



Taming and Dissecting Recursions through Interprocedural Weak Topological Ordering

Jiawei Yang*  

The Hong Kong Polytechnic University, Hong Kong

Xiao Cheng*  

University of New South Wales, Australia

Bor-Yuh Evan Chang[†]  

University of Colorado Boulder & Amazon, USA

Xiapu Luo  

The Hong Kong Polytechnic University, Hong Kong

Yulei Sui  

University of New South Wales, Australia

Abstract

Abstract interpretation provides a foundational framework for approximating program semantics by interpreting code through abstract domains using semantic functions over ordered sets along a program's control flow graph (CFG). To facilitate fixpoint computation in abstract interpretation, weak topological ordering (WTO) is an effective strategy for handling loops, as it identifies strategic control points in the CFG where widening and narrowing operations should be applied. However, existing abstract interpreters still face challenges when extending WTO computation in the presence of recursive programs. Computing a precise whole-program WTO requires full context-sensitive analysis which is not scalable for large programs, while context-insensitive analysis introduces spurious cycles that compromise precision. Current approaches either ignore recursion (resulting in unsoundness) or rely on conservative approximations, sacrificing precision by adopting the greatest elements of abstract domains and applying widening at function boundaries without subsequent narrowing refinements. These can lead to undesired results for downstream tasks, such as bug detection.

To address the above limitations, we present RECTOPO, a new technique to boost the efficiency of precise abstract interpretation in the presence of recursive programs through interprocedural weak topological ordering (IWTO). Rather than pursuing an expensive whole-program WTO analysis, RECTOPO employs an on-demand approach that strategically decomposes programs at recursion boundaries and constructs targeted IWTOs for each recursive component. RECTOPO dissects and analyzes (nested) recursions through interleaved widening and narrowing operations. This approach enables precise control over interpretation ordering within recursive structures while eliminating spurious recursions through systematic correlation of control flow and call graphs.

We implemented RECTOPO and evaluated its effectiveness using an assertion-based checking client focused on buffer overflow detection, comparing it against three popular open-source abstract interpreters (IKOS, Clam, CSA). The experiments on 8312 programs from the NIST dataset demonstrate that, on average, RECTOPO is 31.99% more precise and achieves a 17.49% higher recall rate compared to three other tools. Moreover, RECTOPO exhibits an average precision improvement of 46.51% and a higher recall rate of 32.98% compared to our baselines across ten large open-source projects. Further ablation studies reveal that IWTO reduces spurious widening operations compared to whole-program WTO, resulting in a 12.83% reduction in analysis time.

2012 ACM Subject Classification Theory of computation → Program analysis

*These authors contributed equally to this work.

[†]Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at the University of Colorado Boulder and is not associated with Amazon.



© Jane Open Access and Joan R. Public;

licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).

Editors: Jonathan Aldrich and Alexandra Silva; Article No. 22; pp. 22:1–22:30

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

44 **Keywords and phrases** Abstract interpretation, recursion, weak topological ordering

45 **Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2025.22

46 **Supplementary Material** *Software (Source Code)*: <https://github.com/JoelYYoung/RecTopo>

47 **Funding** This research is supported by Australian Research Council Grants DP250101396 and
48 FT220100391.

49 **1 Introduction**

50 Abstract interpretation [24, 22, 25] offers a framework for approximating program se-
51 mantics by analyzing code through abstract domains and applying semantic functions
52 over ordered sets along the program’s control flow graph (CFG). The analysis converges
53 to a conservative solution at a post-fixpoint, which is an over-approximation of the least
54 fixpoint of the program’s semantics. This least fixpoint is defined by a system of equations
55 $a_1 = \Phi_1(a_1, a_2, \dots, a_n), \dots, a_n = \Phi_n(a_1, a_2, \dots, a_n)$, where each a_i denotes the abstract
56 states at control point i , and Φ_i represents the corresponding semantic function of a program.

57 For programs with acyclic control flows, fixpoint computation typically converges after a
58 few iterations over the CFG. However, the presence of cycles introduces cyclic dependencies
59 between program states, complicating the analysis when working with abstract domains
60 characterized by unbounded or very deep lattice heights [27, 62]. In such domains, such as
61 the standard interval domain [17], the iterative computation may require a prohibitively
62 large number of steps to reach a post-fixpoint. Even worse, termination is not guaranteed,
63 as a post-fixpoint may not be reached within a finite number of steps. To facilitate fixpoint
64 computation in these domains, widening techniques [22, 24, 13] compute a safe (upper)
65 approximation of the least fixpoint (a post-fixpoint, hereafter referred to as a fixpoint for
66 simplicity). However, widening operations must be judiciously applied only at cycle points,
67 as spurious widening at acyclic locations introduces precision loss due to over-approximation,
68 which cannot be recovered through subsequent narrowing [11]. Bourdoncle’s algorithm [13]
69 provides an effective solution through weak topological ordering (WTO) for intraprocedural
70 analysis, which efficiently identifies program loops and their nested structures while requiring
71 only one widening point per loop. The WTO framework further enables sophisticated
72 interleaved widening and narrowing strategies [10], where narrowing compensates precision
73 that may have been lost during widening by incorporating invariants such as loop bounds and
74 branching constraints. These advantages have made WTO a standard method for handling
75 intraprocedural loops in many current abstract interpreters [15, 26, 19, 18].

76 While state-of-the-art abstract interpreters effectively leverage Bourdoncle’s algorithm
77 for loop analysis, they face difficulties when determining WTO for recursive function calls.
78 Several approaches have been proposed for taming recursions and interprocedural abstract
79 interpretation. For non-recursive functions, function inlining enables precise interprocedural
80 analysis. However, this approach becomes infeasible for recursive functions due to potentially
81 infinite inlining sequences in unbounded recursion. To address this, some tools adopt strong
82 assumptions, employing either under- or over-approximation approaches. For example,
83 Astrée [37] completely excludes the analysis of unbounded recursion, while IKOS [15, 40]
84 conservatively assigns top values (\top) to variables in recursive contexts, effectively bypassing
85 the analysis of recursive functions. To improve precision, tools such as Clam [34, 19] and
86 CSA [18] analyze recursive function bodies but introduce many spurious widening operations
87 at function boundaries. Their effectiveness remains limited, as their WTO computations are
88 confined to individual function scopes, preventing the creation of an effective interprocedural

order for efficient fixpoint computation and subsequent narrowing to enhance precision. Consequently, both under-approximation and over-approximation approaches can produce undesirable outcomes for downstream tasks, such as bug detection.

Addressing these limitations necessitates an approach that incorporates both the analysis of recursive function bodies and guarantees termination properties. This requires an interprocedural weak topological ordering (IWTO) that establishes an efficient computation order across function boundaries, enabling minimal widening points and allowing narrowing strategies for precise handling of recursions. The primary challenge lies in accurately identifying the IWTO within the control flow graph, where multiple call sites of the same function can generate spurious recursive paths. These paths arise when call sites connect to prior invocations of the same function, potentially introducing unnecessary widening points and thereby compromising the precision of the analysis.

Figure 1 illustrates this challenge in the context of the McCarthy 91 function [43]. The function `recur` ($\ell_1 - \ell_7$) is a recursive function that takes an integer `p` as input and recursively invokes itself at ℓ_5 to increase `p`'s value until `p` is greater than 100. The main function invokes `recur` twice at ℓ_9 and ℓ_{10} respectively. In the control flow graph, each function invocation at location ℓ_n is decomposed into two distinct nodes: a call site ℓ_n^c and a corresponding return site ℓ_n^r (e.g., the call at ℓ_9 in the source program manifests as ℓ_9^c and ℓ_9^r in the control flow graph).

The control flow edges are established such that each call site connects to its callee's entry point (e.g., $\ell_5^c \rightarrow \ell_1$, $\ell_9^c \rightarrow \ell_1$, and $\ell_{10}^c \rightarrow \ell_1$), while return edges link the callee's exit point to the corresponding return sites (e.g., $\ell_7 \rightarrow \ell_{10}^r$, $\ell_7 \rightarrow \ell_9^r$, and $\ell_7 \rightarrow \ell_5^r$). Note that, a spurious recursive path highlighted in red in the figure ($\ell_9^c \rightarrow \ell_{10}^c \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_6 \rightarrow \ell_7 \rightarrow \ell_9^r$) arises due to the consecutive call to the function `recur` at ℓ_9 and ℓ_{10} , which may lead to a redundant widening operation at the spurious recursion and compromises analysis precision. While a fully context-sensitive analysis could theoretically eliminate such spurious paths through precise call-return matching, the exponential complexity of such an approach renders it impractical for real-world program analysis.

We present RECTOPO, a new interprocedural fixpoint computation technique that computes interprocedural weak topological ordering (IWTO) in the presence of recursive programs, enabling recursions to be handled as efficiently and precisely as loops. Our approach systematically dissects programs into non-recursive and recursive components, avoiding the complexity of whole-program WTO construction. The framework first leverages Tarjan's strongly connected component algorithm [61] on the call graph to identify recursive function calling chains. Subsequently, on the CFG, it coalesces call sites of functions within the same recursive chain and assigns each recursion a unified IWTO. This principled decomposition enables the seamless integration of an efficient worklist algorithm with interleaved widening and narrowing operations for precise recursion analysis. Our approach also eliminates spurious interprocedural cycle, thus preventing spurious widening operations that would otherwise arise from distinct call sites of the same function.

Figure 2 illustrates the three-phase architecture of RECTOPO, demonstrating how our framework systematically addresses the challenges of interprocedural analysis in the presence

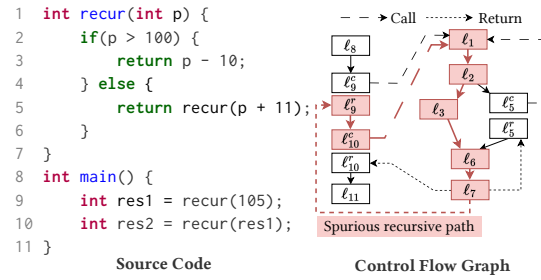
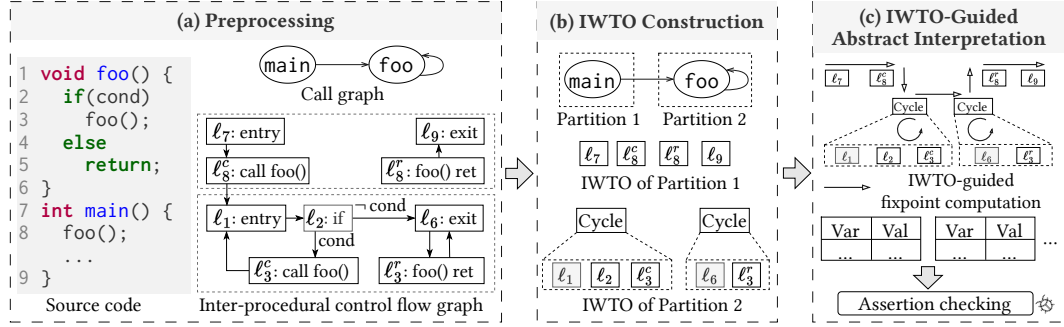


Figure 1 Spurious recursive path when analyzing McCarthy 91 function [43] (a library of mathematical theory of computation).



■ **Figure 2** An overview of RECTOPO framework.

of recursions:

(a) Preprocessing: The framework begins by transforming the input program into LLVM’s intermediate representation (LLVM-IR) [41]. This standardized representation provides a unified foundation for constructing both the interprocedural control-flow graph (ICFG) and the corresponding call graph.

(b) IWTO Construction: At the core of our approach lies a novel two-step IWTO construction process. First, we employ Tarjan’s algorithm [61] on the call graph to identify and partition functions based on their recursive relationships. This partitioning is crucial as it enables us to isolate and group functions within the same recursive calling chain. Then, guided by these function partitions, we construct their corresponding IWTOs on the ICFG through iterative applications of Tarjan’s algorithm. This strategic decomposition yields a precise IWTO for each recursion, providing an effective foundation for subsequent analysis phases while avoiding the complexity of whole-program WTO construction.

(c) IWTO-Guided Abstract Interpretation: The framework leverages the constructed IWTOs to perform precise abstract interpretation. By following the computed order, our analysis ensures effective convergence through strategic application of widening and narrowing operations at carefully selected points. This principled approach yields highly precise abstract states that serve as a foundation for various client analyses, such as buffer overflow detection, while maintaining analysis efficiency.

In summary, this paper makes the following major contributions:

- We introduce RECTOPO, a new fixpoint computation approach for precise interprocedural abstract interpretation with enhanced efficiency. Our approach manages recursions precisely by leveraging interprocedural weak topological ordering (IWTO) and an interleaved strategy of widening and narrowing operations.
- We introduce a scalable algorithm for constructing an IWTO for each recursion, designed to circumvent the high-cost construction of a whole-program weak topological ordering. This is accomplished by applying Tarjan’s strongly connected component algorithm [61] on the call graph, which in turn guides the construction of IWTOs on the control flow graph.
- We comprehensively evaluate RECTOPO’s performance using 8312 programs from the NIST dataset and 10 real-world open-source projects. Experimental results show that RECTOPO outperforms our baselines by achieving 31.99% higher precision and a 17.49% increase in recall rate on average. Furthermore, RECTOPO shows an average precision improvement of 46.51% and a higher recall rate of 32.98% compared to baselines tools on ten open-source projects.

■ **Table 1** Analysis domains and LLVM-like statement set.

ℓ	$\in \mathcal{L}$	Statements
c, fld	$\in \mathcal{C}$	Constants
p, q, idx	$\in \mathcal{S}$	Stack virtual registers
g	$\in \mathcal{G}$	Global variables
f	$\in \mathcal{F} \subseteq \mathcal{G}$	Program functions
p, q, idx, g	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	Top-level variables/pointers
o	$\in \mathcal{O}$	Abstract objects
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	Program variables
<hr/>		
ℓ	$::= p = c \mid p = \&o$	CONST/ADDR
	$\mid p = \&(q \rightarrow fld) \mid p = \&q[idx]$	GEP
	$\mid p = *q \mid *p = q$	LOAD/STORE
	$\mid p = q$	COPY
	$\mid p = \phi(p_1, p_2, \dots)$	PHI
	$\mid r = p \odot q$	BINARY
	$\mid p = \neg q$	UNARY
	$\mid r = f(p_1, \dots, p_n)$	CALL
	$\mid \text{ret}_f q$	RETURN
<hr/>		
$\odot \in \{+, -, *, /, \%, <<, >>, <, >, \&, \&\&, <=, >=, \equiv, \sim, , \wedge\}$		

172 2 Background

173 This section provides the foundational knowledge of the language on which our abstract
 174 interpretation is based (see §2.1), as well as discussions on abstract interpretation (see §2.2)
 175 and weak topological ordering (see §2.3).

176 2.1 Language

177 Table 1 presents the analysis domains and LLVM-like statement set used in abstract inter-
 178 pretation. The set of program variables \mathcal{V} is split into two disjoint subsets: \mathcal{O} , compris-
 179 ing *address-taken variables*, and \mathcal{P} , containing all *top-level variables*. Top-level variables
 180 $p, q, idx, g \in \mathcal{P}$ adhere to static single assignment (SSA) form and are subject to exactly one
 181 definition. Address-taken objects $o \in \mathcal{O}$ can only be accessed through pointer dereferencing
 182 operations at LOAD/STORE instructions. Constant values are initially bound to top-level
 183 variables through CONS instructions ($p = c$) that assign constants c . ADDR statements,
 184 denoted as $p = \&o$, allocate address-taken objects o , which may represent stack variables,
 185 global variables, or abstract heap objects. GEP enables structured access to composite objects:
 186 struct fields are accessed via constant offsets fld , while array elements use variable offsets
 187 idx . For field-sensitive analysis, RECTOPO employs a field-index-based strategy [12, 52],
 188 distinguishing struct fields through unique indices. COPY denotes a simple assignment of
 189 top-level variables/pointers. The PHI instruction, fundamental to SSA form, consolidates
 190 variable values at control flow merge points. BINARY and UNARY instructions represent
 191 arithmetic and bitwise operations, respectively. CALL and RETURN represents parameter
 192 passing and return value handling in function invocations.

2.2 Abstract Interpretation

Abstract interpretation offers a formalized framework for computing program semantics at each control point, represented as an abstract state specific to that point. The collection of all abstract states constitutes an abstract trace, denoted as σ (see Definition 2). The analysis iteratively updates σ by applying a semantic function—assumed to be monotonic to simplify the exposition—until reaching a fixpoint. This fixpoint yields a sound over-approximation of the program’s behaviors.

► **Definition 1** (Concrete and Abstract Domains). Let $\mathbb{C} = 2^{V \rightarrow S}$ be the concrete domain, where S represents all possible runtime values. The abstract domain \mathbb{A} forms a lattice $\langle \mathbb{A}, \sqsubseteq, \sqcap, \sqcup, \perp, \top \rangle$ with partial order \sqsubseteq . A Galois connection $\mathbb{C} \xrightleftharpoons[\gamma]{\alpha} \mathbb{A}$ formalizes their relationship through abstraction function $\alpha : \mathbb{C} \rightarrow \mathbb{A}$ and concretization function $\gamma : \mathbb{A} \rightarrow \mathbb{C}$.

► **Definition 2** (Abstract Trace $\sigma \in \mathbb{L} \rightarrow \mathbb{A}$). An abstract trace maps control points to abstract states, where σ_L denotes the abstract state at control point $L \in \mathbb{L}$. Function $\sigma_L(x)$ yields variable x ’s abstract value at L . For a program statement ℓ , $\sigma_{\bar{\ell}}$ and $\sigma_{\underline{\ell}}$ represent the pre- and post-abstract states, respectively.

Widening. The widening technique [24, 22] is introduced to ensure fixpoint termination and accelerate convergence by approximating infinite loop behavior. It involves choosing a subset of control points $\mathcal{L}' \subseteq \mathcal{L}$ as widening points. For each widening point $\ell_i \in \mathcal{L}'$, we use equation $\sigma_{\ell_i} = \sigma_{\ell_i} \nabla \Phi_i(\sigma_{\ell_1}, \sigma_{\ell_2}, \dots, \sigma_{\ell_n})$ instead of $\sigma_{\ell_i} = \Phi_i(\sigma_{\ell_1}, \sigma_{\ell_2}, \dots, \sigma_{\ell_n})$ for fixpoint computation. In this equation, ∇ is a widening operator, which is formally defined on a poset $(\mathbb{A}, \sqsubseteq)$ that satisfies the following properties: (1) $\forall a, a' \in \mathbb{A}, a \sqcup a' \sqsubseteq a \nabla a'$ and (2) for an increasing chain $a_1 \sqsubseteq a_2 \sqsubseteq \dots \sqsubseteq a_m$, the increasing chain $a'_1 \sqsubseteq a'_2 \sqsubseteq \dots \sqsubseteq a'_m$ where $a'_1 = a_1$ and $a'_m = a'_{m-1} \nabla a_m$ eventually stabilizes. For example, the widening function on the integer interval domain $\mathcal{I} = \{[l, u] \mid l, u \in \mathbb{Z}\}$ is defined as: $[l_0, u_0] \nabla [l_1, u_1] = [\text{if } l_1 < l_0 \text{ then } -\infty \text{ else } l_0, \text{if } u_0 < u_1 \text{ then } +\infty \text{ else } u_0]$.

Narrowing. After the widening iteration converges to a fixpoint, a subsequent narrowing iteration, as described in [15, 24, 11, 39], aims to enhance the post solution through a downward fixpoint iteration. The narrowing operator Δ satisfies the following properties: (1) $\forall a, a' \in \mathbb{A}, a' \sqsubseteq a \implies a \Delta a' \sqsubseteq a$ and (2) for a decreasing chain $a_1 \sqsupseteq a_2 \sqsupseteq \dots \sqsupseteq a_m$, the decreasing chain $a'_1 \sqsupseteq a'_2 \sqsupseteq \dots \sqsupseteq a'_m$ where $a'_1 = a_1$ and $a'_m = a'_{m-1} \Delta a_m$ eventually stabilizes. For example, the narrowing operator on the integer interval domain \mathcal{I} is defined as: $[l_0, u_0] \Delta [l_1, u_1] = [\text{if } l_0 = -\infty \text{ then } l_1 \text{ else } l_0, \text{if } u_0 = +\infty \text{ then } u_1 \text{ else } u_0]$.

2.3 Weak Topological Ordering (WTO)

WTO provides an arbitrary update of abstract states rather than strictly following the order of control points. This arbitrary update (also called *chaotic iteration* [20]) is proven not to affect the precision of fixpoint computation while having the potential to accelerate the convergence to the fixpoint [20, 21]. We first give some basic concepts of WTO and then delve into how WTO can be leveraged to enhance the precision of widening and accelerate the *chaotic iteration algorithm*.

► **Definition 3** (Hierarchical Ordering). A hierarchical ordering of a set of elements V is a well-parenthesized permutation of V that does not contain consecutive “(”s.

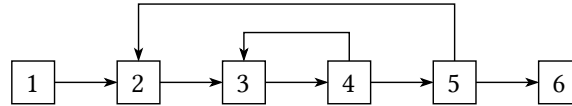
A hierarchical ordering of V induces a total order \preceq , a binary relation over the elements of V , where for any $v, w \in V$, either $v \preceq w$ or $w \preceq v$ holds, establishing a linear sequence.

A *hierarchical component* is formed by a matching pair of “(” and “)”, along with the sub-components or elements in V between them in the specified order. The hierarchical order could also be viewed as a tree, with leaf nodes representing the elements of V , and sub-trees (with non-leaf roots) representing hierarchical components. The first element of a hierarchical component is denoted as *component head*, which is highlighted using double underlining as $\underline{\underline{v}}$. The set of heads of the components containing the element v is denoted by $\omega(v)$. The depth of an element $v \in V$ is defined as $\delta(v) = |\omega(v)|$, which represents the depth of the leaf node corresponding to v in the tree.

► **Example 4.** “($\underline{\underline{1}}$ ($\underline{\underline{2}}$ 3))” is a valid hierarchical ordering of elements $\{1, 2, 3\}$. Both “($\underline{\underline{1}}$ ($\underline{\underline{2}}$ 3))” and “($\underline{\underline{2}}$ 3)” are hierarchical components with 1 and 2 being their respective component heads. The orders like “(1 ((2 3))” or “((1 2) 3)” are invalid hierarchical orders. The former is not *well-parenthesized*, and the latter contains consecutive “(”s. Components have a hierarchical (nested) structure, meaning that one component can be a sub-component of another. For instance, “($\underline{\underline{2}}$ 3)” is a sub-component of “($\underline{\underline{1}}$ ($\underline{\underline{2}}$ 3))”, thus $\omega(3) = \{1, 2\}$ and $\delta(3) = 2$.

► **Definition 5** (Weak Topological Ordering). A *weak topological ordering (WTO)* of a directed graph $G = (V, E)$ is a hierarchical ordering of control points V such that for each edge $u \rightarrow v \in E$, $u \prec v \wedge v \notin \omega(u)$ or $v \preceq u \wedge v \in \omega(u)$. If an edge $u \rightarrow v$ satisfies $v \preceq u \wedge v \in \omega(u)$, $u \rightarrow v$ is called a *back edge*.

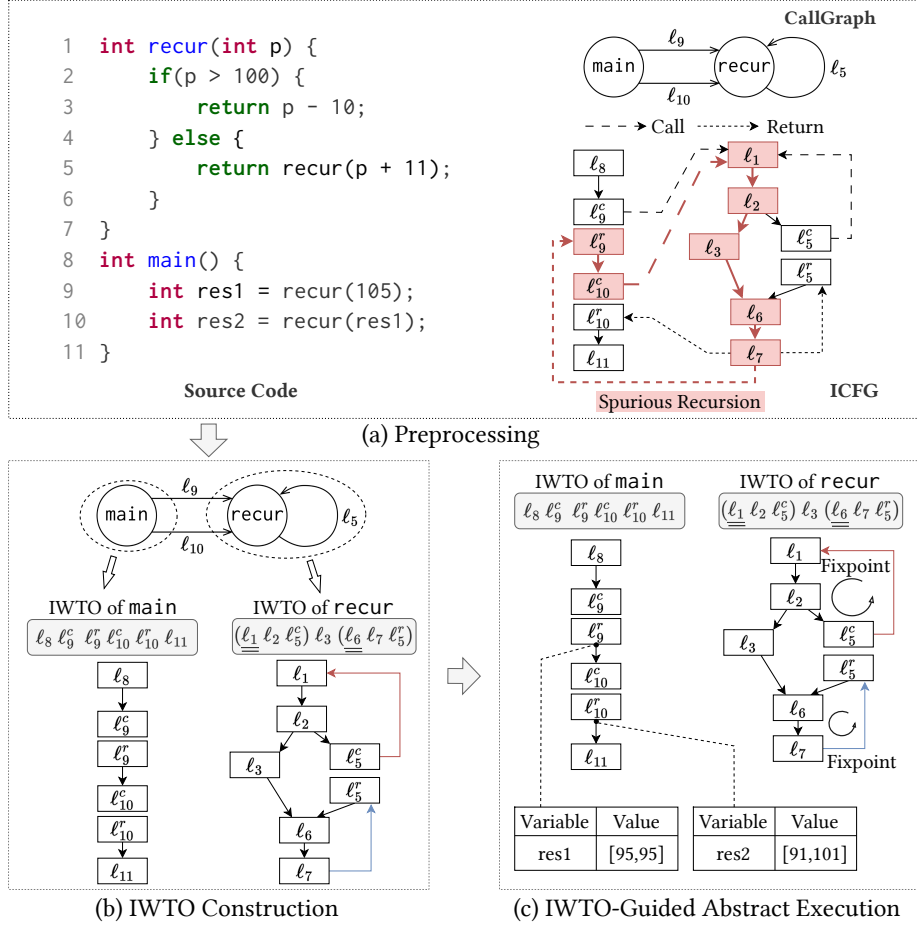
► **Example 6.** One possible WTO of the intraprocedural control flow graph in Figure 3 is “1 ($\underline{\underline{2}}$ ($\underline{\underline{3}}$ 4) 5) 6”. 2 and 3 are the WTO heads and $5 \rightarrow 2$ and $4 \rightarrow 3$ are back edges. It is clear that WTO precisely captures the nested loop structures within the graph. The inside loop is captured by “($\underline{\underline{3}}$ 4)” while the outside loop is “($\underline{\underline{2}}$ ($\underline{\underline{3}}$ 4) 5)”.



■ **Figure 3** An example of control flow graph.

WTO-Guided Chaotic Iteration. Given the explicit WTO of the control flow points, the chaotic iteration recursively computes a fixed point of the sub-components of each component in WTO every time the component reaches a fixed point [13]. WTO also provides an admissible set of widening points (the heads of WTO components) such that every loop in the program is cut by at least one widening point.

WTO Construction. WTO is constructed based on Bourdoncle’s algorithm [32], which utilizes Tarjan’s algorithm [61] to identify all strongly connected components (SCCs) in a directed graph. The algorithm repeatedly applies Tarjan’s algorithm on the graph to break each found SCC into smaller SCCs until no further subdivision is possible. Each time an SCC is divided, smaller SCCs are ordered within the larger one in the WTO. The hierarchical structure of these components reflects the partial ordering of the WTO. Current abstract interpretation frameworks [15, 19, 18] compute WTO at the function level, applying the analysis intraprocedurally. However, the interprocedural fixpoint computation order remains implicit, as it is not guided by an interprocedural WTO structure.



■ **Figure 4** A motivating example by revisiting the example in Figure 1. It provides a detailed illustration of the phases of our framework, as shown in Figure 2.

3 Motivating Example

Figure 4 illustrates how RECTOPO precisely determines the interprocedural fixpoint computation order and widening points by revisiting the example in Figure 1. We use the integer interval domain [17] to reason about its behaviors. The state-of-the-art analyzers, IKOS [15], Clam [19, 34] and CSA [18], all yield an imprecise top value $[-\infty, \infty]$ for `recur`'s return variable `res1` at ℓ_9 and `res2` at ℓ_{10} . In comparison, RECTOPO derives a more precise result of [95, 95] for `res1` and [91, 101] for `res2`, demonstrating its precise interprocedural fixpoint computation order and widening strategies.

(a) Preprocessing. As shown in Figure 4(a), we first generate the call graph and interprocedural control flow graph (ICFG) of the analyzed program. In the call graph, we identify the function `recur` as a recursive function, and `main` as the caller. In the recursive function `recur`, we observe two distinct recursive calling paths: one from ℓ_1 to ℓ_5^c and another from ℓ_6 to ℓ_5^r . Note that, the cyclic path highlighted in red in the figure ($\ell_9^r \rightarrow \ell_{10}^c \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_6 \rightarrow \ell_7 \rightarrow \ell_9^r$) is a spurious recursive path arising due to the consecutive call to the function `recur` at ℓ_9 and ℓ_{10} .

(b) IWTO Construction. Initially, we apply Tarjan's algorithm to the call graph, identifying two distinct function partitions: `main` and `recur`. Leveraging these partitions, we divide the original ICFG into two sub-ICFGs: one for `main` and another for `recur`. This

division eliminates interprocedural edges between `main` and `recur`, as they are not part of the same recursive calling chain, according to the call graph's structure. Consequently, this approach removes the spurious recursion previously depicted in Figure 4(a) and prevents the redundant widening of this spurious recursion, thereby maintaining the precision as inlining-based methods for non-recursive functions. Furthermore, the true recursion within `recur` is comprehensively analyzed, resulting in an interprocedural weak topological ordering (IWTO) of " $(\underline{\ell}_1 \ \ell_2 \ \ell_5^c) \ \ell_3 \ (\underline{\ell}_6 \ \ell_7 \ \ell_5^r)$ " (where $\underline{\ell}_1$ and $\underline{\ell}_6$ are WTO heads). This analysis contrasts with approaches like IKOS [15], which typically omit the function body in recursive analyses, or Clam [19, 34] and CSA [18] which only perform widening on ℓ_1 and ℓ_7 without narrowing.

(c) IWTO-Guided Abstract Interpretation. The abstract interpretation illustrated in Figure 4(c) follows the order outlined in the IWTO constructed for the program. The execution begins at the `main` function and progresses through the call sites ℓ_9^c and ℓ_{10}^c . An interleaved widening and narrowing approach is employed to precisely approximate the behaviors of the two IWTO cycles within `recur`. Specifically, at the call site ℓ_9^c , the formal parameter `p` is initialized with the interval $[105, 105]$. As execution enters the first cycle $(\underline{\ell}_1 \ \ell_2 \ \ell_5^c)$, a fixpoint is quickly reached with `p` remaining at $[105, 105]$, given that the path condition for the branch from ℓ_2 to ℓ_5^c is infeasible. Subsequently, at ℓ_3 , the return value is calculated as $[95, 95]$. This value is then carried over to the second IWTO cycle $(\underline{\ell}_6 \ \ell_7 \ \ell_5^r)$, where a fixpoint is again promptly achieved, resulting in a final return value of $[95, 95]$ for `res1`, much more precise compared to $[-\infty, \infty]$ for existing state-of-the-art tools [15, 19, 18]. Following this, the second invocation at ℓ_{10}^c uses the value of `res1`, $[95, 95]$, as the input for `p`, and re-engages with the IWTO of `recur`. In the first cycle, the initial widening operation expands `p` to $[95, \infty]$, followed by narrowing that refines the value to $[95, 111]$, which is subsequently propagated to ℓ_3 as $[101, 111]$. Consequently, upon returning to ℓ_{10}^r , the value of `res2` is determined to be $[91, 101]$, demonstrating consistently greater precision than IKOS, Clam and CSA.

4 RecTopo Approach

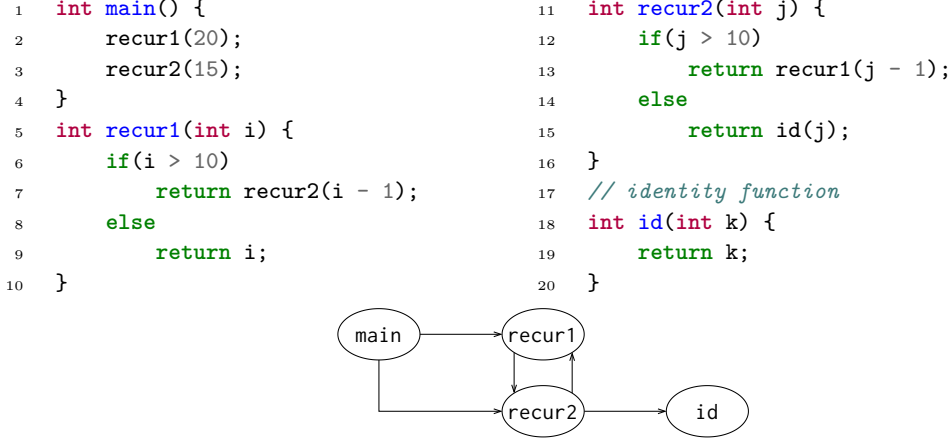
In this section, we introduce our approach for constructing interprocedural weak topological ordering in §4.1, followed by the algorithm for IWTO-guided abstract interpretation in §4.2. Additionally, we provide the proof of termination for our algorithm in §4.3.

4.1 IWTO Construction

The construction of IWTO consists of two stages. In the first stage (see §4.1.1), we perform program partitioning to divide the program into several function partitions (Definition 7). The second stage (see §4.1.2) involves constructing the IWTO on the ICFG for each resulting function partition from the first stage.

4.1.1 Program Partitioning

This stage aims to decompose the input whole program into several function partitions (see Definition 7) and identify all function partition entries (see Definition 8). The process of function partitioning segments the whole program analysis into manageable parts while ensuring that the functions within each recursive call are analyzed collectively and precisely. The identification of function partition entries is critical to guarantee that each IWTO is accurately constructed starting from the entry point as dictated by the program's call sites.



■ **Figure 5** A recursion example and its call graph.

332 ► **Definition 7** (Function Partition). *A function partition, denoted as FPar , consists of a set*
333 *of functions that are directly or indirectly reachable from each other on the call graph.*

334 We apply Tarjan’s algorithm [61] to the pre-built call graph of the analyzed program,
335 identifying all strongly connected components (SCCs). Each SCC on the call graph corres-
336 ponds to a distinct function partition. Non-recursive functions, being single-node SCCs, each
337 forms an individual partition. For recursive functions, those within the same calling chain
338 are aggregated into a single partition. Note that if a recursive function calls a non-recursive
339 function, they are placed in separate partitions because the non-recursive function does not
340 call back, meaning that they do not meet the criteria of mutual reachability required for
341 placement in the same partition (Definition 7).

342 ► **Definition 8** (Function Partition Entry). *For a function partition FPar , a function $f \in \text{FPar}$*
343 *is considered a function partition entry if and only if there exists a function $f' \notin \text{FPar}$ such*
344 *that f' calls f . In other words, f is an entry point to the partition FPar if it is invoked by at*
345 *least one function outside of the partition.*

346 **Function Partition Entries Identification.** For each function f within a function
347 partition FPar , we examine the predecessors of f on the call graph. If any predecessor f' is
348 found not to belong to FPar , then f is designated as a partition entry for that partition.

349 ► **Example 9.** We illustrate the process of program partitioning with an example and its call
350 graph in Figure 5. This example includes a mutual recursion between the functions `recur1`
351 and `recur2`, invoked by `main` at locations ℓ_2 and ℓ_3 , respectively. Additionally, the function
352 `id` is called by `recur2` at location ℓ_{15} . Applying Tarjan’s algorithm to this call graph results
353 in three SCCs: $\{\text{main}\}$, $\{\text{recur1}, \text{recur2}\}$, and $\{\text{id}\}$. Each SCC corresponds to a distinct
354 function partition. For the function partition $\{\text{recur1}, \text{recur2}\}$, both functions are identified
355 as partition entries, as they are separately called by `main` at different locations.
356

357 4.1.2 IWTO Construction

358 For each function partition entry, we establish an interprocedural weak topological ordering
359 (IWTO). As shown in Algorithm 1, the main function `constructIMap` (Lines 1-5) processes
360 all function partitions of the analyzed program, alongside `EMap`, which maps each function

Algorithm 1 IWTO construction algorithm

```

Input:  $G(V, E)$ : ICFG of the program
          $E_{\text{Map}}$ : map from function partition to its partition entries
Output:  $I_{\text{Map}}$ : map from partition entry to IWTO
1 Function  $\text{constructIMap}(E_{\text{Map}}, G)$ :
2    $I_{\text{Map}} := \{\}$ ;
3   foreach  $(F_{\text{Par}} \rightarrow F_{\text{ParEns}}) \in E_{\text{Map}}$  do
4      $\text{constructIWTO}(F_{\text{Par}}, F_{\text{ParEns}}, G, I_{\text{Map}})$ ;
5   return  $I_{\text{Map}}$ ;
6 Function  $\text{constructIWTO}(F_{\text{Par}}, F_{\text{ParEns}}, G, I_{\text{Map}})$ :
7    $G_{F_{\text{Par}}}(V_{F_{\text{Par}}}, E_{F_{\text{Par}}}) := \text{ICFGPartition}(G, F_{\text{Par}})$ ;           // Step 1
8   foreach  $v \in V_{F_{\text{Par}}}$  do                                           // Step 2
9     if  $v$  is a non-recursive call site then
10       $E_{F_{\text{Par}}} := E_{F_{\text{Par}}} \cup (v, v.\text{returnSite}())$ ;
11   foreach  $\text{entry} \in F_{\text{ParEns}}$  do                                       // Step 3
12      $\text{entryICFGNode} := \text{Entry ICFG Node of entry}$ ;
13      $I_{\text{Map}}[\text{entry}] := \text{Bourdoncle}(G_{F_{\text{Par}}}, \text{entryICFGNode})$ ;

```

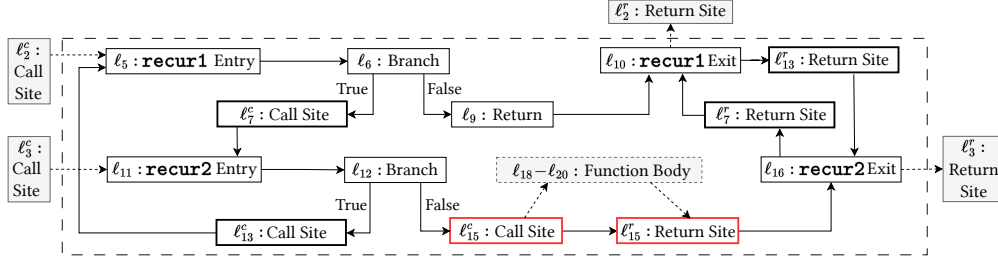
partition to its function partition entries and aims to produce I_{Map} , which maps each function partition entry to its respective IWTO. The subroutine `constructIWTO` defined at Lines 6-13 aims to construct IWTO for a specific function partition F_{Par} and its associated entries F_{ParEns} . `constructIWTO` first extracts the subgraph corresponding to F_{Par} on the ICFG, and then completes the extracted subgraph by connecting non-recursive calls and returns (Definition 10). Finally, based on the completed subgraph, it constructs an IWTO for each function partition entry and stores the results in I_{Map} . The detailed steps of `constructIWTO` are outlined as follows:

Step 1: ICFG Partitioning. Given a function partition F_{Par} , this step (Line 7) aims to extract the induced sub-ICFG corresponding to F_{Par} from the ICFG of the entire input program. The resulting sub-ICFG is composed exclusively of ICFG nodes that are contained within a specific function from F_{Par} .

► **Definition 10** (Recursive and Non-Recursive Call/Return Site). A call site $\ell^c \langle \text{caller}, \text{callee} \rangle$ in the ICFG is defined as a recursive call site iff. `caller` and `callee` are in the same function partition. Else, ℓ^c is classified as a non-recursive call site. Each ℓ^c is associated with a corresponding return site ℓ^r , which facilitates the return from the callee function.

► **Example 11.** Figure 6 shows the sub-ICFG of function partition $\{\text{recur1}, \text{recur2}\}$ for the code example in Figure 5. The call sites ℓ_7^c and ℓ_{13}^c are *recursive call sites* because their callee functions are recursive, and these calls occur within recursive functions (`recur1` and `recur2`). By comparison, the call sites ℓ_2^c and ℓ_3^c are considered *non-recursive* as they are located in the `main` function, which is not part of the recursion involving `recur1` or `recur2`. ℓ_{15}^c is also a *non-recursive call site* because its callee, `id`, is not a recursive function.

Step 2: Sub-ICFG Completion. This step (Lines 8-10) ensures that every node within the sub-ICFG extracted in Step 1 remains reachable from the entry of its corresponding function. During the ICFG partitioning in Step 1, all non-recursive return sites (Definition 10) become unreachable. This occurs because each non-recursive return site is initially linked to its



■ **Figure 6** The sub-ICFG of function partition **recur1** and **recur2** by revisiting the example in Figure 5.

corresponding call site via the callee function body; however, the callee function is excluded from the sub-ICFG because it does not belong to the current function partition. Therefore, we introduce an additional edge from each non-recursive call site directly to its respective return site.

Step 3: IWTO Construction. In this step (Lines 11-13), we employ Bourdoncle’s algorithm [32] to build the IWTO for the completed sub-ICFG derived from Step 2. For each partition entry, denoted as **entry**, Bourdoncle’s algorithm begins at the function entry node associated with the function partition entry, referred to as **entryICFGNode** in Algorithm 1. It then constructs an IWTO across the entire sub-ICFG, incorporating its interprocedural edges.

► **Example 12.** Let us revisit the example in Figure 5 and illustrate the three steps in Algorithm 1 used to construct the IWTO of the function partition $\{\text{recur1}, \text{recur2}\}$. In Step 1, we obtain an induced sub-ICFG shown in Figure 6, where call and return sites are delineated with bold outlines. Non-recursive call/return sites are denoted by , whereas recursive call/return sites are indicated by . In Step 2, since the ICFG nodes of **id** are excluded from the sub-ICFG in Step 1, the return site ℓ_{15}^r becomes unreachable from its call site. Consequently, we add an edge from the call site ℓ_{15}^c to its return site, ensuring reachability from the function entry within the sub-ICFG. Finally, in Step 3, we apply Bourdoncle’s algorithm to the partition entries **recur1** and **recur2**, starting from their respective entry ICFG nodes, ℓ_5 and ℓ_{11} . Specifically, initiating from ℓ_5 , we establish the IWTO as: “($\underline{\ell_5} \ \ell_6 \ \ell_7 \ \ell_{11} \ \ell_{12} \ \ell_{13}^c$) $\ell_9 \ \ell_{15}^c \ \ell_{15}^r \ (\underline{\ell_{10}} \ \ell_{13}^r \ \ell_{16} \ \ell_7^r)$ ”. The IWTO starting from ℓ_{11} is outlined as: “($\underline{\ell_{11}} \ \ell_{12} \ \ell_{13}^c \ \ell_5 \ \ell_6 \ \ell_7^c$) $\ell_9 \ \ell_{15}^c \ \ell_{15}^r \ (\underline{\ell_{16}} \ \ell_7^r \ \ell_{10} \ \ell_{13}^r)$ ”.

4.2 IWTO-Guided Abstract Interpretation

This section first presents the overall algorithm for fixpoint computation in §4.2.1, followed by an example set of feasible node-level interpretation rules that can be applied during the fixpoint computation in §4.2.2.

4.2.1 Abstract Interpretation Algorithm

Algorithm 2 presents the pseudo-code for IWTO-guided abstract interpretation, initialized by the **chaoticIteration** function (Line 1). This algorithm takes two inputs: a map linking each partition entry to its corresponding IWTO (**IMap**), as described in §4.1.2, and the program’s entry function, **root** (e.g., the **main** function). Starting from **root**, which is a function partition entry (Definition 8), the function **handlePartition** iterates through the elements in its IWTO (Lines 5-9), processing each as either a single node (Line 7) or a cycle

Algorithm 2 IWTO-guided abstract interpretation.

Input: IMap: map from partition entry to IWTO
 root: program entry function

Output: Abstract states of the program

```

1 Function chaoticIteration():                                // Algorithm entry
2   | handlePartition(dummyCall(⟦, root));
3 Function handlePartition(callSite(⟦, callee)):
4   | IWTO := IMap[callee];
5   | foreach elem ∈ IWTO do
6   |   | if elem is a single node then
7   |   |   | handleNode(callSite, elem);
8   |   | else
9   |   |   | stabilizeCycle(callSite, elem);
10 Function handleNode(pEntryCallsite, node):
11   | Update abstract state of node;
12   | if node is a non-recursive call site then
13   |   | handlePartition(node);                                // Inline
14 Function stabilizeCycle(pEntryCallsite, cycle):
15   | head := cycle.getHead();                                // Cycle head
16   | increasing := true;
17   | while true do
18   |   | handleNode(pEntryCallsite, head);
19   |   | if increasing then
20   |   |   | Widen abstract state of head;                    // Widening
21   |   |   | if abstract state of head not increasing then
22   |   |   |   | increasing := false;
23   |   |   |   | continue;
24   |   | else
25   |   |   | Narrow abstract state of head;                    // Narrowing
26   |   |   | if abstract state of head not decreasing then
27   |   |   |   | break;
28   |   | foreach elem ∈ cycle.getBody() do
29   |   |   | if elem is a single node then
30   |   |   |   | handleNode(pEntryCallsite, elem);
31   |   |   | else
32   |   |   |   | stabilizeCycle(pEntryCallsite, elem);
  
```

(Line 9), where cycles in the IWTO may represent either intraprocedural loops or recursive calls. For handling cycles, we apply the interleaved widening and narrowing strategy [10] to compute a fixpoint that considers both efficiency, achieved through widening, and precision, refined by narrowing.

The subroutine `handleNode`, detailed from Line 10 to Line 13 in Algorithm 2, processes individual nodes. At Line 11, the procedure begins by updating the abstract state node-wise, based on the semantics of the ICFG node. If this node is identified as a non-recursive call site (as defined in Definition 10), `handlePartition` is recursively invoked to process the corresponding callee function. Recursive call and return sites are treated similarly to normal intraprocedural nodes, as their analysis is integrated within the IWTO.

The algorithm for computing a fixpoint for a cycle is shown in the `stabilizeCycle` subroutine, delineated from Line 14 to Line 32 in Algorithm 2. It employs a while loop to continuously process the IWTO elements constituting the cycle. As mentioned in §2.3, each iteration of the loop begins by processing the component head of the IWTO using either widening or narrowing operations, outlined in Lines 18-27, followed by the process other IWTO components (Lines 28-32). The cycle commences with iterations where the widening operation is applied first (Lines 19-23) to the head node, aiming to accelerate the stabilization process. Once the widening has reached a fixpoint, where no changes in the abstract state are observed from one iteration to the next, the algorithm transitions to the narrowing phase (Lines 24-27). This phase refines the analysis's precision, following the methodologies proposed in [10]. The loop concludes when the narrowing phase also achieves a fixpoint, determined at Line 26, where a consistent abstract state is maintained post-narrowing from one loop iteration to the next.

4.2.2 Node-Level Interpretation Rules

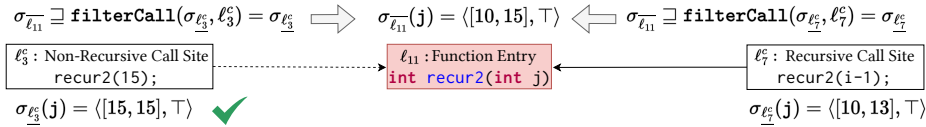
Figure 7 demonstrates an example set of rules for handling each IWTO node, which can be applied at Line 11 in Algorithm 2. Note that these rules can be customized to perform other types of abstract interpretations, such as using different abstract domains or object sensitivity.

The interpretation of each node encompasses two fundamental steps: first, aggregating post-abstract states from predecessors to derive its pre-abstract state; second, transforming the pre-abstract state according to program semantics to obtain the post-abstract state. For interprocedural analysis, abstract states are aggregated via `CALLFLOW` and `RETFLOW` rules, while intraprocedural states are gathered through the `SEQUENCE` rule. Notably, `CALLFLOW` employs selective aggregation of abstract states from call sites: it blocks propagation from mismatching non-recursive call sites while allowing propagation from recursive call sites.

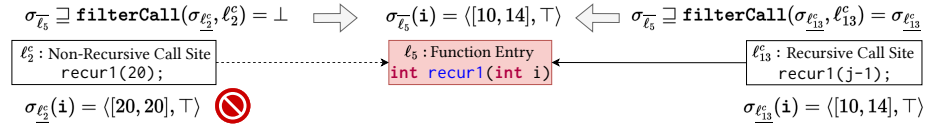
► **Example 13.** Figure 8 illustrates this selective aggregation using the example in Figure 6, demonstrating the application of the `CALLFLOW` rule in deriving pre-abstract states for ℓ_{11} and ℓ_5 . Consider the scenario where abstract interpretation progresses to invoking `handlePartition`($\ell_3^c\langle\text{main}, \text{recur2}\rangle$) to analyze the function partition $\{\text{recur1}, \text{recur2}\}$ from partition entry `recur2`. When deriving the pre-abstract state of ℓ_{11} , as shown in Figure 8a, both $\sigma_{\ell_7^c}$ and $\sigma_{\ell_3^c}$ are permitted to propagate. This is because ℓ_7^c is a recursive call site, whereas ℓ_3^c , while non-recursive, matches the current partition entry's call site, denoted as `pEntryCallsite` in Algorithm 2 at Line 10. Conversely, as shown in Figure 8b, when deriving the pre-abstract state of ℓ_5 , $\sigma_{\ell_2^c}$ is blocked because ℓ_2^c does not match `pEntryCallsite`, ℓ_3^c , whereas $\sigma_{\ell_{13}^c}$ propagates due to the recursive nature of ℓ_{13}^c .

$$\begin{array}{c}
\text{[CALLFLOW]} \frac{\ell@caller \hookrightarrow \ell'@callee}{\sigma_{\ell'} \sqsupseteq \text{filterCall}(\sigma_{\ell}, \ell)} \quad \text{[RETFLOW]} \frac{\ell@caller \hookleftarrow \ell'@callee}{\sigma_{\ell} \sqsupseteq \sigma_{\ell'}} \\
\text{[SEQUENCE]} \frac{\ell \xrightarrow{\text{cond}} \ell'}{\sigma_{\ell'} \sqsupseteq \text{refineSeq}(\sigma_{\ell}, \text{cond})} \quad \text{[CONS]} \frac{\ell : p = c}{\sigma_{\ell} := \sigma_{\ell}[p \mapsto \langle \alpha(\{c\}), \top \rangle]} \\
\text{[COPY]} \frac{\ell : p = q}{\sigma_{\ell} := \sigma_{\ell}[p \mapsto \sigma_{\ell}(q)]} \quad \text{[PHI]} \frac{\ell : r = \phi(p_1, p_2, \dots, p_n)}{\sigma_{\ell} := \sigma_{\ell}[r \mapsto \bigsqcup_{i=1}^n \sigma_{\ell}(p_i)]} \\
\text{[UNARY]} \frac{\ell : p = \neg q}{\sigma_{\ell} := \sigma_{\ell}[p \mapsto \neg^{\#} \sigma_{\ell}(q)]} \quad \text{[BINARY]} \frac{\ell : r = p \odot q}{\sigma_{\ell} := \sigma_{\ell}[r \mapsto \sigma_{\ell}(p) \odot^{\#} \sigma_{\ell}(q)]} \\
\text{[ADDR]} \frac{\ell : p = \&o}{\sigma_{\ell} := \sigma_{\ell}[p \mapsto \langle \top, \{o_i\} \rangle]} \\
\text{[GEP]} \frac{\ell : p = \&(q \rightarrow \text{idx}) \mid p = \&q[\text{idx}]}{\sigma_{\ell} := \sigma_{\ell}[p \mapsto \bigsqcup_{o \in \gamma(\sigma_{\ell}(q))} \bigsqcup_{j \in \gamma(\sigma_{\ell}(\text{idx}))} \langle \top, \{o.\text{fld}_j \} \rangle]} \\
\text{[LOAD]} \frac{\ell : p = *q}{\sigma_{\ell} := \sigma_{\ell}[p \mapsto \bigsqcup_{o \in \gamma(\sigma_{\ell}(q))} \sigma_{\ell}(o)]} \\
\text{[STORE]} \frac{\ell : *p = q}{\sigma_{\ell} := (\{o \mapsto \sigma_{\ell}(q) \mid o \in \gamma(\sigma_{\ell}(p))\} \sqcup \sigma_{\ell} \setminus \text{kill}(\ell))} \\
\text{filterCall}(\sigma_{\ell}, \ell) := \begin{cases} \sigma_{\ell} & \text{if } \ell \text{ is recursive call site } \vee \ell = \text{pEntryCallSite} \\ \perp & \text{otherwise} \end{cases} \\
\text{refineSeq}(\sigma_{\ell}, \text{cond}) := \sigma_{\ell}[p_1 \mapsto \sigma_{\ell}(p_1) \sqcap \alpha(\text{cond}(p_1)), \dots, p_n \mapsto \sigma_{\ell}(p_n) \sqcap \alpha(\text{cond}(p_n))] \\
\text{kill}(\ell : *p = q) := \begin{cases} \{o \mapsto _ \mid o \in \gamma(\sigma_{\ell}(p))\} & \text{if } \sigma_{\ell}(p) = \langle \top, \{o\} \rangle \wedge o \text{ is singleton} \\ \{o \mapsto _ \mid o \in \mathcal{O}\} & \text{if } \sigma_{\ell}(p) = \langle \top, \emptyset \rangle \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

■ **Figure 7** Rules for interpreting each node during abstract interpretation (Line 11 in Algorithm 2). $\ell@caller \hookrightarrow \ell'@callee$ and $\ell@caller \hookleftarrow \ell'@callee$ represent interprocedural control flows. The former indicates a flow from the CALL statement ℓ in **caller** to the entry ℓ' of the **callee** function, while the latter denotes a flow from the exit ℓ' of the **callee** back to the RETURN statement ℓ in **caller**. $\ell \xrightarrow{\text{cond}} \ell'$ denotes an intraprocedural control flow with path condition cond , whose value is null for unconditional flows. $a \sqsupseteq b$ represents $a := a \sqcup b$. $\mathbf{f}^{\#}$ is the abstract operator concerning the concrete operator \mathbf{f} . $\alpha(\text{cond}(p_i))$ abstracts the concrete values of p_i restricted by sequence condition cond based on Definition 1. o.fld_j represents the j^{th} field of the base object o .



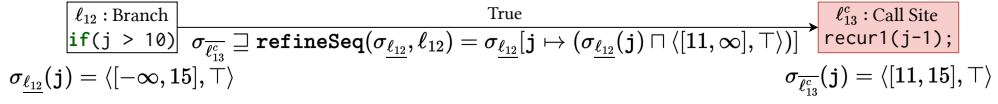
(a) Deriving the pre-abstract state $\sigma_{\ell_{11}}^-$ during running $\text{handlePartition}(\ell_3^c(\text{main}, \text{recur2}))$.



(b) Deriving the pre-abstract state $\sigma_{\ell_5}^-$ during running $\text{handlePartition}(\ell_3^c(\text{main}, \text{recur2}))$.

■ **Figure 8** Illustration of the CALLFLOW rule for deriving pre-abstract state of ℓ_{11} and ℓ_5 by revisiting the example in Figure 6.

466 The RETFLOW rule facilitates the propagation of abstract states from a callee function's
 467 exit node to the corresponding return site. The SEQUENCE rule refines abstract values based
 468 on control flow edge conditions. Figure 9 demonstrates this refinement process for deriving



■ **Figure 9** Illustration of the SEQUENCE rule for deriving the pre-abstract state $\sigma_{\ell'_{13}}^c$ by revisiting the example in Figure 6.

the pre-abstract state of ℓ'_{13} . Since the incoming edge from ℓ_{12} requires condition $i > 10$ to hold, we refine the abstract value of i in $\sigma_{\ell_{12}}$ by computing its meet with $[11, \infty]$.

The post-abstract state σ_{ℓ} of node ℓ is derived from its pre-abstract state σ_{ℓ}^c and the relevant statement. For pointer-free statements (CONS, COPY, PHI, UNARY, BINARY, CALL and RETURN), we calculate the abstract value of the left-hand-side variable based on the right-hand-side values and relevant operators. For instance, given a BINARY statement $\ell : r = p + q$ and $\sigma_{\ell}^c(p) := \langle [1, 3], \top \rangle, \sigma_{\ell}^c(q) := \langle [4, 7], \top \rangle$, we derive $\sigma_{\ell}(r) := \langle [5, 10], \top \rangle$. Particularly, since CALL and RETURN statements involve parameter and return value passing—essentially top-level variable assignments—we handle each assignment using the COPY rule for translation. For pointer-related statements (GEP, LOAD and STORE), we utilize abstract values across interval and address domains to derive the post-abstract state. To elaborate, for GEP, we join all potential address values formed by adding a base address $o \in \gamma(\sigma_{\ell}^c(p))$ with an offset $j \in \gamma(\sigma_{\ell}^c(idx))$ to derive the conservative address value for the left-hand-side variable. As for LOAD, we join the abstract values $\sigma_{\ell}^c(o)$ of all possible addresses $o \in \gamma(\sigma_{\ell}^c(q))$ to form the conservative abstract value of the left-hand-side variable. Lastly, for STORE, we update the abstract value of every possible address $o \in \gamma(\sigma_{\ell}^c(p))$. Particularly, when there are multiple addresses $o \in \gamma(\sigma_{\ell}^c(p))$, we join the abstract value $\sigma_{\ell}^c(q)$ with each existing abstract value $\sigma_{\ell}^c(o)$ to form o 's new abstract value conservatively. In contrast, when there is only one $o \in \gamma(\sigma_{\ell}^c(p))$, it is safe to directly use $\sigma_{\ell}^c(q)$ as o 's new abstract value, because the store must occur at o .

► **Example 14.** We revisit the example in Figure 5 to illustrate the IWTO-guided abstract interpretation. It starts with `handlePartition`($\ell_0(_, \text{main})$). The IWTO of `main` function is outlined as “ $\ell_1 \ell_2^c \ell_2^r \ell_3^c \ell_3^r \ell_4$ ”. Both ℓ_2^c and ℓ_3^c are non-recursive call sites, which require invoking `handlePartition` to handle their callee functions. Take ℓ_3^c as an example. `handlePartition`($\ell_3^c(\text{main}, \text{recur2})$) follows the IWTO of function partition {`recur1`, `recur2`}, which is outlined as “($\underline{\ell_{11}} \ell_{12} \ell_{13}^c \ell_5 \ell_6 \ell_7^c$) $\ell_9 \ell_{15}^c \ell_{15}^r (\underline{\ell_{16}} \ell_7^r \ell_{10} \ell_{13}^r)$ ”, to perform abstract interpretation. The first element of the IWTO, “($\underline{\ell_{11}} \ell_{12} \ell_{13}^c \ell_5 \ell_6 \ell_7^c$)”, is a cycle, with ℓ_{11} being the head. To handle the cycle, the function `stabilizeCycle` is invoked, which iteratively updates the abstract states of the nodes within the cycle. The iteration involves two stages, the widening stage and the narrowing stage. We elaborate on the widening and narrowing stages using the updating process of $\sigma_{\ell_{11}}^c(j)$, as ℓ_{11} is the only widening point of the cycle. During the widening stage, it is initialized as $\langle [15, 15], \top \rangle$ in the first iteration. In the second iteration, it is first updated to $\langle [13, 15], \top \rangle$, and then widened to $\langle [-\infty, 15], \top \rangle$ (refer back to §2.2). The iteration then proceeds to the narrowing stage, during which $\sigma_{\ell_{11}}^c(j)$ is updated to $\langle [10, 15], \top \rangle$, refining the value obtained in the widening stage. This refinement benefits from the adjustments made in the SEQUENCE rule while deriving the pre-abstract state of ℓ_7^c . With the conclusion of the narrowing stage, `stabilizeCycle` finalizes, and the abstract interpreter continues to address the remainder of the IWTO. After finalizing its handling of the entire IWTO, the abstract interpreter proceeds to addressing ℓ_3^r and concludes that the return value of this recursion, entering from ℓ_3^c , is exactly 10. This exact value also benefits from the refinement conducted by the SEQUENCE rule while deriving the pre-abstract state of ℓ_{15}^c .

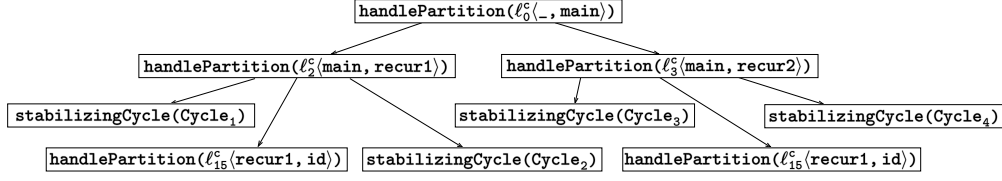


Figure 10 Recursion tree of algorithm 2 executing on the program presented in Figure 5

4.3 Termination Analysis

This section aims to prove that Algorithm 2 is guaranteed to terminate, even in the presence of nested recursions and loops in the input program. Algorithm 2 begins with `chaoticIteration`, which calls `handlePartition` to handle the dummy call site to `root`, the program entry function. It then enters a mutual recursion among three functions: `handlePartition`, `handleNode` and `stabilizeCycle`. Due to the simplicity of `handleNode`, we inline it at the invoking sites in Lines 7 and 30 for the convenience of the proof. In order to model the recursion described in Algorithm 2, we introduce the concept of *recursion tree*.

► **Definition 15** (Recursion Tree). *The recursion tree, denoted as a triple $T = (N, E, r)$, models the invocation process of a running algorithm. The set N is non-empty and consists of nodes, where each $n \in N$ represents an instance of a function invocation with specific parameters during the recursive process. The set $E \subseteq N \times N$ comprises edges, where an edge $(n_1, n_2) \in E$ indicates that function n_1 calls n_2 during the recursion. The element $r \in N$ denotes the root node of T , representing the initial function that starts the recursive calls.*

► **Example 16.** Figure 10 demonstrates the recursion tree of Algorithm 2 as it executes on the program presented in Figure 5. The root node indicates that the recursion of Algorithm 2 begins with `handlePartition(ℓ₀ᶜ⟨_, main⟩)`, where $\ell_0^c\langle_, \text{main}\rangle$ denotes a dummy call to the program entry, `main`. Additionally, the two edges of the root node indicates that it invokes each child subroutine once during execution.

Since all operations in the functions `handlePartition` and `stabilizeCycle` are trivial and guaranteed to terminate, the overall termination of the algorithm can be reduced to the termination of the recursive process. This is equivalent to stating that the recursion tree of Algorithm 2 contains a finite number of nodes for any input.

► **Lemma 17.** *The recursion tree of Algorithm 2 on any input has finite height.*

Proof. Consider a path (sequence of nodes) starting from the root node in the recursion tree. We will show that the number of nodes along this path is bounded by analyzing the invocations of two specific functions: `handlePartition` (referred as P nodes) and `stabilizeCycle` (referred as S nodes). These are the only types of nodes present on the path, as we have inlined all `handleNode` invocations for simplicity.

First, we examine the subsequence of `handlePartition` calls, denoted as:

$$P_1(\ell_1^c\langle\text{caller}_1, \text{callee}_1\rangle), \dots, P_m(\ell_m^c\langle\text{caller}_m, \text{callee}_m\rangle).$$

We use $FP(f)$ to denote the unique function partition to which function f belongs. It follows that for all $i \in [1, m-1]$, $FP(\text{callee}_i) = FP(\text{caller}_{i+1})$. Consequently, for all $i < j$, `callerj` is reachable from `calleri` on the call graph. We aim to prove the following sub-conclusions. **Sub-conclusion 1:** For all $i \leq j$, $FP(\text{caller}_i) \neq FP(\text{callee}_j)$. Suppose this were not the case. Then, there would exists $i \leq j$ such that $FP(\text{caller}_i) = FP(\text{callee}_j)$, implying that `calleri` is reachable from `calleej`. Furthermore, it follows that `callerj` would

also be reachable from `calleej`, which contradicts the definition of $v_j(\text{caller}_j, \text{callee}_j)$ as a non-recursive call site (according to Definition 10). **Sub-conclusion 2:** For all $i < j$, $FP(\text{caller}_i) \neq FP(\text{caller}_j)$. This follows directly from Sub-conclusion 1 and $FP(\text{callee}_i) = FP(\text{caller}_{i+1})$. If $FP(\text{caller}_i) = FP(\text{caller}_j)$ for some $i < j$, it would contradict Sub-conclusion 1. **Sub-conclusion 3:** $m \leq |FPars|$, where $|FPars|$ denotes the number of function partitions in the program. If this were false, then there would exist at least one pair $i < j$ such that $FP(\text{caller}_i) = FP(\text{caller}_j)$, which would contradict Sub-conclusion 2.

Second, we examine a *continuous* subsequence of `stabilizeCycle` invocations,

$S_1(\text{cycle}_1), \dots, S_n(\text{cycle}_n)$.

For this sequence, the inequality $n < \text{cycle}_1.\text{depth}$ is established, where `cycle1.depth` denotes the depth of the `cycle1`. This holds because `cyclei+1` is a sub-IWTO component of `cyclei`. Furthermore, the number of continuous *S* subsequences is limited to length of the longest *P* subsequence on the path, which, as previously concluded, is $|FPars|$. Consequently, the total number of *S* nodes on the path is limited to $|FPars| \times \text{maxCycleDepth}$, where *maxCycleDepth* is the maximum depth of all IWTO cycles.

Finally, since the path consists solely of *P* and *S* nodes, the length of the path is less than $|FPars| \times (\text{maxWTOdepth} + 1)$. ◀

► **Lemma 18.** *The recursion tree of Algorithm 2 for any input has finite degree.*

Proof. To establish this, we consider two cases. **Case 1:** For a $P(\ell^c(\text{caller}, \text{callee}))$ node, its degree is limited by the number of elements in `callee`'s corresponding IWTO. **Case 2:** For a $S(\text{cycle})$ node, its child nodes represent the invoked subroutines within the double-nested loop outlined in Algorithm 2 from Lines 17 to 32. The outer loop is guaranteed to terminate within *h* rounds [13], where *h* represents the height of the lattice when applying the widening operation. The inner loop is bounded by the number of elements in `cycle`. Thus, the degree of an *S* node is also finite.

In both cases, the degree of the nodes are limited, confirming that the recursion tree has finite degree. ◀

► **Theorem 19 (Termination).** *Algorithm 2 is guaranteed to terminate.*

Proof. By Lemmas 17 and 18, both the height and the breadth of the recursion tree are limited. Consequently, the recursion tree contains a finite number of nodes, which implies the termination of Algorithm 2. ◀

5 Evaluation

In this section, we evaluate the performance of RECTOPO, focusing on its precision improvement in abstract interpretation and its scalability when applied to large-scale programs. We conduct a comparative analysis between RECTOPO and three state-of-the-art abstract interpreters, IKOS [15], Clam [19, 34] and CSA [18], in an assertion-checking client, buffer overflow detection.

Additionally, we perform an ablation study to assess the influence of the IWTO implementation on the overall effectiveness of RECTOPO.

Our empirical evaluation addresses the following research questions:

■ **Table 2** Statistics of 10 open-source projects. $\#LOI$ denotes the number of lines of LLVM instructions. $\#Method$, $\#Call$ and $\#Obj$ are the numbers of functions, method calls and memory objects, respectively. $|V|$ and $|E|$ are the numbers of ICFG nodes and ICFG edges.

Project	$\#LOI$	$\#Method$	$\#Call$	$\#Obj$	$ V $	$ E $
<code>paste</code>	8,416	53	758	510	9,395	9,922
<code>md5sum</code>	11,483	63	881	606	12,494	13,064
<code>YAJL</code>	20,592	151	561	208	9,253	9,922
<code>8cc</code>	30,609	480	2,349	3,267	15,984	25,294
<code>libplist</code>	35,324	342	1,743	1,635	13,697	20,978
<code>MP4v2</code>	39,178	601	610	1,991	15,595	16,733
<code>RIOT</code>	54,597	579	1,614	951	20,176	20,843
<code>darknet</code>	210,117	1,004	9,861	2,614	110,424	140,213
<code>tmux</code>	446,626	1,967	22,369	3,879	162,879	178,924
<code>Teeworlds</code>	529,737	2,306	20,701	5,754	251,356	246,029
<i>Total</i>	1,386,679	7,546	61,447	21,415	621,253	681,922

RQ1 **Analysis Effectiveness:** How effective is RECTOPO in detecting buffer overflow vulnerabilities compared to state-of-the-art tools when evaluated on standard NIST benchmarks?

RQ2 **Real-world Applicability:** To what extent can RECTOPO scale to analyze real-world applications while maintaining its precision in bug detection?

RQ3 **Impact of IWTO:** What is the contribution of the IWTO technique to RECTOPO's overall performance, particularly regarding precision improvements and computational overhead?

5.1 Datasets and Implementation

Datasets. We evaluate RECTOPO using (1) a benchmark composed of 8312 programs from NIST [46], which includes comprehensive test cases of buffer overflow vulnerabilities, with 800 cases containing recursions, and (2) 10 popular open-source C/C++ projects across various application domains, with their statistics detailed in Table 2. These projects include `paste` [16] (file merger), `md5sum` [16] (file verifier), `YAJL` [4] (JSON parsing library), `8cc` [7] (C compiler), `libplist` [8] (Property List file handler), `MP4v2` [2] (MP4 file library), `RIOT` [3] (IoT operating system), `darknet` [1] (neural network framework), `tmux` [5] (terminal multiplexer) and `Teeworlds` [9] (online multiplayer game).

Implementation. The experiments are conducted on an Ubuntu 22.04 server with an eight-core 2.60GHz Intel Xeon CPU and 128 GB memory. The interprocedural control and value flow graphs are built upon LLVM-IR (with LLVM version 14.0.0) using the SVF library (version 2.9). The LLVM-IR represents program functions as global variables, further modeled as address-taken variables. The call graph is built by considering the points-to information to resolve indirect calls. Program loops/recursions are captured by IWTO, which is constructed using the algorithm described in Section 4.1 (Algorithm 1). The abstract value representation is implemented using Z3 expressions [45]. We utilize the interleaved widening and narrowing strategy as described in [10] to precisely handle recursions and intraprocedural loops. We detect buffer overflows at each `gep` instruction by comparing the offset size and the memory size, both of which are calculated based on interval and points-to information from abstract interpretation. For the baselines IKOS, Clam and CSA,

■ **Table 3** Comparison of tools using the NIST benchmark, with the recall and precision in percentages (%). ‘NIST (entire)’ refers to the entire dataset, while ‘NIST (recursion)’ specifies the test cases that have recursions.

Dataset	Metric	IKOS	Clam	CSA	RECTOPO
NIST (entire)	Recall (%)	53.90	55.29	80.83	80.83
	Precision (%)	46.38	37.18	78.38	85.97
NIST (recursion)	Recall (%)	100.00	100.00	100.00	100.00
	Precision (%)	50.00	50.00	55.56	95.24

```

1  int mc91(int p) {
2      if(p > 100) {
3          return p - 10;
4      } else {
5          int p1 = p + 11;
6          int p2 = mc91(p1);
7          int res = mc91(p2);
8          return res;
9      }
10 }
11 int main() {
12     char buf[200];
13     int idx = mc91(100);
14     buf[idx] = 0;
15 }

```

```

1  int sum = 0;
2  void adder(int num) {
3      sum += num;
4  }
5  int main() {
6      char buf[10];
7
8      // call adder twice
9      int m = 1;
10     adder(m);
11     int n = 2;
12     adder(n);
13
14     buf[sum] = 0;
15 }

```

(a) Recursive function.

(b) Multiple call sites.

■ **Figure 11** False buffer overflows eliminated by RECTOPO.

we directly use their open-source implementations and their default settings (e.g., interval analysis) for detecting buffer overflows.

5.2 RQ1: NIST Benchmark

Table 3 presents the true positive rates and precision of RECTOPO along with three baseline abstract interpreters in detecting buffer overflows [28, 29, 30] using the pre-labeled NIST benchmark, NIST (complete), and its subset containing test cases that includes recursive functions, NIST (recursion).

Comparison Results. For the complete NIST dataset, RECTOPO consistently surpasses the baseline tools. It identifies, on average, 26.24% more buffer overflows than IKOS and Clam, while also achieving an average precision improvement of 44.19% over these tools. Its precision is also 7.59% higher than CSA. This performance underscores the robustness of RECTOPO in interprocedural abstract interpretation. Moreover, the advantages of RECTOPO are particularly notable in scenarios involving recursive functions, where it achieves a precision rate of 95.24% in detecting buffer overflows. In contrast, the average precision of IKOS, Clam and CSA in these test cases is only approximately 51.85%, 43.39% lower than RECTOPO.

Result Analysis. The improved recall can be attributed to RECTOPO’s adoption of CSA’s robust handling of core program features, such as loop handling, and precise external API modeling [18]. Here, we focus on analyzing the precision improvements in handling recursion through two representative code scenarios shown in Figure 11. A key contributing factor is RECTOPO’s construction of IWTO for recursive functions, which effectively guides the

■ **Table 4** Comparing RECTOPO with three open-source tools (IKOS, Clam and CSA), using ten popular applications. #TP and #FP are true positive and false positive, respectively. Time (secs) is running time.

Project	IKOS			Clam			CSA			RecTopo		
	Report		Time (secs)	Report		Time (secs)	Report		Time (secs)	Report		Time (secs)
	#TP	#FP		#TP	#FP		#TP	#FP		#TP	#FP	
paste	3	21	512	2	15	589	3	0	9	3	0	12
md5sum	2	35	986	1	28	1146	4	1	8	4	2	10
YAJL	1	1625	2895	1	1483	3061	3	0	5	3	0	9
8cc	2	1893	3244	1	683	3824	3	7	36	3	5	38
libplist	1	1206	3312	1	835	4178	5	14	45	5	8	49
MP4v2	1	965	3684	2	681	4375	1	0	13	1	0	16
RIOT	2	1325	5216	2	1206	6387	8	6	27	8	4	31
darknet	14	1265	9531	9	1324	10936	21	10	3507	21	7	3725
tmux	4	1632	11325	2	1359	12894	12	10	824	12	5	983
Teeworlds	2	529	13569	4	1436	14734	15	8	2886	15	8	3032
Total	32	10496	54274	25	9050	62124	75	56	7360	75	39	7905

```

1  int modNb(int n) {
2      if (n - 9 >= 0)
3          return n;
4      else
5          return modNb(n + 1);
6  }
7  int main() {
8      char buf[10];
9      int idx = modNb(0);
10     buf[idx] = 0;
11 }

```

(a) Complex path condition.

```

1  int sum(int n, int m) {
2      if (n <= 0)
3          return m + n;
4      else
5          return sum(n - 1, m + 1);
6  }
7  int main() {
8      char buf[10];
9      int idx = sum(9, 0);
10     buf[idx] = 0;
11 }

```

(b) Variable correlations.

■ **Figure 12** False positives of RECTOPO.

638 narrowing process by leveraging program path conditions. As demonstrated in Figure 11(a),
639 the program accesses the `buf` buffer at ℓ_{14} with an offset `idx`, which is obtained from the
640 return value of the `mc91(100)` function call at ℓ_{13} . RECTOPO precisely determines `mc91`'s
641 return value in the range $[91, 101]$. Consequently, it concludes that the buffer access at
642 ℓ_{14} is safe. In contrast, all our baseline analyses report an overflow alarm here, incorrectly
643 indicating that the value of `mc91(100)` could be anywhere within $[-\infty, \infty]$.

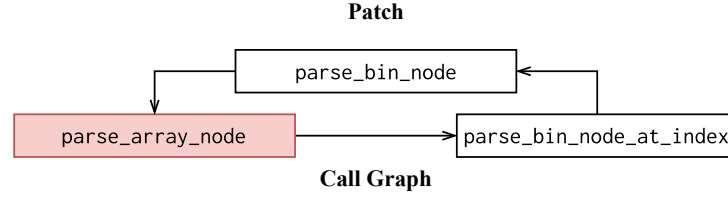
644 Another reason is that the IWTO construction approach effectively eliminates spurious
645 recursions (widening points) arising from multiple call sites of the same function, such as
646 the double call to `adder` in Figure 11(b). RECTOPO accurately differentiates between these
647 call sites, preventing misclassification of program points, $\ell_{12}, \ell_2 - \ell_4, \ell_{10}, \ell_{11}$, as part of a
648 spurious recursion. This precision prevents RECTOPO from widening the value of `sum` at the
649 `adder` entry point (ℓ_2) to $[0, \infty]$. Consequently, RECTOPO derives the precise value of `sum`
650 at ℓ_{14} as $[3, 3]$, thereby eliminating a false overflow alarm.

651 **False Positives of RecTopo.** We further analyze RECTOPO's false positives and identify
652 two primary causes, as demonstrated in the examples in Figure 12. First, RECTOPO
653 compromises precision for efficiency in scenarios involving complex path conditions. A case in

```

1 static plist_t parse_array_node(struct bplist_data *bplist,
2                               const char** bnode, uint64_t size)
3 {
4     plist_data_t data = plist_new_plist_data();
5     const char *index1_ptr = NULL;
6     ...
7     for (uint64_t j = 0; j < size; j++) {
8         index1_ptr = (*bnode) + j * bplist->ref_size;
9 +     if (index1_ptr < bplist->data ||
10 +         index1_ptr + bplist->ref_size >=
11 +         bplist->data + bplist->size) {
12 +         plist_free(data);
13 +         return NULL;
14 +     }
15     index1 = UINT_TO_HOST(index1_ptr, bplist->ref_size);
16     ...
17 }

```



■ **Figure 13** The patch and call graph of a real-world buffer overflow in `libplist` [8] found by our tool.

point is its failure to accurately interpret the path condition at ℓ_2 in Figure 12(a), resulting in an imprecise narrowing of the range of `n` after widening. This imprecision triggers the overflow alarm at ℓ_{10} . This issue primarily stems from the limitations of the abstract interpretation solver, which, although not a key contribution of RECTOPO, affects its precision. Enhancing the solver’s capability for precise path condition interpretation could rectify this. Another source of precision loss is RECTOPO’s inability to track relationships between variables. This limitation is illustrated in Figure 12(b), where the parameters `m` and `n` have a constraint that their sum at each call site remains constant. However, RECTOPO’s interval domain representation only allows for the independent storage of their abstract states, thus failing to capture their interrelation. As a result, when the parameter `m` is widened to $[0, \infty]$, RECTOPO over-approximates `idx`’s range as $[0, \infty]$. Adopting a relational domain to articulate the constraints between program variables could overcome this limitation.

5.3 RQ2: Real-World Projects

Table 4 presents the experimental results for true positives, false positives and running time of RECTOPO compared to the three existing tools (IKOS, Clam and CSA) on ten real-world projects.

Comparison Results and Analysis. RECTOPO demonstrates superior performance across all metrics, successfully identifying 75 real bugs with a precision rate of 65.79%, representing 79.78% of all identified vulnerabilities. In contrast, IKOS and Clam achieve an average recall rate of only 30.32% among discovered bugs. The precision rates of the baseline tools average 19.28%, less than one-third of RECTOPO’s precision. Notably, RECTOPO achieves these significant improvements while maintaining computational efficiency comparable to CSA. The markedly lower false positive rate of RECTOPO can be attributed to its sophisticated abstract interpretation framework, guided by the IWTO approach. This methodology enables precise derivation of abstract states within recursive contexts through carefully orchestrated

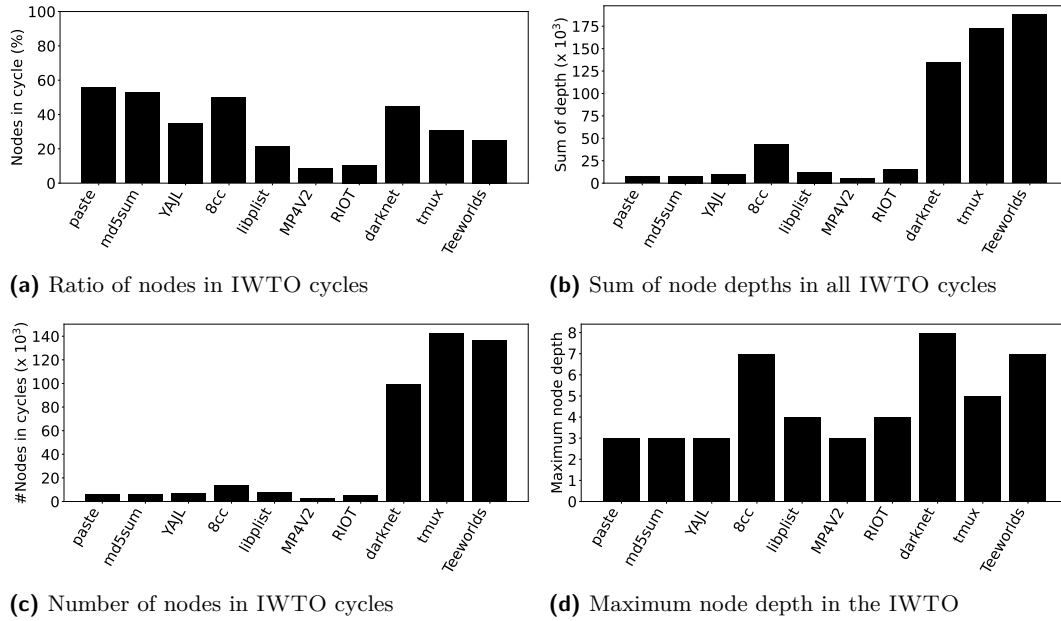


Figure 14 Statistics of IWTO in real-world programs.

interleaved widening and narrowing operations on recursions.

Case Study. Figure 13 illustrates the patch [6] and call graph for a real-world buffer overflow identified by RECTOPO in `libplist` [8], a portable C library to handle Property List files [53]. RECTOPO and CSA initially find this bug, which is missed by Clam and IKOS. However, after the bug is patched, CSA erroneously continue to report it as a buffer overflow—resulting in false positives. By comparison, RECTOPO accurately eliminates the false alarm after the bug is fixed.

The bug resides in the `parse_array_node` function, within a recursive calling chain as depicted in Figure 13. It is triggered at line ℓ_{15} , where the function `UINT_TO_HOST` accesses the address pointed by `index1_ptr` and reads an extra `bplist→ref_size` bytes. This bug arises because `bplist→ref_size`, the buffer access variable in the recursion, is not properly bounded, potentially allowing reads past the allocated buffer of `index1_ptr`, leading to a buffer overflow. When the issue is addressed by introducing a boundary check before ℓ_{15} , RECTOPO stops reporting the problem, while CSA still signal a bug. This discrepancy is attributed to our utilization of widening and narrowing techniques within the recursive function partition, which precisely leverage path conditions to refine the abstract value of `bplist→ref_size`, thereby providing more accurate results and effectively eliminating the false positive.

Statistics. We present detailed statistics of IWTO across the ten real-world projects. Figure 14a illustrates the proportion of nodes residing in IWTO cycles for each program. The data reveals a substantial presence of nodes within IWTO cycles, ranging from 8.4% to 55.9%, with an average of 33.5%. Figure 14b depicts the aggregate depth of nodes (as defined in §2.3) across all IWTO cycles in each program. The data exhibits a near-linear growth pattern relative to program size. Given that the time complexity for cycle stabilization is linear with respect to the sum of node depths [13], this suggests that the overall time complexity for stabilizing IWTO cycles scales approximately linearly with program size. Figure 14c presents the number of nodes within IWTO cycles per program, while Figure

■ **Table 5** Comparing RECTOPO with its two variants, RECTOPO-NR and RECTOPO-GL (described in §5.4), using ten popular applications. #TP and #FP are true positive and false positive, respectively. Time (secs) is running time.

Project	RECTOPO-NR			RECTOPO-GL			RecTopo		
	Report		Time (secs)	Report		Time (secs)	Report		Time (secs)
	#TP	#FP		#TP	#FP		#TP	#FP	
paste	3	13	8	3	32	15	3	0	12
md5sum	4	9	6	4	36	13	4	2	10
YAJL	3	62	3	3	97	12	3	0	9
8cc	3	85	30	3	108	44	3	5	38
libplist	5	119	39	5	230	52	5	8	49
MP4v2	1	131	11	1	193	23	1	0	16
RIOT	8	126	22	8	285	35	8	4	31
darknet	21	217	3168	21	914	4211	21	7	3725
tmux	12	253	727	12	1322	1247	12	5	983
Teeworlds	15	304	2491	15	1641	3416	15	8	3032
<i>Total</i>	75	1319	6505	75	4858	9068	75	39	7905

14d shows the maximum node depth in each program’s IWTO. The data demonstrates that while the number of nodes in cycles increases linearly with program scale, the maximum depth remains relatively stable. This relationship explains the linear growth pattern observed in the aggregate node depth measurements.

5.4 RQ3: Ablation Analysis

This section presents a comprehensive ablation analysis of RECTOPO, with a focus on evaluating its precision enhancements and scalability.

We compare RECTOPO with two distinct variants: RECTOPO-NR, which omits recursive calls by assigning a default top value; and RECTOPO-GL, which integrates a whole-program global IWTO. This comparative analysis underscores the critical contributions of IWTO in enhancing both precision and scalability.

Table 5 presents the true positives, false positives and running time of RECTOPO compared to its two variants (RECTOPO-NR and RECTOPO-GL) on real-world popular applications. The results demonstrate that the IWTO employed in RECTOPO significantly enhances precision while incurring only a moderate increase in computational overhead.

Comparison with RecTopo-NR. RECTOPO exhibits 60.41% higher precision than RECTOPO-NR. This superior precision stems from RECTOPO’s ability to more precisely calculate the values of variables within recursive functions. In contrast, RECTOPO-NR simplifies the process by assigning a generic top value to these variables.

This overly conservative approaches of RECTOPO-NR results in broader intervals that frequently trigger false overflow alarms during buffer access. In terms of performance efficiency, RECTOPO-NR leads in execution speed as it bypasses the body of recursive functions entirely.

However, the slight time advantage of RECTOPO-NR does not compensate for the substantial precision gains achieved by RECTOPO, reinforcing its effectiveness in balancing precision with performance.

Comparison with RecTopo-GL. RECTOPO-GL implements a global whole-program

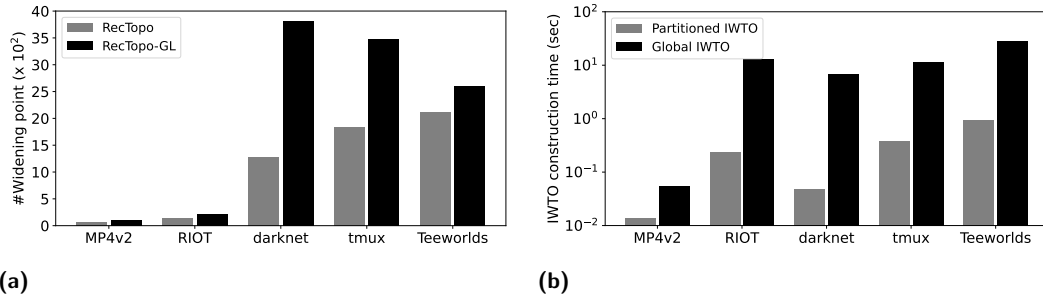


Figure 15 (a) Comparing the number of widening points of RECTOPO and RECTOPO-GL on real-world programs and (b) Comparing IWTO construction time of partitioned and global IWTO on real-world programs.

IWTO that establishes connections between all call sites and their corresponding callee functions. The abstract interpretation within RECTOPO-GL is managed by this singular IWTO, with each cycle processed with precision comparable to that of RECTOPO. Despite this, RECTOPO-GL demonstrates a reduction in precision by 64.27%, and it also operates 1.15 \times longer compared to RECTOPO.

The reduced precision in RECTOPO-GL can be primarily attributed to its redundant widening operations. As depicted in Figure 15a, RECTOPO achieves a significant decrease in the number of widening points, with an average reduction of 58.99% and a peak reduction of up to 81.56%. This substantial decrease in unnecessary widening throughout various program procedures directly contributes to the enhanced precision of RECTOPO. Regarding scalability, the comparison of the time required to construct IWTO in RECTOPO and RECTOPO-GL highlights significant differences. Figure 15b clearly shows that constructing a whole-program IWTO in RECTOPO-GL demands considerably more time—nearly 36 times longer—than the IWTO approach used in RECTOPO. This comparison underscores the efficiency and practicality of the partitioned approach in handling scalability.

6 Related Work

Our discussion centers on areas closely related to RECTOPO, specifically focusing on abstract interpretation, fixpoint computation, and recursion handling in program analysis.

Abstract Interpretation. Abstract interpretation [24, 27] serves as the theoretical foundation for numerous static analysis techniques, encompassing program verification [31, 15, 42, 66, 67], data-flow analysis [50, 51], and static bug detection [59, 60, 18, 55, 56, 63, 48]. These techniques are fundamentally designed to ensure both state over-approximation and analysis termination. Some approaches [48, 36] utilize sparse abstract interpretation framework that propagates abstract states through def-use chains instead of control flows. In contrast, our work operates within the context of control-point-based analysis. Note that this assumption is a design choice made to align with classical abstract interpretation formulations [13, 18, 20, 40, 47], though our approach is not fundamentally restricted to this scope. Weiss et al. [65] proposes database-backed analysis with graph algorithms to enhance both scalability and precision. More recently, Cheng et al. [18] developed cross-domain abstract interpretation to track correlations across different abstract domains. While significant, these approaches are orthogonal to our work.

Fixpoint Computation. Fixpoint computation, originally conceptualized by Cousot et al. [24], forms the theoretical cornerstone of abstract interpretation by providing conservative

approximations of program runtime behaviors. For acyclic control flows, fixpoint computation typically terminates after a few iterations over the CFG. However, in the presence of cyclic control flows, the fixpoint computation may require an impractically large number of iterations to converge, or in the worst case, may never converge, especially when using abstract domains characterized by infinite or very large lattice heights. To address this issue, widening operators [23] are introduced to accelerate the fixpoint computation and ensure its termination. Since its introduction, Bourdoncle’s algorithm [13] has remained the predominant approach for efficient fixpoint computation in abstract interpretation. Subsequent research has largely built upon this foundation, exploring various refinements such as interleaved widening and narrowing strategies to enhance precision [10, 11, 35, 33] and parallel computation techniques [40, 64, 44]. These advances, while valuable, are orthogonal to our focus on interprocedural fixpoint computation. Current interprocedural methods typically either inline callee functions [15], overlooking opportunities for interprocedural topological ordering, or disregard varying calling contexts [48, 47], potentially introducing superfluous widening operations.

Recursion Handling. In interprocedural static analysis, recursive program structures present significant challenges by potentially generating unbounded call stacks that can affect analyzer termination. Existing approaches address this challenge through various strategies: some employ fixed-depth recursive call unrolling [59, 55, 56], while others explicitly bound the analysis of recursive calls [26, 37]. These approaches offer practical trade-offs between precision and analysis feasibility. More conservative methods focus on computing sound approximations of values within recursive structures [15, 18], prioritizing soundness while potentially sacrificing precision when analyzing recursions. Oh et al. [49] proposed techniques for eliminating spurious interprocedural cycles, though not specifically targeting recursion handling. Rival et al. [54] developed specialized methods for handling recursion in shape analysis, focusing primarily on precise approximation of recursive heap structures—an approach complementary to our work. Keidel et al. [38, 14] introduces a terminable big-step abstract interpretation approach even in the presence of loops and recursions. It achieves this by identifying recurrent call traces of the abstract interpreter itself during runtime. However, while this dynamic approach allows the application of widening on recursions, it currently does not support further narrowing to improve precision. Späth et al. [57] proposes a synchronized pushdown system that improves time complexity for recursive data structures. It operates within a pushdown framework, which differs from our abstract interpretation methodology. Stein [58] introduces a versatile framework for incremental and demand-driven tabulation-based abstract interpretation, utilizing dynamic summarization graphs to support sound analysis of recursion. While effective, it does not incorporate Bourdoncle’s strategy, which could enhance efficiency in handling cycles.

7 Conclusion

This paper introduces RECTOPO, a novel approach to fixpoint computation for precise interprocedural abstract interpretation. RECTOPO utilizes interprocedural weak topological ordering to steer chaotic iterations across program procedures effectively. This strategy not only incorporates an interleaved widening and narrowing approach for accurately addressing recursions but also eliminates redundant widening caused by interprocedural dependencies. RECTOPO demonstrates an average precision improvement of 46.51% and an increased recall rate of 32.98% when compared to baseline tools on ten open-source projects.

References

- 1 Darknet: Open source neural networks in C. <https://github.com/pjreddie/darknet>, 2023.
- 2 MP4v2: A C/C++ library to create, modify and read MP4 files. <https://github.com/enzo1982/mp4v2/>, 2023.
- 3 RIOT: The friendly OS for IoT. <https://github.com/RIOT-OS/RIOT>, 2023.
- 4 YAJL: A fast streaming JSON parsing library in C. <https://github.com/lloyd/yajl>, 2023.
- 5 Tmux: Tmux source code. <https://github.com/tmux/tmux>, 2023.
- 6 A buffer overflow patch in libplist. <https://github.com/libimobiledevice/libplist/commit/4765d9a60ca4248a8f89289271ac69cbffcc29bc>, 2024.
- 7 8cc: C compiler. <https://github.com/rui314/8cc>, 2024.
- 8 libplist: A small portable C library to handle Apple Property List files in binary, XML, JSON, or OpenStep format. <https://github.com/libimobiledevice/libplist>, 2024.
- 9 Teeworlds: A retro multiplayer shooter. <https://teeworlds.com/>, 2024.
- 10 Gianluca Amato and Francesca Scozzari. Localizing widening and narrowing. In *International Static Analysis Symposium*, pages 25–42. Springer, 2013. doi:10.1007/978-3-642-38856-9_4.
- 11 Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. Efficiently intertwining widening and narrowing. *Science of Computer Programming*, 120:1–24, 2016. doi:10.1016/j.scico.2015.12.005.
- 12 George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *SAS '16*, 2016. doi:10.1007/978-3-662-53413-7_5.
- 13 François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications: International Conference Academgorodok, Novosibirsk, Russia June 28–July 2, 1993 Proceedings*, pages 128–141. Springer, 2005. doi:10.1007/BFb0039704.
- 14 Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. Modular abstract definitional interpreters for webassembly. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, pages 5–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.
- 15 Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings 12*, pages 271–277. Springer, 2014. doi:10.1007/978-3-319-10431-7_20.
- 16 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, page 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.
- 17 Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *International Static Analysis Symposium*, pages 309–325. Springer, 2009. doi:10.1007/978-3-642-03237-0_21.
- 18 Xiao Cheng, Jiawei Wang, and Yulei Sui. Precise sparse abstract execution via cross-domain interaction. In *46th International Conference on Software Engineering, ICSE '24*. ACM/IEEE, 2024. doi:10.1145/3597503.3639220.
- 19 Clam. Clam: LLVM front-end for Crab, 2024. <https://github.com/seahorn/clam>.
- 20 Patrick Cousot. Asynchronous iterative methods for solving a fixpoint system of monotone equations. Technical report, Research Report IMAG-RR-88, Université Scientifique et Médicale de Grenoble, 1977.
- 21 Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. PhD thesis, Institut National Polytechnique de Grenoble-INPG; Université Joseph-Fourier . . . , 1978. URL: <https://tel.archives-ouvertes.fr/tel-00288657>.

- 861 22 Patrick Cousot. Semantic foundations of program analysis. In *Program flow analysis: theory*
862 *and applications*, pages 303–342. Prentice Hall, 1981.
- 863 23 Patrick Cousot. Abstracting induction by extrapolation and interpolation. In Deepak D’Souza,
864 Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract*
865 *Interpretation*, pages 19–42, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.
866 1007/978-3-662-46081-8_2.
- 867 24 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static
868 analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th*
869 *ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252,
870 1977. doi:10.1145/512950.512973.
- 871 25 Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and*
872 *computation*, 2(4):511–547, 1992. doi:10.1093/logcom/2.4.511.
- 873 26 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David
874 Monniaux, and Xavier Rival. The Astrée analyzer. In *Programming Languages and Systems:*
875 *14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European*
876 *Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8,*
877 *2005. Proceedings 14*, pages 21–30. Springer, 2005. doi:10.1007/978-3-540-31987-0_3.
- 878 27 Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. A²I: Abstract² Interpretation.
879 *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi:10.1145/3290355.
- 880 28 CWE-121: Stack-based buffer overflow. [https://cwe.mitre.org/data/definitions/121.](https://cwe.mitre.org/data/definitions/121.html)
881 [html](https://cwe.mitre.org/data/definitions/121.html), 2024.
- 882 29 CWE-122: Heap-based buffer overflow. [https://cwe.mitre.org/data/definitions/122.](https://cwe.mitre.org/data/definitions/122.html)
883 [html](https://cwe.mitre.org/data/definitions/122.html), 2024.
- 884 30 CWE-126: buffer over-read. <https://cwe.mitre.org/data/definitions/126.html>, 2024.
- 885 31 Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in
886 polynomial time. PLDI ’02, page 57–68, New York, NY, USA, 2002. Association for Computing
887 Machinery. doi:10.1145/512529.512538.
- 888 32 Matt Elder. Bourdoncle components. 2010.
- 889 33 Denis Gopan and Thomas Reps. Lookahead widening. In Thomas Ball and Robert B. Jones,
890 editors, *Computer Aided Verification*, pages 452–466, Berlin, Heidelberg, 2006. Springer Berlin
891 Heidelberg. doi:10.1007/11817963_41.
- 892 34 Arie Gurfinkel and Jorge A. Navas. Abstract interpretation of LLVM with a region-based
893 memory model. In *Software Verification: 13th International Conference, VSTTE 2021,*
894 *New Haven, CT, USA, October 18–19, 2021, and 14th International Workshop, NSV 2021,*
895 *Los Angeles, CA, USA, July 18–19, 2021, Revised Selected Papers*, page 122–144, Berlin,
896 Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-95561-8_8.
- 897 35 Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In Antoine Miné
898 and David Schmidt, editors, *Static Analysis*, pages 198–213, Berlin, Heidelberg, 2012. Springer
899 Berlin Heidelberg. doi:10.1007/978-3-642-33125-1_15.
- 900 36 Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In
901 *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 289–298.
902 IEEE, 2011. doi:10.1109/CGO.2011.5764696.
- 903 37 Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme
904 Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival, et al. Astrée: Proving the absence of
905 runtime errors. *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)*, 9, 2010.
- 906 38 Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. Combinator-based fixpoint algorithms
907 for big-step abstract interpreters. *Proceedings of the ACM on Programming Languages*,
908 7(ICFP):955–981, 2023. doi:10.1145/3607863.
- 909 39 Sol Kim, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Widening with thresholds via
910 binary search. *Software: Practice and Experience*, 46(10):1317–1328, 2016. arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2381>, doi:10.1002/spe.2381.
911

- 912 40 Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. Deterministic parallel fixpoint
913 computation. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi:10.1145/3371082.
- 914 41 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis
915 & transformation. In *International symposium on code generation and optimization, 2004.*
916 *CGO 2004.*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.
- 917 42 Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. A dual number abstraction
918 for static analysis of Clarke Jacobians. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.
919 doi:10.1145/3498718.
- 920 43 Zohar Manna. *Mathematical theory of computation*. Dover Publications, Inc., 2003.
- 921 44 David Monniaux. The parallel implementation of the Astrée static analyzer. In Kwangkeun Yi,
922 editor, *Programming Languages and Systems*, pages 86–96, Berlin, Heidelberg, 2005. Springer
923 Berlin Heidelberg. doi:10.1007/11575467_7.
- 924 45 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International*
925 *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages
926 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.
- 927 46 NIST datasets. <https://samate.nist.gov/SARD/test-suites/116>, 2023.
- 928 47 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and
929 implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd*
930 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI
931 '12, page 229–238, New York, NY, USA, 2012. Association for Computing Machinery. doi:
932 10.1145/2254064.2254092.
- 933 48 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. The SPARROW static
934 analyzer, 2012. <https://opam.ocaml.org/packages/sparrow/>.
- 935 49 Hakjoo Oh and Kwangkeun Yi. An algorithmic mitigation of large spurious interprocedural
936 cycles in static analysis. *Software: Practice and Experience*, 40(8):585–603, 2010. doi:
937 10.1002/spe.969.
- 938 50 Komal Pathade and Uday P. Khedker. Computing partially path-sensitive MFP solutions
939 in data flow analyses. In *Proceedings of the 27th International Conference on Compiler*
940 *Construction*, CC 2018, page 37–47, New York, NY, USA, 2018. Association for Computing
941 Machinery. doi:10.1145/3178372.3179497.
- 942 51 Komal Pathade and Uday P. Khedker. Path sensitive MFP solutions in presence of intersecting
943 infeasible control flow path segments. In *Proceedings of the 28th International Conference on*
944 *Compiler Construction*, CC 2019, page 159–169, New York, NY, USA, 2019. Association for
945 Computing Machinery. doi:10.1145/3302516.3307349.
- 946 52 D.J. Pearce, P.H.J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM*
947 *TOPLAS*, 30(1):4–es, 2007. doi:10.1145/1290520.1290524.
- 948 53 Property list. https://en.wikipedia.org/wiki/Property_list, 2024.
- 949 54 Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. *SIGPLAN*
950 *Not.*, 46(1):173–186, jan 2011. doi:10.1145/1925844.1926406.
- 951 55 Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint:
952 Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th*
953 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages
954 693–706, 2018. doi:10.1145/3192366.3192418.
- 955 56 Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse analysis
956 without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference*
957 *on Programming Language Design and Implementation*, PLDI 2021, page 930–943, New York,
958 NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454086.
- 959 57 Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow
960 analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming*
961 *Languages*, 3(POPL):1–29, 2019. doi:10.1145/3290361.

- 962 **58** Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Interactive abstract interpretation
963 with demanded summarization. *ACM Transactions on Programming Languages and Systems*,
964 46(1):1–40, 2024. doi:10.1145/3648441.
- 965 **59** Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow
966 analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*,
967 ISSTA '12, pages 254–264. ACM, 2012. doi:10.1145/2338965.2336784.
- 968 **60** Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse
969 value-flow analysis. *IEEE Trans. Software Eng (TSE '14)*, 40(2):107–122, 2014. doi:10.
970 1109/TSE.2014.2302311.
- 971 **61** Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*,
972 1(2):146–160, 1972. doi:10.1137/0201010.
- 973 **62** Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.
- 974 **63** Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Detecting blocking errors
975 in go programs using localized abstract interpretation. In *Proceedings of the 37th IEEE/ACM*
976 *International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA,
977 2023. Association for Computing Machinery. doi:10.1145/3551349.3561154.
- 978 **64** Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for
979 large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on*
980 *Programming Language Design and Implementation*, PLDI '04, page 231–242, New York, NY,
981 USA, 2004. Association for Computing Machinery. doi:10.1145/996841.996869.
- 982 **65** Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. Database-backed program analysis
983 for scalable error propagation. In *2015 IEEE/ACM 37th IEEE International Conference on*
984 *Software Engineering*, volume 1, pages 586–597, 2015. doi:10.1109/ICSE.2015.75.
- 985 **66** Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. Program analysis via efficient
986 symbolic abstraction. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi:10.1145/
987 3485495.
- 988 **67** Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang,
989 Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for
990 static finite-state property checking of large-scale systems code. EuroSys '19. ACM, 2019.
991 doi:10.1145/3302424.3303972.