IncSFS against those computed by the full analysis across all experimental configurations. We also compute the average size of point-to set between IncSFS and the full analysis.

Experimental Environment. All experiments were conducted on a high-performance workstation running Ubuntu 20.04 LTS. The workstation features an AMD Ryzen Threadripper PRO 7995WX processor with 128 CPU cores operating at 2.5 GHz and 512 GB of physical memory.

4.3 Experimental Results

4.3.1 Comparison with full analysis and reset-recompute. Table 5 details the comparison result of total analysis time of **IncSFS** with **FullAnalysis** and the **Reset-Recompute (RR)** strategy. Columns **CNodes** and **CEdges** directly reflect the extent of source code modifications, serving as a concrete indicator of the incremental update magnitude. The exact old and new commit IDs and complete results for every sampled pair are archived with the artifact (Section 8).

Performance Analysis. The **BuildCG** phase alone accounts for 49.89% of the total IncSFS time. This is primarily we transform the whole VFG to the constraint graph eagerly instead of lazily created if needed, which results in the huge node and edges of the constraint graph and the cost of the initial SCC detection. Overall, IncSFS achieves a **9.60x** speedup over Full Analysis and a **5.84x** speedup over RR. The inefficiency of RR is highlighted by the **INodes** column: simple code changes often impact 20%–81% of the

Table 5: Comparison result with full analysis and reset-recompute, all times are measured in seconds. The "Prog" column presents the program. The "I" column presents the commit interval. The "CNode" and "CEdge" columns present changed nodes and changed edges respectively. The "INode" presents the influenced nodes by the changed nodes and edges in reset-recompute strategy. The "RR" presents the reset-recompute strategy. The "BuildCG" presents the time spent on transforming VFG into constraint graph.

Prog	I	CNode	CEdge	Full Time	INodes	RR	BuildCG	IncSFS
zstd	1	0.04%	0.00%	39.56 (4.16×)	40.50%	29.82 (3.14×)	4.32 (45.48%)	9.50
	10	0.24%	0.01%	42.27 (4.02×)	41.16%	30.17 (2.87×)	3.85 (36.61%)	10.51
	20	0.10%	0.01%	42.28 (3.85×)	53.20%	38.13 (3.48×)	5.18 (47.18%)	10.97
	30	0.19%	0.04%	42.19 (2.38×)	53.64%	39.34 (2.22×)	5.80 (32.64%)	17.76
janet	1	0.01%	0.04%	77.46 (9.00×)	23.72%	32.26 (3.75×)	5.95 (69.17%)	8.61
	10	0.01%	0.03%	92.31 (4.26×)	39.62%	68.63 (3.17×)	12.34 (56.90%)	21.68
	20	0.03%	0.06%	84.71 (5.48×)	36.00%	58.67 (3.80×)	11.10 (71.83%)	15.45
	30	0.04%	0.09%	88.83 (2.43×)	40.25%	63.55 (1.74×)	11.07 (30.23%)	36.62
tmux	1	0.02%	0.10%	475.84 (6.35×)	64.62%	313.29 (4.18×)	45.73 (61.04%)	74.92
	10	0.08%	0.11%	457.38 (4.85×)	71.41%	353.34 (3.75×)	47.10 (49.99%)	94.22
	20	0.16%	0.21%	448.74 (3.10×)	64.99%	337.33 (2.33×)	52.67 (36.39%)	144.74
	30	0.17%	0.48%	476.62 (3.23×)	72.32%	371.05 (2.52×)	54.07 (36.65%)	147.52
astyle	1	0.02%	0.02%	1271.98 (52.19×)	20.46%	351.29 (14.41×)	11.26 (46.21%)	24.37
	10	0.13%	1.93%	1288.84 (5.77×)	61.74%	1153.73 (5.16×)	34.13 (15.27%)	223.54
	20	0.27%	1.41%	1361.89 (7.07×)	61.99%	1167.50 (6.06×)	38.91 (20.21%)	192.59
	30	0.29%	1.43%	1358.15 (7.63×)	68.99%	1288.19 (7.23×)	29.07 (16.32%)	178.07
nginx	1	0.01%	0.01%	1387.76 (16.81×)	68.35%	1113.79 (13.49×)	56.51 (68.46%)	82.54
	10	0.02%	0.01%	1463.75 (8.57×)	75.93%	1246.75 (7.30×)	110.09 (64.47%)	170.75
	20	0.06%	0.05%	1525.32 (17.66×)	75.96%	1329.46 (15.39×)	43.20 (50.00%)	86.40
	30	0.09%	0.06%	1529.43 (8.34×)	75.97%	1345.46 (7.33×)	113.23 (61.73%)	183.43
sqlite	1	0.00%	0.01%	1670.40 (23.92×)	30.24%	485.39 (6.95×)	56.73 (81.23%)	69.83
	10	0.01%	0.03%	1763.95 (11.25×)	75.81%	1225.98 (7.82×)	110.56 (70.53%)	156.76
	20	0.06%	0.07%	1775.67 (8.76×)	68.34%	1066.22 (5.26×)	120.84 (59.61%)	202.72
	30	0.01%	0.01%	1787.30 (9.42×)	76.14%	1276.73 (6.73×)	131.42 (69.23%)	189.83
Avg				9.60×		5.84×	49.89%	

Table 6: Comparison result with SILVA. The "Prog" column presents the program. The "I" column presents the commit interval. The "PNode" presents the number of processed constraint nodes. The "SCC" presents the average number of SCC detections performed on the constraint graph.

Prog	I	IPA			$IncSFS^{RE-}$	IncSFS		
		Time (s)	PNodes	SCC	Time (s)	Time (s)	PNodes	SCC
janet	1	5.29 (49.8%)	200780.9 (47.1%)	1.7 (11.8%)	5.53	2.65	106208.2	1.5
	10	14.83 (37.0%)	736260.6 (49.7%)	18.4 (0.0%)	14.99	9.34	370634.4	18.4
	20	9.49 (54.2%)	494669.7 (60.0%)	2.5 (8.0%)	10.52	4.35	197944.3	2.3
	30	40.28 (36.6%)	1799976.6 (49.0%)	74.7 (0.7%)	47.02	25.55	917106.1	74.2
tmux	1	29.05 (-0.5%)	2268396.8 (45.2%)	6.1 (3.3%)	31.22	29.19	1243195.6	5.9
	10	49.00 (3.8%)	3546953.2 (38.9%)	123.3 (0.6%)	54.78	47.12	2166905.4	122.6
	20	136.54 (32.6%)	9830775.9 (59.8%)	5.6 (16.1%)	155.37	92.08	3953844.5	4.7
	30	134.79 (30.7%)	9957423.5 (59.0%)	18.4 (5.4%)	152.93	93.46	4084256.7	17.4
zstd	1	4.50 (-15.0%)	513078.8 (30.0%)	18.0 (1.7%)	5.06	5.18	359341.1	17.7
	10	7.28 (8.4%)	1264640.5 (57.6%)	79.3 (0.5%)	9.52	6.66	535809.9	78.9
	20	7.02 (17.5%)	1425544.6 (64.4%)	20.7 (2.9%)	8.13	5.79	506880.3	20.1
	30	13.81 (13.4%)	3059401.1 (60.3%)	25.5 (2.7%)	17.58	11.96	1214429.3	24.8
astyle	1	19.81 (33.8%)	826874.2 (39.4%)	17.0 (0.6%)	20.34	13.11	500778.3	16.9
	10	278.38 (32.0%)	12198190.9 (67.1%)	3118.8 (0.0%)	344.94	189.41	4013862.5	3118.0
	20	271.89 (43.5%)	13051482.3 (75.0%)	2591.1 (0.0%)	362.17	153.68	3259865.7	2590.5
	30	257.25 (42.1%)	12656285.0 (75.0%)	2725.7 (0.0%)	323.86	149.00	3161002.1	2725.0
nginx	1	24.66 (-5.6%)	400144.5 (50.0%)	2.6 (19.2%)	30.14	26.03	199902.8	2.1
	10	60.04 (-1.0%)	556158.6 (47.7%)	28.9 (2.4%)	71.94	60.66	290764.6	28.2
	20	39.89 (-8.3%)	572395.2 (48.7%)	4.0 (12.5%)	46.91	43.19	293619.2	3.5
	30	63.74 (-10.1%)	832000.8 (50.4%)	158.1 (0.5%)	75.73	70.20	412325.8	157.3
sqlite	1	11.80 (-11.1%)	395411.8 (36.3%)	1.2 (0.0%)	12.85	13.11	251969.5	1.2
	10	43.46 (-6.3%)	897227.7 (37.1%)	943.5 (0.0%)	48.33	46.19	563911.6	943.4
	20	82.85 (1.2%)	1681167.0 (49.6%)	838.6 (0.0%)	101.51	81.88	847423.3	838.5
	30	58.93 (0.9%)	1817195.9 (48.2%)	907.7 (0.0%)	67.53	58.41	940493.2	907.4
Avg.		15.8%	51.9%	3.7%	27.3%	–	–	–

VFG nodes due to transitive dependencies. Consequently, RR is forced to recompute the fix-point for a large portion of the program. Our approach only propagates deltas and reuse all old points-to result, resulting in fast reaching a fixed-point.

4.3.2 Comparison with SILVA. Table 6 details the comparison between IncSFS and SILVA. We introduce two fine-grained metrics to dissect the performance gains: **PNodes** and **SCC**. The results indicate that IncSFS is algorithmically superior: it reduces SCC detections by an average of **3.7%** and the number of propagated nodes by **51.9%**. Consequently, the overall analysis time is reduced by **15.8%**. The performance gain of IncSFS over SILVA primarily stems from the reduction in SCC detections. However, as observed in the results, the absolute reduction in the number of SCCs is not substantial. Some of the avoided SCCs are small-sized components, which incurs minimal computational overhead on incremental detection, the time savings derived from skipping these operations are consequently moderate. Compared with SILVA, to guarantee that the increase Δ^+ and decrease Δ^- remain *mutually exclusive* during simultaneous propagation, IncSFS performs additional set difference operations (*i.e.*, $\Delta^+ \setminus \Delta^-$). In scenarios where updates are naturally exclusive which SILVA handles without extra cost, our method still incurs this calculation overhead, resulting in a slight increase in the average processing time per node.

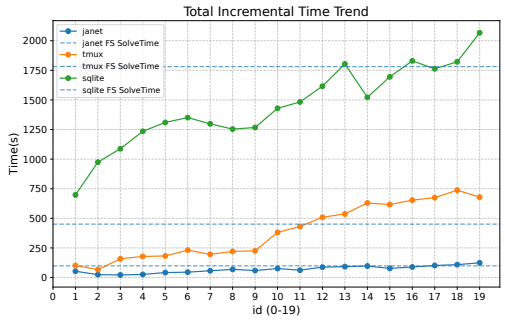


Figure 12: Running with linearly increasing number of commits.

To further separate the effect of set difference and redundancy elimination, Table 6 reports an ablation variant, $IncSFS^{RE-}$, which retains the set-difference filtering of IncSFS but disables redundant-propagation avoidance. Compared with $IncSFS^{RE-}$, IncSFS reduces the analysis time by 27.3%, demonstrating the effectiveness in eliminating redundant propagation, repeated load/store-induced edge updates, and unnecessary SCC maintenance. On the other hand, comparing $IncSFS^{RE-}$ with IPA shows that set-difference filtering introduces an average overhead of 11.5%.

4.3.3 Scalability. Figure 12 studies when the incremental overhead exceeds the cost of full re-analysis. The dashed line represents the runtime of full analysis for each program. The results show that IncSFS remains faster than full re-analysis across most modification scales. For insertion-only changes, incremental analysis is always faster than full re-analysis in our experiments. Therefore, we separately measure the ratios of deleted code and inserted code relative to the original codebase. For `janet`, `tmux`, and `sqlite`, the incremental overhead exceeds full re-analysis only under modification: deleted code accounts for 67.11%, 33.82%, and 6.95% of the original source code size, respectively, while inserted code reaches 150.92%, 68.67%, and 5.66% of the original source code size. These results suggest that the crossover point is mainly caused by large mixed deletion/insertion updates rather than by insertions alone.

Table 7: Equivalence between IncSFS and full analysis.

Prog	Version pairs	Compared variables	Mismatches
janet	40	1,766,214	0
nginx	40	2,653,743	0
tmux	40	2,947,271	0
astyle	40	2,531,033	0
sqlite	40	5,582,440	0
zstd	40	8,142,750	0
Total	240	23,623,451	0

4.3.4 Correctness verification. Figure 13 visualizes the average points-to set size of variables between our incremental approach and the baseline. As observed, all data points fall precisely on the line $y = x$, which demonstrates that IncSFS produces identical analysis results to the baseline across all tested benchmarks, confirming that our incremental updates preserve the strict flow-sensitivity of the original analysis.

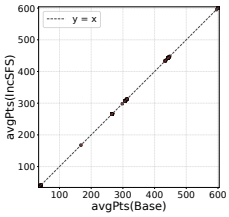


Figure 13: Avg pts-to set size (IncSFS vs Base).

Table 7 reports the number of variables whose points-to sets are compared. Across 23,623,451 variables in 240 configurations, IncSFS produces zero mismatches with full analysis.

4.4 Threats to Validity

We checked for object points-to cycles in `janet`, `tmux`, and `zstd`, but omitted the other programs due to prohibitive memory costs of object point-to cycle detection. Such cycles do occur in most of the checked programs. However, these cycles either fell outside the affected regions or, when cycle-related facts were removed by deletions, still retained independent non-cyclic support. Consequently, no cycle caused IncSFS to reach a non-minimal fixed point in any evaluated configuration, consistent with our exact comparison against full analysis.

5 Discussion

Memory usage. Our rename approach inherently introduces extra constraint nodes resulting in a maximum 3.8× memory overhead. By collapsing SCC nodes into a single representative, we significantly reduce computational costs compared to vanilla SFS. This space-for-time trade-off is indispensable for time-cost flow-sensitive analyses. Moreover memory issues is a problem also faced by other incremental analysis efforts [29, 43] to analyze larger-scale programs. We will investigate reductions in redundant renamed nodes in future work.

Downstream client impact. We have not yet integrated IncSFS into downstream clients such as bug detection. To estimate the potential impact, we use SVF's flow-sensitive pointer-analysis results to construct the value-flow graph and then run the SABER defect detector. In this pipeline, flow-sensitive pointer analysis accounts for 47% of the total analysis time on average across the benchmark programs evaluated in our experiments. Combining this fraction with IncSFS's average 9.60x speedup over full analysis, providing an expected end-to-end speedup of 1.73x ($1/((1-0.47)+0.47/9.60)$) for the full downstream analysis pipeline.

6 Related Work

Incremental Program Analysis. Incremental analysis has been extensively studied across dataflow analysis[27, 38],abstract interpretation [34, 35], program testing [9] and regression testing [32]. Reset-recompute is a widely used strategy in incremental analysis. Yur *et al.* [45] introduced the first incremental flow- and context-sensitive pointer analysis using this strategy. However, these methods incur redundant computations limiting the efficiency gain for large-scale pointers analysis as shown in our experiment results. Lu *et al.* [25] proposed an incremental flow-insensitive pointer analysis based on CFL-reachability, which incrementally answer queries spanning from variables to corresponding objects. Krainz *et al.* [17] represents the accesses of a method in a flow-sensitive way, but only handles code insertion. There are other incremental pointer analysis methods [39, 40] based on Datalog but hard to leverage the change-local properties based on constraint graph.

To overcome the overhead of recomputation, IPA [23, 24, 43] proposed efficient incremental algorithms by propagating deltas in points-to set. However, these approaches primarily target *flow-insensitive* analysis and fail to exploit the correlation between added and deleted statements [24]. In contrast, our approach targets *flow-sensitive* analysis and exploit a unified way.

Flow-Sensitive Pointer Analysis. In last decades, various optimizations have been proposed to improve performance of flow-sensitive pointer analysis. Li *et al.* [21] converts the flow-sensitive pointer analysis into a graph reachability problem on a value-flow graph. Ben *et al.* [14, 15] introduced *sparse* flow-sensitive pointer analysis. Barbar *et al.* [2] merge equivalent points-to sets via object versioning , thereby reducing the representational redundancy of classical SFS. IncSFS dynamically collapses SCCs into representative nodes during incremental solving. The work proposed by Zhang *et al.* [46] also transforms value-flow information into a constraint graph and improves the efficiency of flow-sensitive pointer analysis in a whole-program setting. Independently and concurrently with their work, our work performs variable renaming in the incremental setting so that points-to deltas can be propagated safely after code changes.

7 Conclusion

We propose the first incremental fully sparse flow-sensitive pointer analysis algorithm by transforming the value-flow graph into a constraint graph and performing SCC detection on that graph. Furthermore, our method simultaneously propagates both increased and decreased points-to sets, improving analysis efficiency. Under the object-acyclic condition: points-to relation do not formulate a cycle, IncSFS is guaranteed to terminate and compute the least fixed point. Experiments on six large real-world programs show substantial improvements over both full analysis and the reset–recompute incremental algorithm. In addition, our method delivers higher efficiency than the state-of-the-art incremental pointer analysis algorithm.

8 Data Availability Statement

The artifacts and supplementary material are available at https://doi.org/10.5281/zenodo.21771594 [18].

Acknowledgments

This research was supported by the National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Program (Nos. 62172429 and 62032024).

References

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

[2] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 222–235. doi:10.1109/CGO51591.2021.9370334

[3] R. Barua, W. Lee, S. Arnarasinghe, and A. Agarwal. 2001. Compiler support for scalable and efficient memory systems. *IEEE Trans. Comput.* 50, 11 (2001), 1234–1247. doi:10.1109/12.966497

[4] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. 2003. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/781498.781502

[5] Keith D. Cooper and Ken Kennedy. 1984. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 247–258. doi:10.1145/502874.502898

[6] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Fluid updates: beyond strong vs. weak updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems* (Paphos, Cyprus) *(ESOP'10)*. Springer-Verlag, Berlin, Heidelberg, 246–266. doi:10.1007/978-3-642-11957-6_14

[7] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 106–117. doi:10.1145/277650.277670

[8] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. doi:10.1145/1348250.1348255

[9] Yu Gao, Dong Wang, Wensheng Dou, Wenhan Feng, Yu Liang, Yuan Feng, and Jun Wei. 2025. Model Checking Guided Incremental Testing for Distributed Systems. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA014 (June 2025), 23 pages. doi:10.1145/3728883

[10] Rakesh Ghiya and Laurie J. Hendren. 1998. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 121–133. doi:10.1145/268946.268957

[11] Vida Ghodssi. 1983. *Incremental analysis of programs.* University of Central Florida.

[12] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '93)*. Association for Computing Machinery, New York, NY, USA, 157–166. doi:10.1145/170035.170066

[13] Samuel Z. Guyer and Calvin Lin. 2003. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 214–236. doi:10.1007/3-540-44898-5_12

[14] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. *SIGPLAN Not.* 44, 1 (Jan. 2009), 226–238. doi:10.1145/1594834.1480911

[15] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 289–298. doi:10.1109/CGO.2011.5764696

[16] Michael Hind and Anthony Pioli. 2000. Which pointer analysis should I use?. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Portland, Oregon, USA) *(ISSTA '00)*. Association for Computing Machinery, New York, NY, USA, 113–123. doi:10.1145/347324.348916

[17] Jakob Krainz and Michael Philippsen. 2017. Diff Graphs for a fast Incremental Pointer Analysis. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Barcelona, Spain) *(ICOOOLPS'17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. doi:10.1145/3098572.3098578

[18] Liu KunLin, Chen Zhenbang, Zu Piyi, Du Yide, and Wang Ji. 2026. IncSFS: Incremental Full-Sparse Flow-Sensitive Pointer Analysis For C/C++(Artifact). doi:10.5281/zenodo.21771594

[19] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 278–289. doi:10.1145/1250734.1250766

[20] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. *SIGPLAN Not.* 46, 1 (Jan. 2011), 3–16. doi:10.1145/1925844.1926389

[21] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 343–353. doi:10.1145/2025113.2025160

[22] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:27. doi:10.4230/LIPIcs.ECOOP.2016.15

[23] Bozhen Liu and Jeff Huang. 2022. SHARP: fast incremental context-sensitive pointer analysis for Java. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 88 (April 2022), 28 pages. doi:10.1145/3527332

[24] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 6 (March 2019), 31 pages. doi:10.1145/3293606

[25] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An incremental points-to analysis with CFL-reachability. In *International Conference on Compiler Construction*. Springer, 61–81.

[26] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. Checker: a soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 1007–1024. doi:10.5555/3241189.3241268

[27] Aman Nougrahiya and V. Krishna Nandivada. 2025. IncIDFA: An Efficient and Generic Algorithm for Incremental Iterative Dataflow Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 102 (April 2025), 32 pages. doi:10.1145/3720436

[28] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.* 47, 6 (June 2012), 229–238. doi:10.1145/2345156.2254092

[29] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/1069774.1069785

[30] Alexandru Salcianu and Martin Rinard. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming* (Snowbird, Utah, USA) *(PPoPP '01)*. Association for Computing Machinery, New York, NY, USA, 12–23. doi:10.1145/379539.379553

[31] Marc Shapiro and Susan Horwitz. 1997. The Effects of the Precision of Pointer Analysis. In *Proceedings of the 4th International Symposium on Static Analysis (SAS '97)*. Springer-Verlag, Berlin, Heidelberg, 16–34. doi:10.1007/BFb0032731

[32] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 228–238. doi:10.1109/ISSRE.2019.00031

[33] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 112–122. doi:10.1145/1250734.1250748

[34] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021. Demanded abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 282–295. doi:10.1145/3453483.3454044

[35] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2024. Interactive Abstract Interpretation with Demanded Summarization. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 4 (March 2024), 40 pages. doi:10.1145/3648441

[36] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 160–170. doi:10.1145/2854038.2854043

[37] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[38] Zewen Sun, Yujin Zhang, Duanchen Xu, Yiyu Zhang, Yun Qi, Yueyang Wang, Yi Li, Zhaokang Wang, Yue Li, Xuandong Li, et al. 2024. Scaling Inter-procedural Dataflow Analysis on the Cloud. *arXiv preprint arXiv:2412.12579* (2024).

[39] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (Oct. 2018), 29 pages. doi:10.1145/3276509

[40] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3453483.3454026

[41] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. 2006. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the 15th International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. Springer-Verlag, Berlin, Heidelberg, 17–31. doi:10.1007/11688839_3

[42] Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. 2023. Result Invalidation for Incremental Modular Analyses. In *Verification, Model Checking, and Abstract Interpretation: 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16–17, 2023, Proceedings* (Boston, MA, USA). Springer-Verlag, Berlin, Heidelberg, 296–319. doi:10.1007/978-3-031-24950-1_14

[43] Jiayi Wang, Yu Wang, Ke Wang, and Linzhang Wang. 2025. SILVA: A Scalable Incremental Layered Sparse Value-Flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 34, 8, Article 229 (Oct. 2025), 40 pages. doi:10.1145/3725214

[44] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 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* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 218–229. doi:10.1145/1772954.1772985

[45] J.-S. Yur, B.G. Ryder, and W.A. Landi. 1999. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 442–451. doi:10.1145/302405.302676

[46] Jiahao Zhang, Xiao Cheng, and Yuxiang Lei. 2025. Flow Sensitivity without Control Flow Graph: An Efficient Andersen-Style Flow-Sensitive Pointer Analysis. *arXiv preprint arXiv:2508.01974* (2025).