ENHANCING POINTS-TO ANALYSIS FOR STATIC DATA RACE DETECTION IN LARGE INDUSTRIAL PROGRAMS

YANG JIAWEI

MPhil

The Hong Kong Polytechnic University

2026

The Hong Kong Polytechnic University

Department of Computing

Enhancing Points-to Analysis for Static Data Race Detection in Large Industrial Programs

YANG Jiawei

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Philosophy

August 2025

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____ (Signed)

YANG Jiawei (Name of student)

Abstract

The prevalence of parallel programming underscores the importance of static data race detection for software reliability. However, its effectiveness is often limited by imprecise points-to analysis. Conventional approaches, such as Andersen's points-to analysis, struggle to distinguish between fields within compound data structures, such as structs and arrays, leading to a high rate of false positives that hinders developer trust and adoption. In contrast, abstract interpretation-based points-to analysis offers better precision for handling compound data structures but faces scalability issues and may struggle to handle recursion precisely and efficiently.

This thesis presents a two-pronged approach to enhance the precision of points-to analysis for race detection, without compromising scalability on large industrial programs. First, it introduces SLIRD, a framework designed to accurately resolve points-to relationships for compound data structures while maintaining scalability. SLIRD introduces a novel program slicing method to narrow the analysis on a minimal yet complete code scope, thereby ensuring scalability for the subsequent cross-domain abstract interpretation, which provides precise alias information to facilitate the downstream race detection.

Second, this thesis also tackles a key challenge in abstract interpretation: the precise handling of recursion. To address this challenge, this thesis proposes REC-

TOPO, an innovative interprocedural fixpoint computation technique. RECTOPO establishes a new interprocedural weak topological ordering (IWTO) that allows recursive functions to be analyzed with the same precision and efficiency as loops. By systematically decomposing the program and managing analysis states across recursive call chains, this method avoids the significant over-approximations prevalent in existing tools.

Empirical evaluation confirms the efficacy of both contributions. SLIRD demonstrates an average precision improvement of 89.1% over the baseline tool SVFRD on large-scale industrial programs while only incurring an average 4.8-fold overhead. RECTOPO shows a significant precision improvement of 46.51% and an increased bug detection recall rate of 32.98% on real-world recursive benchmarks. Collectively, this thesis delivers a more robust points-to analysis framework capable of tackling the complexities of large industrial parallel programs.

Publications arising from the thesis

[1] Jiawei Yang, Xiao Cheng, Bor-Yuh Evan Chang, Xiapu Luo, and Yulei Sui. "Taming and Dissecting Recursions Through Interprocedural Weak Topological Ordering". In: *39th European Conference on Object-Oriented Programming (ECOOP 2025)*. Ed. by Jonathan Aldrich and Alexandra Silva. Vol. 333. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 34:1–34:31. ISBN: 978-3-95977-373-7. DOI: 10.4230/LIPIcs.ECOOP.2025.34.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Xiapu Luo, for his patient guidance throughout my entire study. His profound knowledge and experience in this field have been invaluable in preparing me with foundational knowledge in my research area. He has helped me shape my research mindset and skills, greatly inspiring my work. He has also enabled me to identify valuable research questions and conduct rigorous research to address these questions effectively.

Next, I would like to extend my heartfelt thanks to Prof. Yulei Sui, who provided me with the valuable opportunity to work on the impactful project SVF. His advice on academic presentations and guidance for collaboration on open-source projects have been immensely beneficial for my academic career. I would also like to thank Prof. Bor-Yuh Evan Chang, who possesses profound knowledge in static analysis, particularly in abstract interpretation, and has provided insightful advice on my research work.

I am also deeply grateful to Dr. Xiao Cheng and Xudong Wang, whose profound expertise and brilliant insights have enriched my research journey. Their valuable experience in conducting research and their practical suggestions on the two projects have been instrumental. Collaborating with them has helped me

progress rapidly.

I would like to thank my colleagues for fostering a supportive and inspiring environment for research. Their brilliant ideas and engaging discussions have broadened my research perspectives and sparked the generation of numerous innovative ideas.

Lastly, I would like to thank all my friends in Hong Kong. I have cherished every moment spent with you during my two years of study in Hong Kong, and I am fortunate to have shared this journey with you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Due to the increasing demand for computational power, multicore architecture and parallel programs have become increasingly prevalent in modern computing [3, 40]. It is reported that a significant portion of modern applications are parallel programs [37, 72]. However, due to the inherent complexity of parallel programming, data races have become a significant problem for developers [77]. Data races occur when multiple threads concurrently access shared data, with at least one thread writing to it, without proper synchronization, leading to unpredictable behaviors. The non-deterministic execution order of parallel threads makes detecting such races through dynamic testing exceptionally difficult [77, 99]. To address these issues, static data race detection techniques have been proposed [10, 31, 42, 68, 76, 36, 69]. These techniques conservatively approximate program behaviors, enabling the detection of data races or proving their absence.

A fundamental prerequisite for static data race detection is obtaining points-to

information for functions, locks, and accessed data [10, 42, 31, 36]. This information serves as the foundation for subsequent thread analysis and race detection. The precision of the points-to information is crucial for downstream race detection, as spurious alias relationships can lead to downstream race detector reporting false race alarms [31].

1.2 Research Gap and Motivation

Pure points-to analyses [31, 10, 47, 51], however, faces significant challenges in providing precise alias information for pointers within compound data structures, such as structs or arrays. For instance, Steensgaard's points-to analysis [81], adopted by a recent static race detector, RacerF [31], utilizes a unification-based approach that provides efficient but coarse-grained alias information. SVF [83] conducts flow-sensitive pointer analysis on an over-approximated value-flow graph built using Andersen's points-to analysis [7]. Sparrow [61] employs a flow-insensitive pointer analysis to support the subsequent scalar analysis. These approaches conduct points-to analysis on a sole memory address domain. Consequently, the points-to analysis fails to distinguish between different fields within the same compound data structure, such as a structs or arrays. Since multi-thread programs often use structs/arrays to manage the thread functions and accessed data, this imprecise points-to information can result in numerous false positives in data race detection.

To distinguish between fields in a compound data structure, it is necessary to model scalar behavior for offset information. Abstract interpretation [24, 23], a general framework for static analysis, provides a practical method for modeling this scalar behavior. Within this framework, an interval domain models the scalar

behavior of variables, while a memory address domain models their points-to behavior. Furthermore, combining the address and interval domains yields more precise points-to information [18]. Although effective for deriving precise points-to information for compound data structures, abstract interpretation can exhibit exponential complexity in context-sensitive analyses [18], limiting its scalability to large-scale projects. This highlights the need for a method to enhance the efficiency of abstract interpretation.

Program slicing is a promising solution to enhance the efficiency [91]. It extracts subsets of code that preserve specific behaviors, aiding tasks like debugging, testing, and static analysis. Empirical evaluations conducted by Chalupa and Strejček [15] report that program slicing has a positive impact and can significantly enhance the performance of software verification tools. Sun et al. [86] employs slicing to enhance the performance of model invariant checking. However, these methods lack uniformity, as no single approach consistently addresses all aspects of static analysis requirements. For race detection in particular, existing static slicing techniques [52, 15, 86] fall short because they are primarily designed for sequential programs and thus fail to address concurrency-specific issues such as may-happen-in-parallel (MHP) relations or race patterns.

Moreover, a significant limitation of abstract interpretation is its difficulty in handling recursion, a common control structure in real-world programs. While state-of-the-art abstract interpreters effectively leverage Bourdoncle's algorithm for loop analysis, they face difficulties when determining WTO for recursive function calls. Several approaches have been proposed for taming recursions and interprocedural abstract interpretation. For non-recursive functions, function inlining enables precise interprocedural analysis. However, this approach becomes infea-

sible for recursive functions due to potentially infinite inlining sequences in unbounded recursion. To address this, some tools adopt strong assumptions, employing either under- or over-approximation approaches. For example, Astrée [43] completely excludes the analysis of unbounded recursion, while IKOS [13, 46] conservatively assigns top values ($\top$) to variables in recursive contexts, effectively bypassing the analysis of recursive functions. To improve precision, tools such as Clam [38, 19] and CSA [18] analyze recursive function bodies but introduce many spurious widening operations at function boundaries. Their effectiveness remains limited, as their WTO computations are 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 data race detection.

1.3 Aims and Objectives

This research aims to advance static data race detection by developing a points-to analysis that achieves high precision while maintaining scalability for large-scale industrial programs.

To efficiently generate precise points-to information for race detection, this research introduces a three-stage framework, SLIRD. The first stage employs program slicing to extract a minimal, sound code slice relevant to race detection. The second stage applies cross-domain abstract interpretation to compute a refined points-to information by combining address and interval domains, incorporating offset information to distinguish between fields within compound structures. Fi-

nally, this information is utilized in thread analysis to detect data races.

Additionally, to enhance the precision of abstract interpretation-based points-to analysis in handling recursion, a novel interprocedural fixpoint computation technique, RecTopo, is introduced. It computes interprocedural weak topological ordering (IWTO) to handle recursions as efficiently and precisely as loops. RecTopo systematically decomposes programs into non-recursive and recursive components by applying Tarjan's strongly connected component algorithm [88] on the call graph. Using these decompositions, RecTopo coalesces recursive components within the control flow graph and computes unified IWTO for each. This approach avoids the complexity of whole-program WTO construction. Leveraging the constructed IWTO, RecTopo integrates an efficient fixpoint computation algorithm with interleaved widening and narrowing operations, enabling precise and scalable recursion analysis.

1.4 Contribution

This research has made the following major contributions:

- This work introduces SliRD, a novel static data race detection framework that leverages cross-domain abstract interpretation to provide precise points-to information for race detection. Furthermore, SliRD incorporates a sound slicing method to reduce the analysis scope, ensuring scalability on large industrial programs.

- It presents RecTopo, a new technique that significantly enhances the precision of abstract interpretation for recursive programs. RecTopo introduces

an efficient interprocedural weak topological ordering (IWTO) computation and IWTO-guided fixpoint computation, enabling precise handling of recursions and benefiting downstream analyses such as data race and buffer overflow detection.

- An empirical evaluation demonstrating that SLIRD achieves an average precision improvement of 89.1% while only incurring an average 4.8 times overhead compared to SVFRD across five large-scale industrial programs. Furthermore, RECTOPO demonstrates an average precision improvement of 46.51% and an increased recall of 32.98% on real-world bugs when compared to baseline tools across ten open-source projects.

Chapter 2

Related Work

This chapter discusses areas related to this thesis. It reviews the literature across several interconnected domains: static data race detection, the underlying principles of points-to analysis, program slicing as a scalability-enhancing technique, the formal framework of abstract interpretation, and specialized methods for recursion handling. This review highlights existing limitations and motivates the novel approaches developed in SLIRD and RECTOPO.

2.1 Static Data Race Detection

Static race detection identifies potential data races in concurrent programs without executing them, using conservative approximations of program behavior. Since its proposal, researchers have made efforts to balance precision and scalability. An early tool, RacerX [36], is a static analyzer for C programs based primarily on a lockset analysis. The tool employs heuristics to improve precision and has successfully detected bugs in the kernels of several operating systems. Compared to

SLIRD, RacerX utilizes a relatively simple points-to analysis for the representation of locks. Locksmith [69], a context-sensitive analysis for C programs, uses points-to information to identify races by correlating lock usage. Chord [58], designed for Java, integrates alias analysis with happens-before relations to reduce false positives. Despite their precision, both approaches face scalability challenges when applied to large industrial codebases. Recent advancements target precision and scalability for large programs. RacerD [10] is a compositional analyser implemented in the Infer framework. It achieves scalability by analyzing each function only once. RacerF [31], a thread-modular analyzer for C. Although scalable, RacerF leverages Steensgaard's algorithm [81] to obtain alias information, making it insensitive to flow, context and fields, which leaves room for precision improvements. OMPRacer [87], which focuses on OpenMP programs, performs a context- and field-sensitive Andersen-like points-to analysis [7]. Despite their efficacy, these methods all perform pure points-to analyses by relying on a single address domain and omitting scalar information, thereby failing to model complex data structures precisely. While some approaches employ abstract interpretation [24] to provide more precise analysis results, such as Astrée [26, 25] and Goblint [94], they exhibit the inherent challenge of the exponential complexity of context-sensitive analysis.

2.2 Points-to Analysis

Points-to analysis determines the memory locations pointers may reference, serving as a foundation for race detection. Various approaches differ in precision and scalability. Andersen's analysis [7], a flow-insensitive, context-insensitive

inclusion-based method, models points-to sets with cubic time complexity, offering high precision for many programs. Steensgaard's algorithm [81], a unification-based approach, merges aliasing nodes to achieve near-linear time complexity but sacrifices precision due to bidirectional flow. Type-based analyses [8, 51] leverage type information for faster approximations. While techniques such as scaling type-based points-to analysis can achieve rapid results, they do so at the cost of precision and may only provide alias information for function pointers. Modern tools like the SVF [83] framework perform iterative value-flow and pointer analysis on LLVM IR, enabling scalable interprocedural analysis with customizable precision. Despite their efficacy, these methods are often limited by the omission of scalar information, which leads to the imprecise modeling of compound data structures such as arrays. This limitation necessitates cross-domain approaches to enhance precision. While some approaches employ abstract interpretation [18, 26] to provide more precise analysis results, they exhibit the inherent challenge of the exponential complexity of context-sensitive analysis, requiring a method to improve their scalability in large programs.

2.3 Program Slicing

Program slicing extracts code subsets that preserve specific behaviors, aiding tasks like debugging and testing. First introduced by Weiser in 1981 [95], this technique reduces programs to minimal forms while maintaining subsets of behavior, with applications in debugging, maintenance, optimization, and program analysis. Dynamic slicing, as proposed by Agrawal and Horgan in 1990 [4], utilizes execution traces for higher precision by considering specific inputs and runtime paths, mak-

ing it suitable for scenarios where exact behavior during execution needs to be isolated. In contrast, our work focuses on static slicing, which computes slices without requiring execution inputs or traces. This allows for broader coverage of potential program behaviors without dependency on specific runtime conditions.

Comprehensive surveys by Tip [91] outlines various static slicing techniques, including path slicing and their applications in debugging, program integration, dataflow testing, and software maintenance. Lydia, Srinath, and Gyani [52] provide a detailed analysis of static slicing for object-oriented programs, highlighting challenges in handling polymorphism and inheritance, which are critical for modern software systems. Empirical evaluations conducted by Chalupa and Strejček [15] report that program slicing has a positive effect and can significantly enhance the performance of software verification tools. Sun et al. [86] employs slicing to enhance the performance of model invariant checking. However, these methods lack uniformity, as no single approach consistently addresses all aspects of static analysis requirements. For race detection in particular, existing static slicing techniques [15, 86] fall short because they are primarily designed for sequential programs and thus fail to address concurrency-specific issues such as may-happen-in-parallel (MHP) relations or race patterns. Consequently, these methods may produce incomplete or imprecise slices by including irrelevant code or missing subtle race conditions caused by non-deterministic thread interleavings[10]. This gap underscores the need for the specialized slicing approach proposed in SLIRD, which targets race detection by integrating points-to and MHP analysis to preserve race-relevant behaviors.

2.4 Abstract Interpretation

Abstract interpretation [24, 27] serves as the theoretical foundation for numerous static analysis techniques, encompassing program verification [33, 13, 49, 98, 100], data-flow analysis [65, 66], and static bug detection [85, 84, 18, 78, 79, 93, 61]. These techniques are fundamentally designed to ensure both state over-approximation and analysis termination. Some approaches [61, 39] utilize sparse abstract interpretation framework that propagates abstract states through def-use chains instead of control flows to improve efficiency. In contrast, RecTopo 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 [11, 18, 20, 46, 60], though the proposed approach is not fundamentally restricted to this scope. Weiss, Rubio-González, and Liblit [96] proposed a database-backed analysis using graph algorithms to improve scalability. This approach leverages the extensive storage capacity of a persistent abstract state management system, enabling more fine-grained analysis of large codebases. While significant, this approach addresses challenges that are orthogonal to the primary contributions of this thesis. Cheng, Wang, and Sui [18] recently introduced cross-domain abstract interpretation to track correlations across multiple abstract domains. Additionally, they proposed a hash-consed approach to manage points-to sets, significantly reducing memory overhead, which is incorporated in SLiRD. However, these approaches do not solve the problem of exponential complexity inherent in context-sensitive analysis—a challenge addressed by SLiRD, the first contribution of this thesis. Furthermore, they do not handle recursion effectively, which necessitates the specialized approach of RecTopo.

2.5 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 [85, 78, 79], while others explicitly bound the analysis of recursive calls [26, 43]. These approaches offer practical trade-offs between precision and analysis feasibility. More conservative methods focus on computing sound approximations of values within recursive structures [13, 18], prioritizing soundness while potentially sacrificing precision when analyzing recursions. Oh and Yi [62] proposed techniques for eliminating spurious interprocedural cycles through matching call/return sites during fixpoint computation, however, this approach fails to leverage the efficacy of Bourdoncle's fixpoint computation algorithm for recursion handling. Rival and Chang [75] developed specialized methods for handling recursion in shape analysis, focusing primarily on precise approximation of recursive heap structures—an approach complementary to RecTopo. Keidel, Erdweg, and Hombücher [44] and Brandl et al. [12] 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, Ali, and Bodden [80] proposes a synchronized pushdown system that improves time complexity for recursive data structures. It operates within a pushdown framework, which differs from the proposed abstract interpretation methodology. Stein, Chang, and Srid-

haran [82] 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.

Chapter 3

Enhancing Precision in Points-to Analysis for Data Race Detection While Preserving Scalability

3.1 Introduction

Parallel programs have become increasingly prevalent in modern computing architectures [3, 40]. The inherent complexity of parallel programs often leads to data races. A data race occurs when multiple threads concurrently access shared data, with at least one access being a write, without proper synchronization, leading to unpredictable behaviors. Detecting these data races through testing is challenging [99] because races occur unpredictably due to the non-deterministic execution order of parallel threads. To address these issues, static data race detection techniques have been proposed [42]. These techniques soundly approximate program behaviors, enabling detecting data races or proving their absence.

A preliminary step for static data race detection is obtaining points-to information for functions, locks, and accessed data. This information serves as the foundation for subsequent thread analysis and race detection [42]. The precision of the points-to information is crucial for downstream race detection, as spurious alias relationships can result in false data race alarms. However, pure points-to analysis [83, 47, 51] faces significant challenges in providing precise alias information for pointers within compound data structures, such as structs or arrays. For instance, SVF [83] conducts flow-sensitive pointer analysis on an over-approximated value-flow graph built using Andersen's points-to analysis [7]. Sparrow [61] employs an flow-insensitive pointer analysis to support the subsequent scalar analysis. These approaches conducts points-to analysis on a sole memory address domain. Consequently, the points-to analysis fails to distinguish between different fields within the same compound data structure, such as a structs or arrays. Since multi-thread programs often uses structs/arrays to manage the thread functions and accessed data, this imprecise points-to information can result in numerous false positives in data race detection.

To distinguish between fields in a compound data structure, it is necessary to model scalar behavior for offset information. Abstract interpretation [24, 23], a general framework for static analysis, provides a practical method for modeling this scalar behavior. Within this framework, an interval domain models the scalar behavior of variables, while a memory address domain models their points-to behavior. Furthermore, combining the address and interval domains yields more precise points-to information, namely the cross-domain abstract interpretation approach [18]. Although precise, abstract interpretation can exhibit exponential complexity in context-sensitive analyses, hindering its applicability to large-scale, real-

world projects.

To tackle these challenges, this study aims to employ cross-domain abstract interpretation to achieve more points-to analysis and further ensure its scalability through a slicing method. Base on the observation that data race detection actually requires only partial points-to information, it is possible to analyze only a portion of the program while still providing the necessary information for the downstream tasks. The major challenge lies in extracting the minimal yet complete code slice to analyze.

This study proposes SLIRD, a three-stage method. First, an initial slicing step extracts a minimal yet complete code slice. Second, a cross-domain abstract interpretation on the slice provides the necessary points-to information. Finally, this information is used to perform thread analysis and race detection.

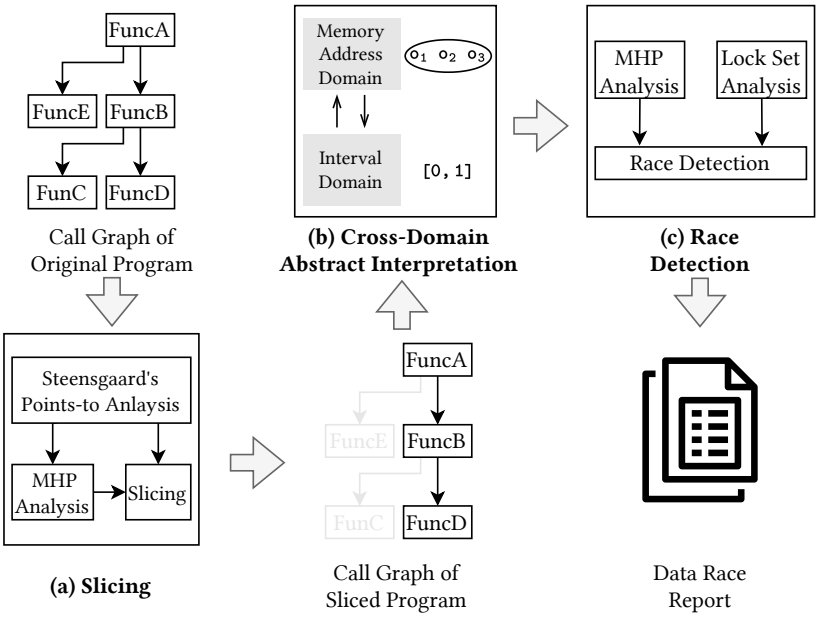


Figure 3.1: Overview of SLIRD.

Figure 3.1 illustrates the three-stage architecture of SLIRD:

(a) Slicing: The proposed slicing method extracts minimal yet complete code for the the subsequent race detection. It employs Steensgaard's algorithm for points-to analysis due to its linear complexity, which minimizes overhead. Next, it performs a may-happen-in-parallel (MHP) analysis using the acquired points-to information. Finally, it generates a function-level slice of the original program that preserves the two race patterns summarized in this study.

(b) Cross-domain abstract interpretation: This abstract interpretation step combines address and interval domains to obtain a refined points-to information. It incorporates offset information to distinguish between fields within the same compound structure.

(c) Race detection: This step performs MHP and lockset analyses to inform the final race detection.

In summary, this study makes the following contributions:

- This study proposes a three-stage approach that first performs slicing to reduce the code scope for analysis, then applies cross-domain abstract interpretation to provide precise points-to information, and finally conducts race detection based on the points-to results.

- This study develops a novel code slicing method that extracts a minimal yet complete code slice for subsequent analysis, accelerating the analysis while maintaining precision and soundness.

- This study implements the proposed method in a tool named SLIRD, based on the SVF framework [83]. Evaluation on five large-scale industrial programs demonstrates that SLIRD achieves an average precision improvement

of 89.1% while incurring an average 4.8 times the execution time of SVFRD across five large-scale industrial programs, all with a retained soundness.

3.2 Motivating Example

Static data race detection relies on precomputed points-to information for thread functions, accessed data, and locks. The precision of this information is crucial for the subsequent race detection as spurious alias relationships can lead to a high volume of false positives. Pure points-to analysis, which typically models program behavior using only an address domain, often fails to distinguish between different fields within the same compound data structure, resulting in such false alarms.

Figure 3.2 illustrates such a situation. In the example, `thread_function1` increments the first element of a global array `arr`, while `thread_function2` increments the second. These functions are spawned in `main` and can execute concurrently. However, because they access distinct elements of `arr`, no data race exists.

However, a single-domain points-to analysis (e.g., SVF [83], RacerF [31], Goblint [94]) treats the entire `arr` as a single object, failing distinguish between its elements. Consequently, a downstream race detector would judge that the write operations at Line 7 and Line 11 could modify the same object, reporting a false positive. This scenario is common when analyzing real-world programs, which often store thread function pointers and accessed data in compound structures like structs and arrays. This false positive could be eliminated by SliRD's cross-domain abstract interpretation, which treats the elements of `arr` distinctly. It informs the race detector that Line 7 and Line 11 write to different memory locations,

```
1  int arr[10];  // global array
2  void sequential_function(int input) {
3      int result = input * 2;
4      printf(%d", result);
5  }
6  void* thread_function1(void* arg) {
7      arr[0]++;
8      return NULL;
9  }
10 void* thread_function2(void* arg) {
11     arr[1]++;
12     return NULL;
13 }
14 int main() {
15     pthread_t thread1, thread2;
16
17     sequential_function(5);
18
19     pthread_create(&thread1, NULL, thread_function1, NULL);
20     pthread_create(&thread2, NULL, thread_function2, NULL);
21
22     pthread_join(thread1, NULL);
23     pthread_join(thread2, NULL);
24 }
```

Figure 3.2: A parallel program with irrelevant sequential code.

thus preventing the false alarm.

Moreover, revisiting Figure 3.2, it is apparent that there exists some code segments, such as `sequential_function`, that are irrelevant for data race detection, since they do not influence data access in parallel threads. Excluding these segments could mitigate the performance issue by focusing points-to analysis solely on relevant code. Specifically, for race detection, static analysis must first determine the points-to sets for `thread_function1`, `thread_function2`, and the object referenced by `arr[0]` and `arr[1]`. Using this information, a subsequent May-Happen-in-Parallel (MHP) analysis can deduct that the two thread functions

may execute concurrently. The final detection step then checks for conflicting memory accesses. Crucially, the MHP analysis and final race check only require pointer information related to the threads and the shared array, while the analysis of `sequential_function` does not contribute.

Nevertheless, a whole-program points-to analysis must process the entire program, including such irrelevant functions. Since context sensitive abstract interpretation can exhibit exponential complexity, analyzing this redundant code severely impairs scalability.

To address the performance issue, this research proposes a novel slicing method that can preserve the essential code for race detection, while eliminating irrelevant code. This method employs a fast but imprecise pointer analysis to generate a call graph. It then performs a slicing on the graph, guided by data-flow information relevant to potential races. This method allows the subsequent, precise points-to analysis to focus exclusively on relevant code by excluding redundant segments (e.g., `sequential_function`), thereby improving scalability.

3.3 Approach

This research proposes first employing a fast points-to analysis and may-happen-in-parallel (MHP) analysis, using their results to guide program slicing for a reduced codebase, detailed in Section 3.3.1. Subsequently, it uses a precise points-to analysis to support the subsequent MHP analysis, lock set analysis, and final race detection, as detailed in Section 3.3.3.

3.3.1 Slicing

Race detection-targeted slicing relies on a preliminary points-to analysis and MHP analysis, which provides the parallel information at the function level. This research employs the Steensgaard algorithm to compute the points-to information, as its linear complexity ensures scalability for analyzing the entire program. Points-to information can be formalized as a function $PTS := \mathcal{V} \rightarrow \mathcal{O}$, where $\mathcal{V} = \mathcal{P} \cup \mathcal{O}$ represents all program variable, consisting of top-level variables $\mathcal{P}$ and virtual objects $\mathcal{O}$. The MHP analysis then provides parallelization information, formalized as a set $MHP_F \subseteq F \times F$, where F denotes all functions in the program. Each pair $(f_1, f_2) \in MHP_F$ indicates that f_1 may be executed concurrently with f_2 during real execution. Using the information, slicing can be performed to minimize the code scope for a more precise subsequent analysis.

To identify the required code for race detection, this study first summarizes race cases into two primary categories. Figure 3.3 illustrates the two categories summarized. Figure 3.3a demonstrate concurrent access via global, where the thread access the raced object through global variables, while Figure 3.3b depicts parallel access to variables shared via pointers, where the thread accesses the object through pointers passed as arguments.

To address these patterns, this study proposes a function-level slicing method based on a concurrent call graph (CCG) that captures both normal function calls and thread creations via `pthread_create`. The algorithm is detailed in Algorithm 1. It consists of two phases: initial slicing and backward slicing. Initial slicing aims at identifying all functions that may contain a race site, including all thread functions, functions that may execute concurrently with them, and their transi-

Algorithm 1: Slicing Algorithm

Input: Points-to analysis result *PTS*, concurrent call graph $CCG(F, E)$, and MHP analysis result MHP_F

Output: Sliced program P'

```
1  Function Slicing(PTS, CCG, MHP_F):
2      Initialize slice S := {};
3      // Initial slicing
4      Compute the transitive points-to closure of all global variables PTSC_GLOBAL;
5      foreach f ∈ F do
6          if f is a thread function then
7              S := S ∪ {f};
8              Compute f's argument's transitive points-to closure PTSC_ARG;
9              Compute f's transitive callees TC_f ;
10             foreach (f, f') ∈ (MHP_F ∪ TC_f) do
11                 if f' accesses an object o ∈ PTSC_GLOBAL ∪ PTSC_ARG then
12                     S := S ∪ {f'};
13     // Backward slicing
14     Init work list wl := [F], visited set visited := ∅;
15     while wl ≠ ∅ do
16         f = wl.pop();
17         foreach (f', f) ∈ E do
18             S := S ∪ f';
19             if f' ∉ visited then
20                 visited = visited ∪ {f'};
21                 wl.push(f');
22     return Program P' induced by S ;
```

```c
int counter = 0;

void* threadFunc(void* arg) {
    counter++;
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL,
        threadFunc, NULL);
    pthread_create(&t2, NULL,
        threadFunc, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

(a) Concurrent access to a shared object via global variables

```c
void* threadFunc(void* arg) {
    int* c = (int*)arg;
    (*c) ++;
    return NULL;
}

int main() {
    int c;
    pthread_t t1, t2;
    pthread_create(&t1, NULL,
        threadFunc, &c);
    pthread_create(&t2, NULL,
        threadFunc, &c);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

(b) Concurrent access to a shared object via pointers

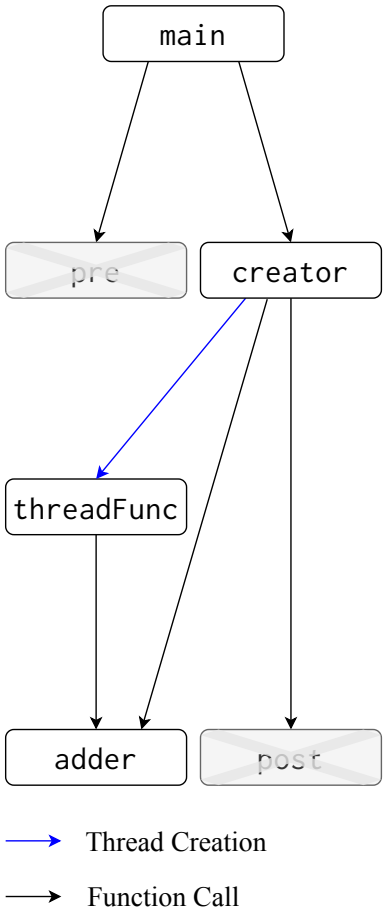
Figure 3.3: Two data race patterns

tive callees. For each thread function f, the slice includes f itself. Its transitive callees and concurrent functions is selectively included based on the two previously summarized race patterns. To address concurrent access via global variables, the slice includes functions accessing the transitive points-to closure of global variables ($PTSC_{GLOBAL}$ in the algorithm), defined as the set of all objects reachable through a pointer and any subsequent pointers, recursively, until no further pointers are found. Similarly, to address concurrent access via pointers, the slice includes functions accessing the transitive points-to closure of f's argument ($PTSC_{ARG}$ in algorithm), representing all objects accessible through the argument pointer in the thread. Subsequent backward slicing provides the context for potential race sites.

The remainder of this section uses the example in Figure 3.4 to elaborate on the slicing algorithm. The CCG of the code is shown in Figure 3.4b. In the ex-

```c
void adder(int *i){
    (*i) ++;
}
void* threadFunc(void* arg) {
    int *c = (int *)arg;
    adder(c);
    return NULL;
}
void post(int *i){
    (*i) = 0;
}
void creator(int *counter) {
    pthread_t t;
    pthread_create(&t, NULL,
    threadFunc, counter);
    printf("%d", *counter);
    pthread_join(t, NULL);
    post(counter);
}
void pre(int *i){
    (*i) = 0;
}
int main() {
    int counter;
    pre(&counter);
    creator(&counter);
}
```

(a) Source code



(b) CCG for code in Figure 3.4a

Figure 3.4: Data race example

ample, `threadFunc` is executes concurrently with `adder`, and `threadFunc` also invokes `adder`. This results in an access race between the two accesses at Line 12 invoked from `creator` and `threadFunc` respectively. When performing slicing for this example, the algorithm first identifies functions that may contain race sites through a initial slicing, which checks the functions that is in `threadFunc`'s thread or could execute parallelly with `threadFunc`'s thread , and only include those who might access the same variable with the created thread to include in the sliced program. For example, function `adder` is included as it is called by `threadFunc` and it accesses objects in `threadFunc`'s argument's points-to set, while `creator` is included since it may happen in parallel with `threadFunc` and also accesses objects in `threadFunc`'s argument's points-to set. A subsequent backward slicing provides the context for static analysis through iteratively include callers of the function in the sliced function set. For example, `main` is included in the backward slicing. The final sliced program includes $\{$`adder`, `threadFunc`, `creator`, `main`$\}$, successfully excluding two redundant functions `pre` and `post`.

3.3.2 Cross-domain abstract interpretation

SLIRD employs cross-domain abstract interpretation to provide precise points-to information for downstream race detection. The cross-domain abstract interpretation utilizes a composite abstract domain $\mathbb{A}_C := \mathbb{A}_{ADDR} \times \mathbb{A}_{ITV}$ that combines the address domain $\mathbb{A}_{ADDR}$ and interval domain $\mathbb{A}_{ITV}$. Abstract interpretation collects program semantics by deriving the abstract state of program variables (Definition 1) at each program program point to construct an abstract trace σ (Definition 2). This is achieved by performing fixpoint computation on a system of

equations (Definition 3).

Definition 1 (Abstract State $AS : \mathcal{V} \to \mathbb{A}$)**.** The *abstract state* for a given program is defined as a map $AS = \mathcal{V} \to \mathbb{A}$ from program variables $\mathcal{V}$ to the abstract domain $\mathbb{A}$. $\mathcal{V}$ encompasses top-level variables and address taken variables (i.e. memory objects), whose addresses are abstracted using $\mathbb{A}_{ADDR}$.

Definition 2 (Abstract Trace $\sigma : \mathbb{L} \to AS$)**.** The *abstract trace* is a set of abstract states, where each abstract state is associated with a specific program point. $\sigma_L(x)$ denotes the value of variable x at a program point $L \in \mathbb{L}$, where this value is an element of the abstract domain $\mathbb{A}$. The program point L can be one immediately preceding ($\bar{\ell}$) or succeeding ($\underline{\ell}$) a program statement ℓ.

Definition 3 (Equation System $\sigma = \Phi(\sigma)$)**.** The *equation system* for a program represents its abstract semantics. The system comprises a set equations, one for each program point ℓ_i, of the form $\sigma_{\underline{\ell_i}} = \Phi_i(\sigma_{\underline{\ell_1}}, \sigma_{\underline{\ell_2}}, \ldots, \sigma_{\underline{\ell_n}})$. Each equation reflects the concrete semantics of corresponding program statement ℓ_i, defining the relationship between the abstract state of that program point ($\sigma_{\underline{\ell_i}}$) and the states at related program points.

Figure 3.5 presents the equations defined for four types of statements crucial for points-to analysis (AddrStmt, GepStmt, LoadStmt, and StoreStmt). AddrStmt, denoted as `p = &o`, allocate address-taken objects `o`, which may represent stack variables, global variables, or abstract heap objects. GepStmt enables structured access to composite objects: struct fields are accessed via constant offsets `fld`, while array elements use variable offsets `idx`.

For GepStmt with a variable offset, a precise field- and array-sensitive analysis captures the correlation between the two domains by using the numerical value of

the offset to refine the resulting memory addresses of an object's fields. The refined pointer result remains conservative, as the interval value of the offset variable is an over-approximated. In contrast, a standalone pointer analysis must either consider all possible fields or aggregate them into a single object to ensure soundness. This approach, however, can be overly conservative and thus compromise precision.

For a LoadStmt of the form $\ell : \mathtt{p} = *\mathtt{q}$, the analysis refines the derived value of p based on the precise memory addresses of q and the abstract values of the addresses from the abstract trace $\sigma_{\overline{\ell}}$. The analysis first obtain the memory addresses of q from the memory address domain. For each memory address o that q refers to, the analysis retrieves the abstract value (either an interval or a set of addresses) associated with o. The join of all abstract values for all such memory addresses constitutes the final result for $\sigma_{\underline{\ell}}(\mathtt{p})$.

For a StoreStmt of the form $\ell : *\mathtt{p} = \mathtt{q}$, the analysis updates the abstract value of every possible address o that p refers to. Particularly, when there are multiple addresses $\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))$, the abstract value $\sigma_{\overline{\ell}}(\mathtt{q})$ is joint with each existing abstract value $\sigma_{\overline{\ell}}(\mathtt{o})$ to form o's new abstract value conservatively. In contrast, when there is only one $\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))$, it is safe to directly use $\sigma_{\overline{\ell}}(\mathtt{q})$ as o's new abstract value, because the store must occur at o.

3.3.3 Race Detection

Race detection is performed on a sliced version of the program, which provides a focused scope. It utilizes the previously computed points-to information to support subsequent MHP analysis, lockset analysis and race detection. The points-to information can be formalized as a query $\mathit{PTS}(\ell, v) = \sigma_{\overline{\ell}}(v)$, where $\mathcal{L}$ denote

$$[\textsc{AddrStmt}]\ \frac{\ell : \mathtt{p} = \mathtt{alloc}_{\mathtt{o_i}}}{\sigma_{\underline{\ell}}(\mathtt{p}) := \langle \top, \{\mathtt{o_i}\}\rangle}$$

$$[\textsc{GepStmt}]\ \frac{\ell : \mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{i})\ \mathit{or}\ \ell : \mathtt{p} = \&(\mathtt{q}[\mathtt{i}])}{\sigma_{\underline{\ell}}(\mathtt{p}) := \bigsqcup_{\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{q}))} \bigsqcup_{j \in \gamma(\sigma_{\overline{\ell}}(\mathtt{i}))} \langle \top, \{\mathtt{o.fld}_j\}\rangle}$$

$$[\textsc{LoadStmt}]\ \frac{\ell : \mathtt{p} = *\mathtt{q}}{\sigma_{\underline{\ell}}(\mathtt{p}) := \bigsqcup_{\mathtt{o} \in \sigma_{\overline{\ell}}(\mathtt{q})} \sigma_{\overline{\ell}}(\mathtt{o})}$$

$$[\textsc{StoreStmt}]\ \frac{\ell : *\mathtt{p} = \mathtt{q}}{\sigma_{\underline{\ell}} := (\{o \mapsto \sigma_{\overline{\ell}}(\mathtt{q}) | o \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))\} \sqcup \sigma_{\overline{\ell}} \setminus \mathsf{kill}(\ell))}$$

$$\mathsf{kill}(\ell : *\mathtt{p} = \mathtt{q}) := \begin{cases} \{\mathtt{o} \mapsto \_ \mid \mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))\} & \text{if } \sigma_{\overline{\ell}}(\mathtt{p}) = \langle \top, \{\mathtt{o}\}\rangle \wedge \mathtt{o} \text{ is singleton} \\ \{\mathtt{o} \mapsto \_ \mid \mathtt{o} \in \mathcal{O}\} & \text{if } \sigma_{\overline{\ell}}(\mathtt{p}) = \langle \top, \varnothing\rangle \\ \varnothing & \text{otherwise} \end{cases}$$

Figure 3.5: Analysis rules for cross-domain abstract interpretation. $\sigma_{\overline{\ell}}$ is obtained through joining the abstract states of program points right after the program statements before ℓ. $\gamma(\sigma(\mathtt{p}))$ concretes the abstract value $\sigma(\mathtt{p})$ and returns the memory address set pointed by $\mathtt{p}$ or the integers residing inside the interval value of variable $\mathtt{p}$. $\mathtt{o.fld}_j$ represents the j^{th} field of the base object $\mathtt{o}$. $\mathtt{o.fld}_j$ represents the j^{th} field of the base object $\mathtt{o}$.

a specific program statement, and v is a program variable. A subsequent MHP analysis provides a fine-grained parallel information that can be formalized as a set $\mathit{MHP}_S := S \times S$, where S denotes all statements in a program. Each pair $(s_1, s_2) \in \mathit{MHP}_S$ indicates that statement s_1 and s_2 could execute in parallel. Followed by that, a subsequent lock set analysis is performed to provide the lock set information for each statement. The lock set information can be formalized as a function $\mathit{LS} := S \rightarrow \mathcal{P}(L)$, where $\mathcal{P}(L)$ denotes the power set of the lock set L. Using the aforementioned information, the analysis can perform a final race detection, as detailed in algorithm 2. This algorithm iterates over all distinct unordered statement pairs that contains at least one memory write statement (store statement in LLVM). Two statements is identified as potential races when and only when: 1) they could happen in parallel according to the MHP analysis result, 2) their lock set has no intersection, meaning their parallel execution could not be forbidden by the

same lock, 3) they may access the same object according to the points-to analysis result.

Algorithm 2: Data Race Detection Algorithm

Input: Andersen's points-to analysis result *PTS*, MHP analysis result MHP_S, and lock set analysis result LS
Output: Data race set $DRS \subseteq S \times S$

```
1 Function detectRace(MHP_S, LS, PTS):
2     Initialize data race set DRS := {};
3     Let S_W ⊆ S be the set of write statements, and S_{R/W} be the set of
        read/write statements;
4     foreach (s1, s2) ∈ S_W × S_{R/W} such that ID(s1) ≤ ID(s2) do
5         if (s1, s2) ∈ MHP_S then
6             if LS(s1) ∩ LS(s2) = ∅ then
7                 if PTS(s1, access(s1)) ∩ PTS(s2, access(s2)) ≠ ∅ then
8                     DRS := DRS ∪ {(s1, s2)};
9     return DRS ;
```

The remainder of this section uses the same example to illustrate the race detection process. Since the sliced program only contains four functions: `threadFunc`, `adder`, `main`, `creator`, the initial points-to analysis is restricted the sliced code base. Followed by that, the MHP analysis provides the statements that could happen in parallel, which indicates that the statement in Line 2 could happen in parallel with Line 16. The subsequent lock set analysis indicates that Line 2 and Line 16 has no common lock. Finally the points-to analysis indicates that they access the same variable, `counter` in `main`. Therefore, they are judged as potential race.

3.4 Evaluation

This section evaluates the effectiveness of SLIRD in detecting data races in large programs, as detailed in Section 3.4.1. It presents an ablation analysis demonstrating how the slicing and cross-domain abstract interpretation module affect the race detection efficiency and precision, as described in Section 3.4.2.

Implementation SLIRD is built on the SVF framework [83]. SVF implementations both Steensgaard's and Andersen's points-to analysis and a data race detection module that uses these points-to results. This module encompasses MHP analysis and lockset analysis for its race detection client. In addition, it implements a cross-domain abstract interpretation with a buffer overflow detection client. SLIRD extends SVF by introducing a slicing module and adapting its cross-domain abstract interpretation module to provide points-to information for the race detection client. For comparison, we use the data race detection component from SVF, based on Steensgaard's points-to analysis, as our baseline, which we refer to as SVFRD.

Table 3.1: Statistics of five open-source C/C++ programs. *#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 interprocedural control flow graph nodes and edges.

Project	*#LOI*	*#Method*	*#Call*	*#Obj*	$\|V\|$	$\|E\|$
`Memcached`	54,218	392	1,437	532	23,180	24,890
`OpenBLAS`	108,732	984	2,783	1,689	35,210	36,890
`Qt`	2,729,645	10,536	124,876	29,917	1,253,720	1,402,185
`MySQL`	2,837,102	17,823	219,415	20,321	1,286,450	1,441,032
`OpenCV`	3,264,930	9,357	134,502	31,204	1,451,600	1,518,300
Total	8,994,627	39,092	483,013	83,663	4,050,160	4,423,297

Experimental setup The dataset consists of 5 large-scale open-source parallel programs. Table 3.1 presents their statistics. These programs include `Memcached` [54]

(a cache management library), `OpenBLAS` [63] (a linear algebra library), `Qt` [71] (a GUI library), `MySQL` [57] (a relational database manager), and `OpenCV` [64] (an image/video precessing library). Experiments were conducted on an Ubuntu 22.04 server with an eight-core 2.60GHz Intel Xeon CPU and 512 GB of RAM.

3.4.1 Effectiveness on Large Programs

This experiment evaluates the effectiveness of SLIRD against three other baselines RACERF [31], GOBLINT [94], and SVFRD on five large programs. Table 3.2 presents the results. SLIRD demonstrates higher precision and notable speedups compared with RACERF and GOBLINT, successfully completing the analysis of three additional benchmarks that these tools fail to finish. RACERF and GOBLINT both suffer from scalability limitations: the former relies on Andersen's points-to analysis, while the latter uses abstract interpretation as its backend. Neither approach scales effectively to codebases with millions of lines of instructions.

Notably, SLIRD detects the same 12 true positives as SVFRD. This result verifies that the program slices are complete (sound) and exclude no data race-related code. Furthermore, compared with SVFRD, SLIRD reports fewer false positives across all five programs, achieving an average reduction of 89.1%. This improvement in precision can be attributed to the more precise points-to information provided by the cross-domain abstract interpretation of SLIRD. Although SLIRD requires 4.8 times more time than SVFRD, this overhead is justified by the substantial reduction in false positives. This reduction significantly decreases the manual effort needed to examine each race alarm, which is a crucial advantage given the inherent complexity of such bugs.

Table 3.2: Comparison of SLIRD, SVFRD, RACERF, and GOBLINT on five large parallel programs. #TP and #FP are true positive and false positive, respectively. Time (secs) denotes running time.

Project	RACERF			GOBLINT			SVFRD			**SLIRD**		
	Report		Time	Report		Time	Report		Time	Report		Time
	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)
`Memcached`	1	15	5,183	0	8	6134	1	35	373	1	4	828
`OpenBLAS`	2	17	36,157	0	13	41,423	2	43	1,026	2	7	4,268
`Qt`	-	-	-	-	-	-	4	131	17,189	4	9	65,590
`MySQL`	-	-	-	-	-	-	2	119	16,287	2	11	123,206
`OpenCV`	-	-	-	-	-	-	3	156	23,153	3	17	146,876
Total	3	32	41,340	0	21	47,368	12	484	58,028	12	48	340,768

3.4.2 Ablation Analysis

This section presents an ablation study to evaluate the impact of the two core components of SLIRD: the slicing module and the cross-domain abstract interpretation.For this analysis, two variations of SLIRD is implemented: SLIRDA and CROSSRD. SLIRDA differs from SLIRD by replacing the cross domain abstract interpretation with as standard Andersen's points-to analysis to generate the points-to information for the race detection phase. CROSSRD, in contrast, omits the slicing module, performing the points-to analysis and race detection on the entire program.

Table 3.3 presents a comparison of SLIRD with its variations, SLIRDA and CROSSRD. Compared to SLIRDA, SLIRD reports fewer false positives on all five programs, achieving an average reduction of 77.6%. In terms of analysis time alone, SLIRD spends 1.6 times more time than SLIRDA. Furthermore, the cross-domain abstract interpretation method, despite its worst-case exponential complexity, performs comparably to the worst-case cubic Andersen's points-to analysis [7]. This efficiency can be attributed to the 3-limited context-sensitivity strategy [18] adopted by the abstract interpretation, which prevents the analysis tree from growing exponentially.

Table 3.3: A comparison of SLIRD with its two variations, SLIRDA and CROSSRD, across five large-scale parallel programs. #TP and #FP denote the number of true positives and false positives, respectively. All execution times are reported in seconds (s). Time is divided into slicing time (ST) and analysis time (AT), where AT represents the analysis time (including points-to, MHP, and lockset analyses, plus the final race detection step). Total time is the sum of ST and AT. Since CROSSRD does not perform slicing, its total time is equivalent to its analysis time (AT). A dash (-) signifies a timeout, where the execution exceeded a 5-day limit (432,000 seconds).

Project	SLIRDA			CROSSRD			**SLIRD**				
	Report		Time (s)	Report		Time (s)	Report		Time (s)		
	#TP	#FP	AT	#TP	#FP	Total/AT	#TP	#FP	ST	AT	Total
`Memcached`	1	23	238	1	4	13,273	1	4	131	697	828
`OpenBLAS`	2	29	1,398	2	6	92,435	2	7	314	3,954	4,268
`Qt`	4	38	28,357	-	-	-	4	9	5,317	60,273	65,590
`MySQL`	2	52	41,035	-	-	-	2	11	6,834	116,372	123,206
`OpenCV`	3	67	47,264	-	-	-	3	17	7,233	139,643	146,876
Total	12	209	118,337	3	10	105,708	12	48	192,829	320,939	340,768

Compared with CROSSRD, SLIRD completes its analysis of all five programs within the 5-days time limit, whereas CROSSRD only finishes the two relatively small ones, `Memcached` and `OpenBLAS`. On these two programs, SLIRD is 18.8 times faster than CROSSRD. Table 3.4 reports the number of functions before and after slicing. The results highlight the effectiveness of the slicing method, which reduces the analyzed codebases to as little as 6.2% of their original size. However, on `OpenBLAS`, SLIRD reports one additional false positive. This discrepancy arises because the slicing module excludes an allocation-related wrapper function. Consequently, the conservative pointers-to analysis generates spurious alias relationships, causing the subsequent race detection to report a false alarm. Despite this minor increase in false positives, the trade-off is acceptable given the significant efficiency gains that SLIRD provides.

Table 3.5 presents the phase-wise runtime statistics. The results indicates that

Table 3.4: Comparison of method counts before and after slicing across open-source projects. *#Method* and *#Method'* denote the number of methods before and after slicing, respectively.

Project	*#Method*	*#Method'*	Ratio (%)
`Memcached`	392	28	7.1
`OpenBLAS`	984	122	12.4
`Qt`	10,536	649	6.2
`MySQL`	17,823	1,141	6.4
`OpenCV`	9,357	891	9.5
Total	39,092	2,831	7.2

the slicing method significantly accelerates all the three phases under same alorithmm setting. However, since the pointer analysis adopted by SLIRDA and SLIRD are both inherently less scalable than the one adopted by SVFRD, therefore, even with the speedup brought by slicing, their overall analysis time still exceeds SVFRD. This trade-off is justified by the substantial improvement in precision. Moreover, the findings highlight the necessity of future research into more fine-grained slicing strategies to further accelerate points-to analysis, thereby enabling the practical application of fine-grained points-to techniques for enhanced precision.

Table 3.5: Runtime statistics of three analysis phases (PTA, MTA, RD) across four tools. PTA = Pointer Analysis, MTA = Multi-Thread Analysis (MHP + Lockset), RD = Race Detection. All times in seconds.

Project	SVFRD			SLIRDA			CROSSRD			**SLIRD**		
	PTA	MTA	RD	PTA	MTA	RD	PTA	MTA	RD	PTA	MTA	RD
`Memcached`	5	355	13	261	19	3	12,962	301	10	676	18	3
`OpenBLAS`	12	967	47	1,339	50	9	91,541	862	32	3,900	46	8
`Qt`	300	16,800	89	27,894	444	19	-	-	-	59,878	380	15
`MySQL`	315	15,903	69	40,363	641	31	-	-	-	115,737	612	23
`OpenCV`	268	22,782	103	46,633	597	34	-	-	-	139,037	575	31
Total	900	56,807	321	116,490	1,751	96	104,503	1,163	42	319,228	1,631	80

3.4.3 Case Study

This section presents a case study of OpenBLAS that illustrates how SLIRD eliminates a false positive data race alarm. Figure 3.6 illustrates the race site. In this code, exec_threads, a thread function created by exec_blas using pthread_create, invokes buffer_rw to write to a buffer pointed to by an element in the global blas_thread_buffer array. When exec_blas creates threads, it assigns a unique pos to each thread, preventing data races among the created threads.

```
1  static void * blas_thread_buffer[MAX_CPU_NUMBER];
2  void *exec_threads(void *queue){
3      blas_queue_t *blas_queue = (blas_queue_t *)queue;
4      int pos = blas_queue->pos;
5      buffer = blas_thread_buffer[pos];
6      buffer_rw(buffer);
7  }
8  void buffer_rw(void *buffer){
9      // write buffer;
10 }
11 int exec_blas(BLASLONG num, blas_queue_t *queue){
12     ...
13     for (int i = 0; i < num; i++){
14         phtread_create(..., exec_threads, &queue[i]);
15     }
16     ...
17 }
```

Figure 3.6: A code extracted from OpenBLAS illustrating a false positive data race.

In this case, SLIRDA reports a false alarm that SLIRD successfully eliminates. The slicing module extracts all relevant functions. It begins with the thread function exec_threads. It then transitively includes all callers from the call graph, such as exec_blas and its higher-level callers, in the slice. Because a callee of exec_threads, buffer_rw, accesses an object belonging to the transitive points-

to closure of the global `blas_thread_buffer` array, SLIRD correctly identifies a potential concurrent access and includes `buffer_rw` in the slice.

However, the Andersen's points-to analysis adopted by SLIRDA fails to distinguish between the different fields of `blas_thread_buffer`. It therefore informs the subsequent race detection client that the write operation at Line 6 could access the same buffer from different threads. In contrast, the more precise points-to information from the cross-domain abstract interpretation allows SLIRD to distinguish these fields. This distinction eliminates the possibility of a may-alias relationship of `buffer` pointer between different threads at Line 6, thereby resolving the false positive.

3.5 Conclusion

This study introduces SLIRD, a novel approach for improving the precision of data race detection while maintaining a reasonable performance overhead. SLIRD employs cross-domain abstract interpretation to provide more precise points-to information for subsequent race detection. To ensure efficiency, it incorporates a race detection-targeted slicing method that reduces the analysis scope. This slicing method is designed based on the two race patterns summarized in this study, therefore providing a minimal yet complete code slice for race detection. Empirical results show that SLIRD achieves an average precision improvement of 89.1%, underscoring the substantial benefits of more precise points-to analysis. Moreover, the slicing method significantly enhances the scalability of abstract interpretation, enabling successful analysis of three additional real-world programs with millions of lines of instructions. Importantly, this improvement is achieved with-

out compromising soundness. Although SLIRD incurs an average of 4.8× time consumption compared with SVFRD due to the inherent complexity of abstract interpretation, the precision gains justify the adoption of a more accurate points-to analysis. Furthermore, the speedup observed relative to CROSSRD underscores the effectiveness of slicing in extending complex points-to analyses to large-scale programs. Taken together, these findings motivate future research into more fine-grained slicing strategies.

Chapter 4

Enhancing Abstract Interpretation Precision on Recursion Handling

(The study presented in this chapter was published in a peer-reviewed conference: Jiawei Yang, Xiao Cheng, Bor-Yuh Evan Chang, Xiapu Luo, Yulei Sui. 2025. Taming and Dissecting Recursions through Interprocedural Weak Topological Ordering. 39th European Conference on Object-Oriented Programming. DOI: 10.4230/LIPIcs.ECOOP.2025.34)

4.1 Introduction

Static program analysis aims to infer a program's runtime properties by analyzing its code without executing it. Due to its ability to provide various valuable insights into the program properties of interest, such as alias information and loop invariants, it is widely applied in downstream software engineering tasks, including software security analysis, vulnerability detection, compiler optimization, and

performance analysis [17, 18, 34, 41].

Abstract interpretation [24, 23] offers a theoretical framework for static analysis. Unlike concrete execution, which operates with concrete values, abstract interpretation employs abstract domains to represent program abstract states. It derives program properties through computing an over-approximation of least-fixpoint on an equation system, as shown below:

$$a_1 = \Phi_1(a_1, a_2, \ldots, a_n),\ \ldots,\ a_n = \Phi_n(a_1, a_2, \ldots, a_n) \tag{4.1}$$

Equation 4.1 represents the abstract semantics of a program. It is defined by translating the program statement at each control point i along the control flow graph (CFG) into a semantic function Φ_i, where a_i denotes the abstract states at the control point i.

For programs with acyclic control flows, fixpoint computation typically converges after a few iterations over the CFG. However, the presence of cycles introduces cyclic dependencies between program states, complicating the analysis when working with abstract domains characterized by unbounded or very deep lattice heights [27, 89]. In such domains, such as the standard interval domain [16], the iterative computation may require a prohibitively large number of steps to reach a post-fixpoint. Even worse, termination is not guaranteed, as a post-fixpoint may not be reached within a finite number of steps. To facilitate fixpoint computation in these domains, widening techniques [22, 24, 11] compute a safe (upper) approximation of the least fixpoint (a post-fixpoint, hereafter referred to as a fixpoint for simplicity). However, widening operations must be judiciously applied only at cycle points, as spurious widening at acyclic locations introduces preci-

sion loss due to over-approximation, which cannot be recovered through subsequent narrowing [6]. Bourdoncle's algorithm [11] provides an effective solution through weak topological ordering (WTO) for intraprocedural analysis, which efficiently identifies program loops and their nested structures while requiring only one widening point per loop. The WTO framework further enables sophisticated interleaved widening and narrowing strategies [5], where narrowing compensates precision that may have been lost during widening by incorporating invariants such as loop bounds and branching constraints. These advantages have made WTO a standard method for handling intraprocedural loops in many current abstract interpreters [13, 26, 19, 18].

While state-of-the-art abstract interpreters effectively leverage Bourdoncle's algorithm for loop analysis, they face difficulties when determining WTO for recursive function calls. Several approaches have been proposed for taming recursions and interprocedural abstract interpretation. For non-recursive functions, function inlining enables precise interprocedural analysis. However, this approach becomes infeasible for recursive functions due to potentially infinite inlining sequences in unbounded recursion. To address this, some tools adopt strong assumptions, employing either under- or over-approximation approaches. For example, Astrée [43] completely excludes the analysis of unbounded recursion, while IKOS [13, 46] conservatively assigns top values ($\top$) to variables in recursive contexts, effectively bypassing the analysis of recursive functions. To improve precision, tools such as Clam [38, 19] and CSA [18] analyze recursive function bodies but introduce many spurious widening operations at function boundaries. Their effectiveness remains limited, as their WTO computations are confined to individual function scopes, preventing the creation of an effective interprocedural order for efficient fixpoint

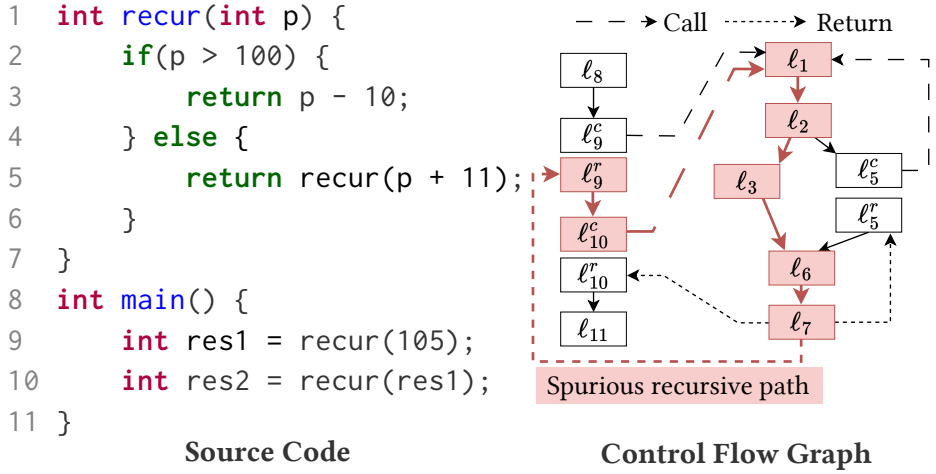


Figure 4.1: Spurious recursive path when analyzing McCarthy 91 function [53] (a library of mathematical theory of computation).

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 4.1 illustrates this challenge in the context of the McCarthy 91 func-

tion [53]. 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., $\ell_5^c \dashrightarrow \ell_1$, $\ell_9^c \dashrightarrow \ell_1$, and $\ell_{10}^c \dashrightarrow \ell_1$), while return edges link the callee's exit point to the corresponding return sites (e.g., $\ell_7 \dashrightarrow \ell_{10}^r$, $\ell_7 \dashrightarrow \ell_9^r$, and $\ell_7 \dashrightarrow \ell_5^r$). Note that, a spurious recursive path highlighted in red in the figure ($\ell_9^r \longrightarrow \ell_{10}^c \dashrightarrow \ell_1 \longrightarrow \ell_2 \longrightarrow \ell_3 \longrightarrow \ell_6 \longrightarrow \ell_7 \dashrightarrow \ell_9^r$) 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 [88] 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

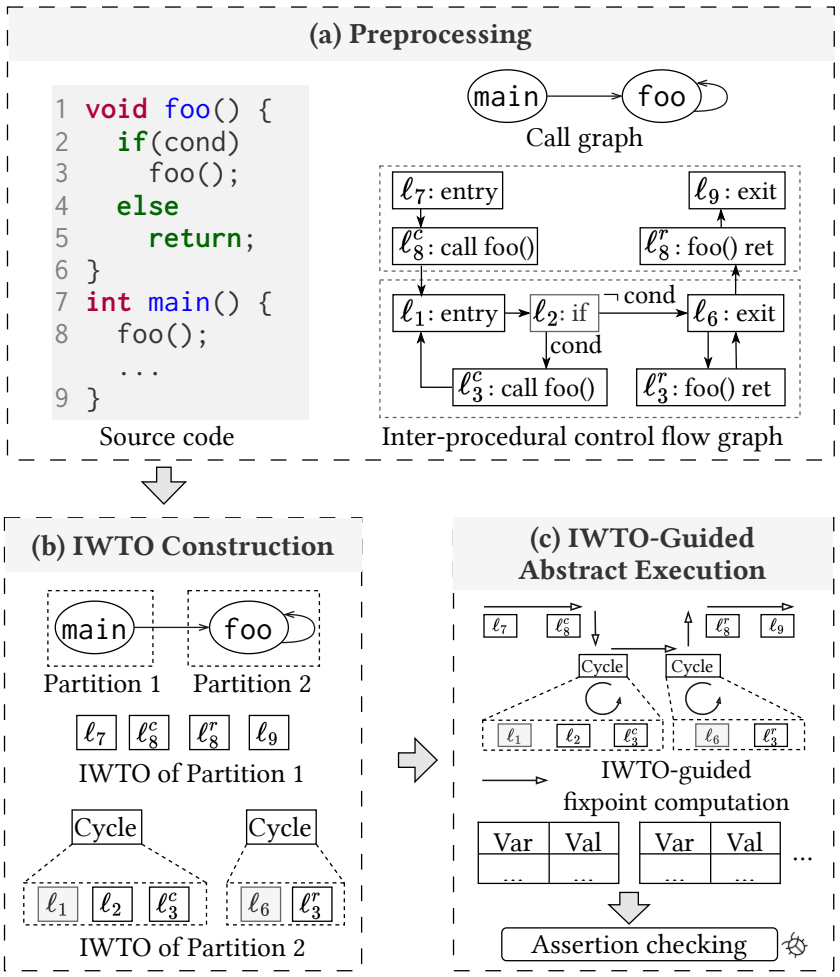


Figure 4.2: An overview of RecTopo framework.

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 4.2 illustrates the three-phase architecture of RecTopo, demonstrating how our framework systematically addresses the challenges of interprocedural analysis in the presence of recursions:

(a) Preprocessing: The framework begins by transforming the input program into LLVM's intermediate representation (LLVM-IR) [48]. 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 [88] 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 [88] 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.

Data Availability Statement. The implementation of RecTopo is open-source and available at [73].

4.2 Background

This section provides the foundational knowledge of the language on which our abstract interpretation is based (see §4.2.1), as well as discussions on abstract interpretation (see §4.2.2) and weak topological ordering (see §4.2.3).

Table 4.1: Analysis domains and LLVM-like statement set.

ℓ		$\in \mathcal{L}$	Statements
$\mathtt{c}, \mathtt{fld}$		$\in \mathcal{C}$	Constants
$\mathtt{p}, \mathtt{q}, \mathtt{idx}$		$\in \mathcal{S}$	Stack virtual registers
$\mathtt{g}$		$\in \mathcal{G}$	Global variables
$\mathtt{f}$		$\in \mathcal{F} \subseteq \mathcal{G}$	Program functions
$\mathtt{p}, \mathtt{q}, \mathtt{idx}, \mathtt{g}$		$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	Top-level variables/pointers
$\mathtt{o}$		$\in \mathcal{O}$	Abstract objects
$\mathtt{v}$		$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	Program variables
ℓ	$::=$	$\mathtt{p} = \mathtt{c} \mid \mathtt{p} = \&\mathtt{o}$	CONST/ADDR
	$\mid$	$\mathtt{p} = \&(\mathtt{q} \rightarrow \mathtt{fld}) \mid \mathtt{p} = \&\mathtt{q}[\mathtt{idx}]$	GEP
	$\mid$	$\mathtt{p} = *\mathtt{q} \mid *\mathtt{p} = \mathtt{q}$	LOAD/STORE
	$\mid$	$\mathtt{p} = \mathtt{q}$	COPY
	$\mid$	$\mathtt{p} = \phi(\mathtt{p}_1, \mathtt{p}_2, \dots)$	PHI
	$\mid$	$\mathtt{r} = \mathtt{p} \odot \mathtt{q}$	BINARY
	$\mid$	$\mathtt{p} = \neg\mathtt{q}$	UNARY
	$\mid$	$\mathtt{r} = \mathtt{f}(\mathtt{p}_1, \dots, \mathtt{p}_n)$	CALL
	$\mid$	$\mathtt{ret}_\mathtt{f}\ \mathtt{q}$	RETURN
$\odot \in \{$ +, -, $*$, /, %, <<, >>, <, >, &, &&, <=,>=, $\equiv$, $\sim$, $\mid$, $\wedge$ $\}$			

4.2.1 Language

Table 4.1 presents the analysis domains and LLVM-like statement set used in abstract interpretation. The set of program variables $\mathcal{V}$ is split into two disjoint subsets: $\mathcal{O}$, comprising *address-taken variables*, and $\mathcal{P}$, containing all *top-level variables*. Top-level variables $\mathtt{p}, \mathtt{q}, \mathtt{idx}, \mathtt{g} \in \mathcal{P}$ adhere to static single assignment (SSA) form and are subject to exactly one definition. Address-taken objects $\mathtt{o} \in \mathcal{O}$ can only be accessed through pointer dereferencing operations at LOAD/STORE instructions. Constant values are initially bound to top-level variables through CONS

instructions ($\texttt{p} = \texttt{c}$) that assign constants $\texttt{c}$. ADDR statements, denoted as $\texttt{p} = \&\texttt{o}$, allocate address-taken objects $\texttt{o}$, which may represent stack variables, global variables, or abstract heap objects. GEP enables structured access to composite objects: struct fields are accessed via constant offsets $\texttt{fld}$, while array elements use variable offsets $\texttt{idx}$. For field-sensitive analysis, RECTOPO employs a field-index-based strategy [9, 67], distinguishing struct fields through unique indices. COPY denotes a simple assignment of top-level variables/pointers. The PHI instruction, fundamental to SSA form, consolidates variable values at control flow merge points. BINARY and UNARY instructions represent arithmetic and bitwise operations, respectively. CALL and RETURN represents parameter passing and return value handling in function invocations.

4.2.2 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 5). 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 4 (Concrete and Abstract Domains)**.** Let $\mathbb{C} = 2^{\mathcal{V} \to S}$ be the concrete domain, where S represents all possible runtime values. The abstract domain $\mathbb{A}$ forms a lattice $\langle \mathbb{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$ with partial order $\sqsubseteq$. A Galois connection $\mathbb{C} \underset{\gamma}{\overset{\alpha}{\leftrightharpoons}} \mathbb{A}$ formalizes their relationship through abstraction function $\alpha : \mathbb{C} \to \mathbb{A}$ and concretization function $\gamma : \mathbb{A} \to \mathbb{C}$.

Definition 5 (Abstract Trace $\sigma \in \mathbb{L} \to \mathbb{A}$)**.** 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 $\sigma_{\underline{\ell_i}} = \sigma_{\underline{\ell_i}} \nabla \Phi_i(\sigma_{\underline{\ell_1}}, \sigma_{\underline{\ell_2}}, \ldots, \sigma_{\underline{\ell_n}})$ instead of $\sigma_{\underline{\ell_i}} = \Phi_i(\sigma_{\underline{\ell_1}}, \sigma_{\underline{\ell_2}}, \ldots, \sigma_{\underline{\ell_n}})$ 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 $a_1 \sqsubseteq a_2 \sqsubseteq ... \sqsubseteq a_m$, the increasing chain $a'_1 \sqsubseteq a'_2 \sqsubseteq ... \sqsubseteq a'_m$ 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 $\mathcal{I} = \{[l, u] | l, u \in \mathbb{Z}\}$ 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 [13, 24, 6, 45], 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 $a_1 \sqsupseteq a_2 \sqsupseteq ... \sqsupseteq a_m$, the decreasing chain $a'_1 \sqsupseteq a'_2 \sqsupseteq ... \sqsupseteq a'_m$ 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: $[l_0, u_0] \Delta [l_1, u_1] = [\text{if } l_0 = -\infty \text{ then } l_1 \text{ else } l_0, \text{if } u_0 = +\infty \text{ then } u_1 \text{ else } u_0]$.

4.2.3 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 6 (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 1. "$(\underline{\underline{1}}\ (\underline{\underline{2}}\ 3))$" 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, "$(\underline{2}\ 3)$" is a sub-component of "$(\underline{1}\ (\underline{2}\ 3))$", thus $\omega(3) = \{1, 2\}$ and $\delta(3) = 2$.

Definition 7 (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 $u \to v \in E$, $u \prec v \wedge v \notin \omega(u)$ or $v \preceq u \wedge v \in \omega(u)$. If an edge $u \to v$ satisfies $v \preceq u \wedge v \in \omega(u)$, $u \to v$ is called a *back edge*.

Example 2. One possible WTO of the intraprocedural control flow graph in Figure 4.3 is "$1\ (\underline{2}\ (\underline{3}\ 4)\ 5)\ 6$". 2 and 3 are the WTO heads and $5 \to 2$ and $4 \to 3$ are back edges. It is clear that WTO precisely captures the nested loop structures within the graph. The inside loop is captured by "$(\underline{3}\ 4)$" while the outside loop is "$(\underline{2}\ (\underline{3}\ 4)\ 5)$".

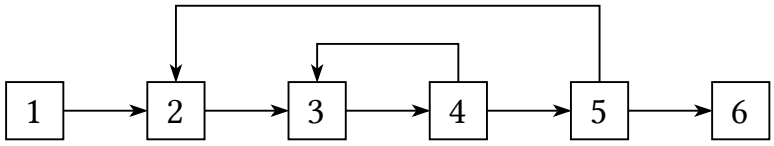


Figure 4.3: An example of control flow graph.

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 [11]. 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 [35], which utilizes Tarjan's algorithm [88] to identify all strongly connected compo-

nents (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 [13, 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.

4.3 Motivating Example

Figure 4.4 illustrates how RecTopo precisely determines the interprocedural fixpoint computation order and widening points by revisiting the example in Figure 4.1. We use the integer interval domain [16] to reason about its behaviors. The state-of-the-art analyzers, IKOS [13], Clam [19, 38] 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 4.4(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 ($\ell_9^r \longrightarrow \ell_{10}^c \dashrightarrow \ell_1 \longrightarrow \ell_2 \longrightarrow \ell_3 \longrightarrow \ell_6 \longrightarrow \ell_7 \dashrightarrow \ell_9^r$) is a spurious

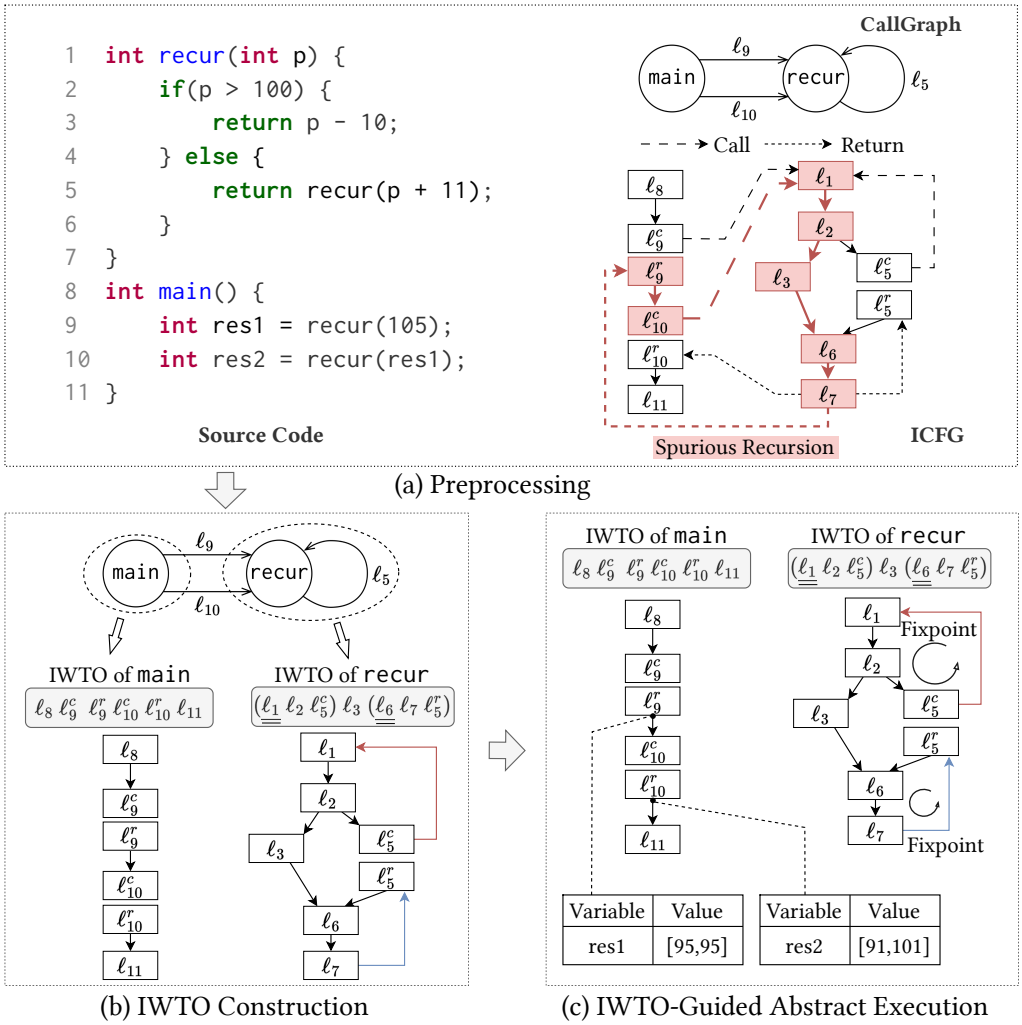


Figure 4.4: A motivating example by revisiting the example in Figure 4.1. It provides a detailed illustration of the phases of our framework, as shown in Figure 4.2.

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 4.4(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 "$(\underline{\underline{\ell_1}}\ \ell_2\ \ell_5^c)\ \ell_3\ (\underline{\underline{\ell_6}}\ \ell_7\ \ell_5^r)$" (where $\underline{\underline{\ell_1}}$ and $\underline{\underline{\ell_6}}$ are WTO heads). This analysis contrasts with approaches like IKOS [13], which typically omit the function body in recursive analyses, or Clam [19, 38] 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 4.4(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 $(\underline{\underline{\ell_1}}\ \ell_2\ \ell_5^c)$, 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 $(\underline{\underline{\ell_6}}\ \ell_7\ \ell_5^r)$, 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 [13, 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^r_{10}, the value of `res2` is determined to be $[91, 101]$, demonstrating consistently greater precision than IKOS, Clam and CSA.

4.4 Approach

In this section, we introduce our approach for constructing interprocedural weak topological ordering in §4.4.1, followed by the algorithm for IWTO-guided abstract interpretation in §4.4.2. Additionally, we provide the proof of termination for our algorithm in §4.4.3.

4.4.1 IWTO Construction

The construction of IWTO consists of two stages. In the first stage (see §4.4.1.1), we perform program partitioning to divide the program into several function partitions (Definition 8). The second stage (see §4.4.1.2) involves constructing the IWTO on the ICFG for each resulting function partition from the first stage.

4.4.1.1 Program Partitioning

This stage aims to decompose the input whole program into several function partitions (see Definition 8) and identify all function partition entries (see Definition 9). 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.

Definition 8 (Function Partition)**.** A *function partition*, denoted as `FPar`, consists of a set of functions that are directly or indirectly reachable from each other on the call graph.

We apply Tarjan's algorithm [88] to the pre-built call graph of the analyzed program, identifying all strongly connected components (SCCs). Each SCC on the call graph corresponds to a distinct function partition. Non-recursive functions, being single-node SCCs, each forms an individual partition. For recursive functions, those within the same calling chain are aggregated into a single partition. Note that if a recursive function calls a non-recursive function, they are placed in separate partitions because the non-recursive function does not call back, meaning that they do not meet the criteria of mutual reachability required for placement in the same partition (Definition 8).

Definition 9 (Function Partition Entry)**.** For a function partition `FPar`, a function $\texttt{f} \in \texttt{FPar}$ is considered a *function partition entry* if and only if there exists a function $\texttt{f}' \notin \texttt{FPar}$ such that $\texttt{f}'$ calls `f`. In other words, `f` is an entry point to the partition `FPar` if it is invoked by at least one function outside of the partition.

Function Partition Entries Identification. For each function `f` within a function partition `FPar`, we examine the predecessors of `f` on the call graph. If any predecessor $\texttt{f}'$ is found not to belong to `FPar`, then `f` is designated as a partition entry for that partition.

Example 3. We illustrate the process of program partitioning with an example and its call graph in Figure 4.5. This example includes a mutual recursion between the functions `recur1` and `recur2`, invoked by `main` at locations ℓ_2 and ℓ_3, respectively. Additionally, the function `id` is called by `recur2` at location ℓ_{15}. Applying Tarjan's

```
1  int main() {
2      recur1(20);
3      recur2(15);
4  }
5  int recur1(int i) {
6      if(i > 10)
7          return recur2(i - 1);
8      else
9          return i;
10 }
11 int recur2(int j) {
12     if(j > 10)
13         return recur1(j - 1);
14     else
15         return id(j);
16 }
17 // identity function
18 int id(int k) {
19     return k;
20 }
```

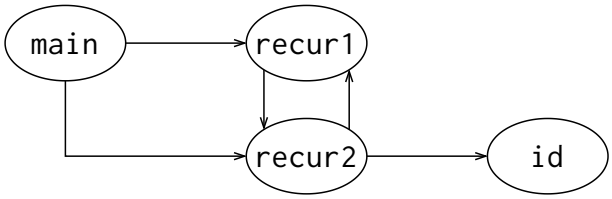


Figure 4.5: A recursion example and its call graph.

algorithm to this call graph results in three SCCs: {main}, {recur1, recur2}, and {id}. Each SCC corresponds to a distinct function partition. For the function partition {recur1, recur2}, both functions are identified as partition entries, as they are separately called by main at different locations.

4.4.1.2 IWTO Construction

For each function partition entry, we establish an interprocedural weak topological ordering (IWTO). As shown in Algorithm 3, the main function constructIMap (Lines 1-5) processes all function partitions of the analyzed program, alongside EMap, which maps each function partition to its function partition entries and aims to produce IMap, 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 FPar and its associated entries FParEns. constructIWTO first extracts the subgraph corresponding to FPar on the ICFG, and then

Algorithm 3: IWTO construction algorithm

```
Input: G(V, E): ICFG of the program
       EMap: map from function partition to its partition entries
Output: IMap: map from partition entry to IWTO
1  Function constructIMap(EMap, G):
2      IMap := {};
3      foreach (FPar → FParEns) ∈ EMap do
4          constructIWTO(FPar, FParEns, G, IMap);
5      return IMap;
6  Function constructIWTO(FPar, FParEns, G, IMap):
7      G_FPar(V_FPar, E_FPar) := ICFGPartition(G, FPar);        // Step 1
8      foreach v ∈ V_FPar do                                    // Step 2
9          if v is a non-recursive call site then
10             E_FPar := E_FPar ∪ (v, v.returnSite());
11     foreach entry ∈ FParEns do                               // Step 3
12         entryICFGNode := Entry ICFG Node of entry;
13         IMap[entry] := Bourdoncle(G_FPar, entryICFGNode);
```

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 `IMap`. The detailed steps of `constructIWTO` are outlined as follows:

Step 1: ICFG Partitioning. Given a function partition `FPar`, this step (Line 7) aims to extract the induced sub-ICFG corresponding to `FPar` 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 `FPar`.

Definition 10 (Recursive and Non-Recursive Call/Return Site)**.** A call site $\ell^c\langle \texttt{caller}, \texttt{callee}\rangle$ in the ICFG is defined as a *recursive call site* iff. `caller` and `callee` are in the same function partion. 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

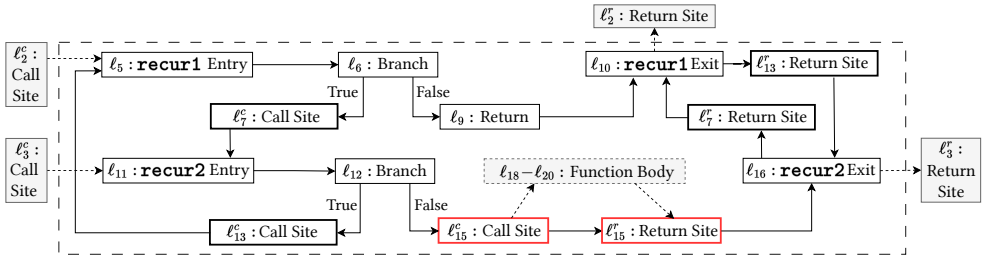


Figure 4.6: The sub-ICFG of function partition `recur1` and `recur2` by revisiting the example in Figure 4.5.

the return from the callee function.

Example 4. Figure 4.6 shows the sub-ICFG of function partition {`recur1`, `recur2`} for the code example in Figure 4.5. 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 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 [35] 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 3. It then constructs an IWTO across the entire sub-ICFG, incorporating its interprocedural edges.

Example 5. Let us revisit the example in Figure 4.5 and illustrate the three steps in Algorithm 3 used to construct the IWTO of the function partition $\{\texttt{recur1}, \texttt{recur2}\}$. In Step 1, we obtain an induced sub-ICFG shown in Figure 4.6, 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: "$(\underline{\underline{\ell_5}}\ \ell_6\ \ell_7^c\ \ell_{11}\ \ell_{12}\ \ell_{13}^c)\ \ell_9\ \ell_{15}^c\ \ell_{15}^r\ (\underline{\underline{\ell_{10}}}\ \ell_{13}^r\ \ell_{16}\ \ell_7^r)$". The IWTO starting from ℓ_{11} is outlined as: "$(\underline{\underline{\ell_{11}}}\ \ell_{12}\ \ell_{13}^c\ \ell_5\ \ell_6\ \ell_7^c)\ \ell_9\ \ell_{15}^c\ \ell_{15}^r\ (\underline{\underline{\ell_{16}}}\ \ell_7^r\ \ell_{10}\ \ell_{13}^r)$".

4.4.2 IWTO-Guided Abstract Interpretation

This section first presents the overall algorithm for fixpoint computation in §4.4.2.1, followed by an example set of feasible node-level interpretation rules that can be applied during the fixpoint computation in §4.4.2.2.

Algorithm 4: 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⟨caller, 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);
```

4.4.2.1 Abstract Interpretation Algorithm

Algorithm 4 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 §4.4.1.2, and the program's entry function, `root` (e.g., the `main` function). Starting from `root`, which is a function partition entry (Definition 9), 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 (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 [5] 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 4, 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 (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 4. It employs a while loop to continuously process the IWTO elements constituting the cycle. As mentioned in §4.2.3, 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 [5]. 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.

4.4.2.2 Node-Level Interpretation Rules

Figure 4.7 demonstrates an example set of rules for handling each IWTO node, which can be applied at Line 11 in Algorithm 4. 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 6. Figure 4.8 illustrates this selective aggregation using the example

in Figure 4.6, 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 $\texttt{handlePartition}(\ell_3^c\langle\texttt{main}, \texttt{recur2}\rangle)$ to analyze the function partition $\{\texttt{recur1}, \texttt{recur2}\}$ from partition entry `recur2`. When deriving the pre-abstract state of ℓ_{11}, as shown in Figure 4.8a, both $\sigma_{\underline{\ell_7^c}}$ and $\sigma_{\underline{\ell_3^c}}$ 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 4 at Line 10. Conversely, as shown in Figure 4.8b, when deriving the pre-abstract state of ℓ_5, $\sigma_{\underline{\ell_2^c}}$ is blocked because ℓ_2^c does not match `pEntryCallsite`, ℓ_3^c, whereas $\sigma_{\underline{\ell_{13}^c}}$ propagates due to the recursive nature of ℓ_{13}^c.

The RETFLOW rule facilitates the propagation of abstract states from a callee function's exit node to the corresponding return site. The SEQUENCE rule refines abstract values based on control flow edge conditions. Figure 4.9 demonstrates this refinement process for deriving the pre-abstract state of ℓ_{13}^c. Since the incoming edge from ℓ_{12} requires condition `i > 10` to hold, we refine the abstract value of `i` in $\sigma_{\underline{\ell_{12}}}$ by computing its meet with $[11, \infty]$.

The post-abstract state $\sigma_{\underline{\ell}}$ of node ℓ is derived from its pre-abstract state $\sigma_{\overline{\ell}}$ 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, for a BINARY statement $\ell : \texttt{r} = \texttt{p} + \texttt{q}$ and $\sigma_{\overline{\ell}}(\texttt{p}) := \langle[1, 3], \top\rangle, \sigma_{\overline{\ell}}(\texttt{q}) := \langle[4, 7], \top\rangle$, we derive $\sigma_{\underline{\ell}}(\texttt{r}) := \langle[5, 10], \top\rangle$. 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 inter-

val 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 $\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))$ with an offset $\mathtt{j} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{idx}))$ to derive the conservative address value for the left-hand-side variable. As for LOAD, we join the abstract values $\sigma_{\overline{\ell}}(\mathtt{o})$ of all possible addresses $\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{q}))$ to form the conservative abstract value of the left-hand-side variable. Lastly, for STORE, we update the abstract value of every possible address $\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))$. Particularly, when there are multiple addresses $\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))$, we join the abstract value $\sigma_{\overline{\ell}}(\mathtt{q})$ with each existing abstract value $\sigma_{\overline{\ell}}(\mathtt{o})$ to form o's new abstract value conservatively. In contrast, when there is only one $\mathtt{o} \in \gamma(\sigma_{\overline{\ell}}(\mathtt{p}))$, it is safe to directly use $\sigma_{\overline{\ell}}(\mathtt{q})$ as o's new abstract value, because the store must occur at o.

Example 7. We revisit the example in Figure 4.5 to illustrate the IWTO-guided abstract interpretation. It starts with $\mathtt{handlePartition}(\ell_0\langle \_, \mathtt{main}\rangle)$. 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. $\mathtt{handlePartition}(\ell_3^c\langle \mathtt{main}, \mathtt{recur2}\rangle)$ follows the IWTO of function partition {recur1, recur2}, which is outlined as "$(\underline{\underline{\ell_{11}}}\ \ell_{12}\ \ell_{13}^c\ \ell_5\ \ell_6\ \ell_7^c)\ \ell_9\ \ell_{15}^c\ \ell_{15}^r\ (\underline{\underline{\ell_{16}}}\ \ell_7^r\ \ell_{10}\ \ell_{13}^r)$", to perform abstract interpretation. The first element of the IWTO, "$(\underline{\underline{\ell_{11}}}\ \ell_{12}\ \ell_{13}^c\ \ell_5\ \ell_6\ \ell_7^c)$", 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 $\sigma_{\overline{\ell_{11}}}(\mathtt{j})$, 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 §4.2.2). The iteration then proceeds to the narrowing stage, during which $\sigma_{\overline{\ell_{11}}}(\mathtt{j})$ 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.

4.4.3 Termination Analysis

This section aims to prove that Algorithm 4 is guaranteed to terminate, even in the presence of nested recursions and loops in the input program. Algorithm 4 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 4, we introduce the concept of *recursion tree*.

Definition 11 (Recursion Tree)**.** The recursion tree, denoted as $T = (N, E, r)$, 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 8. Figure 4.10 demonstrates the recursion tree of Algorithm 4 as it executes on the program presented in Figure 4.5. The root node indicates that the recursion of Algorithm 4 begins with `handlePartition`$(\ell_0^c\langle \_, \texttt{main}\rangle)$, where $\ell_0^c\langle \_, \texttt{main}\rangle$ 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 4 contains a finite number of nodes for any input.

Lemma 1. The recursion tree of Algorithm 4 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` (reffered as S nodes). These are the only types of nodes present on the path, as we have inlined all `handleNode` invokations for simplicity.

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

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

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*($\texttt{callee}_i$) = *FP*($\texttt{caller}_{i+1}$). Consequently, for all $i < j$, $\texttt{caller}_j$ is reachable from $\texttt{caller}_i$ on the call graph. We aim to prove the following sub-conclusions. **Sub-conclusion 1**: For all $i \leq j$, *FP*($\texttt{caller}_i$) $\neq$ *FP*($\texttt{callee}_j$). Suppose this were not the case. Then, there would exists $i \leq j$ such that *FP*($\texttt{caller}_i$) = *FP*($\texttt{callee}_j$), implying that $\texttt{caller}_i$ is reachable from $\texttt{callee}_j$. Furthermore, it follows that $\texttt{caller}_j$ would also be reachable from $\texttt{callee}_j$, which contradicts the definition of $\texttt{v}_j\langle\texttt{caller}_j, \texttt{callee}_j\rangle$ as a non-recursive call site (according to Definition 10). **Sub-conclusion 2**: For all $i < j$, *FP*($\texttt{caller}_i$) $\neq$ *FP*($\texttt{caller}_j$). This follows directly from Sub-conclusion 1 and *FP*($\texttt{callee}_i$) = *FP*($\texttt{caller}_{i+1}$). If *FP*($\texttt{caller}_i$) = *FP*($\texttt{caller}_j$) for some $i < j$, it would contradict Sub-conclusion 1. **Sub-conclusion 3**: $m \leq |\textit{FPars}|$, where $|\textit{FPars}|$ 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*($\texttt{caller}_i$) = *FP*($\texttt{caller}_j$), which would contradict Sub-conclusion 2.

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

$$S_1(\texttt{cycle}_1), \ldots, S_n(\texttt{cycle}_n).$$

For this sequence, the inequality $n < \texttt{cycle}_1\texttt{.depth}$ is established, where $\texttt{cycle}_1\texttt{.depth}$ denotes the depth of the $\texttt{cycle}_1$. This holds because $\texttt{cycle}_{i+1}$ is a sub-IWTO component of $\texttt{cycle}_i$. Furthermore, the number of continuous S subsequences is limited to length of the longest P subsequence on the path, which, as previously concluded, is $|\textit{FPars}|$. Consequently, the total number of S nodes on the path is limited to $|\textit{FPars}| \times \textit{maxCycleDepth}$, 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 $|\mathit{FPars}| \times (\mathit{maxWTOdepth} + 1)$. □

Lemma 2. The recursion tree of Algorithm 4 for any input has finite degree.

Proof. To establish this, we consider two cases. **Case 1:** For a $P(\ell^c\langle\texttt{caller}, \texttt{callee}\rangle)$ node, its degree is limited by the number of elements in `callee`'s corresponding IWTO. **Case 2:** For a $S(\texttt{cycle})$ node, its child nodes represent the invoked subroutines within the double-nested loop outlined in Algorithm 4 from Lines 17 to 32. The outer loop is guaranteed to terminate within h rounds [11], 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 1 (Termination)**.** Algorithm 4 is guaranteed to terminate.

Proof. By Lemmas 1 and 2, 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 4. □

4.5 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 [13], Clam [19, 38] 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:

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?

4.5.1 Datasets and Implementation

Datasets. We evaluate RecTopo using (1) a benchmark composed of 8312 programs from NIST [59], 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 4.2. These projects include `paste` [14] (file merger), `md5sum` [14] (file verifier), `YAJL` [97] (JSON parser), `8cc` [1] (C compiler), `libplist` [50] (Property List file handler), `MP4v2` [56] (MP4 file library), `RIOT` [74] (IoT operating system), `darknet`[32] (neural network framework), `tmux` [92] (terminal multiplexer) and `Teeworlds` [90] (online multiplayer game).

Table 4.2: 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*	$\|V\|$	$\|E\|$
`paste`	8,416	53	758	510	9,395	9,922
`md5sum`	11,483	63	881	606	12,494	13,064
`YAJL`	20,592	151	561	208	9,253	9,922
`8cc`	30,609	480	2,349	3,267	15,984	25,294
`libplist`	35,324	342	1,743	1,635	13,697	20,978
`MP4v2`	39,178	601	610	1,991	15,595	16,733
`RIOT`	54,597	579	1,614	951	20,176	20,843
`darknet`	210,117	1,004	9,861	2,614	110,424	140,213
`tmux`	446,626	1,967	22,369	3,879	162,879	178,924
`Teeworlds`	529,737	2,306	20,701	5,754	251,356	246,029
Total	1,386,679	7,546	61,447	21,415	621,253	681,922

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 4.4.1 (Algorithm 3). The abstract value representation is implemented using Z3 expressions [55]. We utilize the interleaved widening and narrowing strategy as described in [5] 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, we directly use their open-source implementations and their

Table 4.3: 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**

default settings (e.g., interval analysis) for detecting buffer overflows.

4.5.2 RQ1: NIST Benchmark

Table 4.3 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

Table 4.4: 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	Report		Time	Report		Time	Report		Time
	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)
`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

external API modeling [18]. Here, we focus on analyzing the precision improvements in handling recursion through two representative code scenarios shown in Figure 4.11. A key contributing factor is RecTopo's construction of IWTO for recursive functions, which effectively guides the narrowing process by leveraging program path conditions. As demonstrated in Figure 4.11(a), the program accesses the `buf` buffer at ℓ_{14} with an offset `idx`, which is obtained from the return value of the `mc91(100)` function call at ℓ_{13}. RecTopo precisely determines `mc91`'s return value in the range $[91, 101]$. Consequently, it concludes that the buffer access at ℓ_{14} is safe. In contrast, all our baseline analyses report an overflow alarm here, incorrectly indicating that the value of `mc91(100)` could be anywhere within $[-\infty, \infty]$.

Another reason is that the IWTO construction approach effectively eliminates spurious recursions (widening points) arising from multiple call sites of the same function, such as the double call to `adder` in Figure 4.11(b). RecTopo accurately differentiates between these call sites, preventing misclassification of pro-

gram points, $\ell_{12}, \ell_2 - \ell_4, \ell_{10}, \ell_{11}$, as part of a spurious recursion. This precision prevents RecTopo from widening the value of `sum` at the `adder` entry point (ℓ_2) to $[0, \infty]$. Consequently, RecTopo derives the precise value of `sum` at ℓ_{14} as $[3, 3]$, thereby eliminating a false overflow alarm.

False Positives of RecTopo. We further analyze RecTopo's false positives and identify two primary causes, as demonstrated in the examples in Figure 4.12. First, RecTopo compromises precision for efficiency in scenarios involving complex path conditions. A case in point is its failure to accurately interpret the path condition at ℓ_2 in Figure 4.12(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 4.12(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.

4.5.3 RQ2: Real-World Projects

Table 4.4 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 interleaved widening and narrowing operations on recursions.

Case Study. Figure 4.13 illustrates the patch [2] and call graph for a real-world buffer overflow identified by RecTopo in `libplist` [50], a portable C library to handle Property List files [70]. 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 4.13. 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 4.14a 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 4.14b depicts the aggregate depth of nodes (as defined in §4.2.3) 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 [11], this suggests that the overall time complexity for stabilizing IWTO cycles scales approximately linearly with program size. Figure 4.14c presents the number of nodes within IWTO cycles per program, while Figure 4.14d 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.

Table 4.5: Comparing RecTopo with its two variants, RecTopo-NR and RecTopo-GL (described in §4.5.4), 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	Report		Time	Report		Time
	#TP	#FP	(secs)	#TP	#FP	(secs)	#TP	#FP	(secs)
`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
Total	75	1319	6505	75	4858	9068	**75**	**39**	7905

4.5.4 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 4.5 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 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× longer compared to RecTopo.

The reduced precision in RecTopo-GL can be primarily attributed to its redundant widening operations. As depicted in Figure 4.15a, 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 4.15b 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.

4.6 Conclusion

This study 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 baselines tools on ten open-source projects.

$$[\textsc{CallFlow}]\ \frac{\ell@\texttt{caller} \hookrightarrow \ell'@\texttt{callee}}{\sigma_{\overline{\ell'}} \sqsupseteq \texttt{filterCall}(\sigma_{\underline{\ell}}, \ell)}$$

$$[\textsc{RetFlow}]\ \frac{\ell@\texttt{caller} \hookleftarrow \ell'@\texttt{callee}}{\sigma_{\overline{\ell}} \sqsupseteq \sigma_{\underline{\ell'}}}$$

$$[\textsc{Sequence}]\ \frac{\ell \overset{\texttt{cond}}{\hookrightarrow} \ell'}{\sigma_{\overline{\ell'}} \sqsupseteq \texttt{refineSeq}(\sigma_{\underline{\ell}}, \texttt{cond})} \qquad [\textsc{Cons}]\ \frac{\ell : \texttt{p} = \texttt{c}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{p} \mapsto \langle \alpha(\{\texttt{c}\}), \top \rangle]}$$

$$[\textsc{Copy}]\ \frac{\ell : \texttt{p} = \texttt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{p} \mapsto \sigma_{\overline{\ell}}(\texttt{q})]} \qquad [\textsc{Phi}]\ \frac{\ell : \texttt{r} = \phi(\texttt{p}_1, \texttt{p}_2, ..., \texttt{p}_\texttt{n})}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{r} \mapsto \bigsqcup_{\texttt{i}=1}^{\texttt{n}} \sigma_{\overline{\ell}}(\texttt{p}_\texttt{i})]}$$

$$[\textsc{Unary}]\ \frac{\ell : \texttt{p} = \neg\texttt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{p} \mapsto \neg^{\#}\sigma_{\overline{\ell}}(\texttt{q})]} \qquad [\textsc{Binary}]\ \frac{\ell : \texttt{r} = \texttt{p} \odot \texttt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{r} \mapsto \sigma_{\overline{\ell}}(\texttt{p}) \odot^{\#} \sigma_{\overline{\ell}}(\texttt{q})]}$$

$$[\textsc{Addr}]\ \frac{\ell : \texttt{p} = \&\texttt{o}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{p} \mapsto \langle \top, \{o_i\} \rangle]}$$

$$[\textsc{Gep}]\ \frac{\ell : \texttt{p} = \&(\texttt{q} \rightarrow \texttt{idx}) \mid \texttt{p} = \&\texttt{q}[\texttt{idx}]}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{p} \mapsto \bigsqcup_{\texttt{o} \in \gamma(\sigma_{\overline{\ell}}(\texttt{q}))} \bigsqcup_{j \in \gamma(\sigma_{\overline{\ell}}(\texttt{idx}))} \langle \top, \{\texttt{o.fld}_j\} \rangle]}$$

$$[\textsc{Load}]\ \frac{\ell : \texttt{p} = *\texttt{q}}{\sigma_{\underline{\ell}} := \sigma_{\overline{\ell}}[\texttt{p} \mapsto \bigsqcup_{o \in \gamma(\sigma_{\overline{\ell}}(\texttt{q}))} \sigma_{\overline{\ell}}(o)]}$$

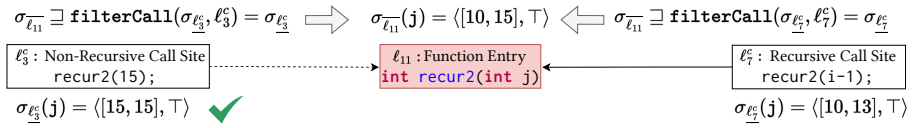
$$[\textsc{Store}]\ \frac{\ell : *\texttt{p} = \texttt{q}}{\sigma_{\underline{\ell}} := (\{o \mapsto \sigma_{\overline{\ell}}(\texttt{q}) | o \in \gamma(\sigma_{\overline{\ell}}(\texttt{p}))\} \sqcup \sigma_{\overline{\ell}} \setminus \mathsf{kill}(\ell))}$$

$$\mathsf{filterCall}(\sigma_{\underline{\ell}}, \ell) := \begin{cases} \sigma_{\underline{\ell}} & \text{if } \ell \text{ is recursive call site} \vee \ell = \texttt{pEntryCallsite} \\ \bot & \text{otherwise} \end{cases}$$

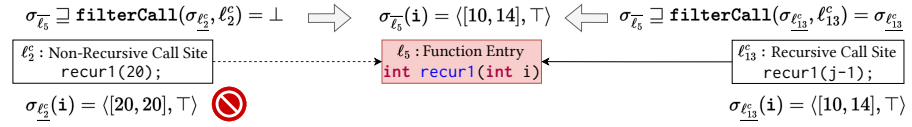
$$\mathsf{refineSeq}(\sigma_{\underline{\ell}}, \texttt{cond}) := \sigma_{\underline{\ell}}[\texttt{p}_1 \mapsto \sigma_{\underline{\ell}}(\texttt{p}_1) \sqcap \alpha(\texttt{cond}(\texttt{p}_1)), \ldots, \texttt{p}_\texttt{n} \mapsto \sigma_{\underline{\ell}}(\texttt{p}_\texttt{n}) \sqcap \alpha(\texttt{cond}(\texttt{p}_\texttt{n}))]$$

$$\mathsf{kill}(\ell : *\texttt{p} = \texttt{q}) := \begin{cases} \{\texttt{o} \mapsto \_ \mid \texttt{o} \in \gamma(\sigma_{\overline{\ell}}(\texttt{p}))\} & \text{if } \sigma_{\overline{\ell}}(\texttt{p}) = \langle \top, \{\texttt{o}\} \rangle \wedge \texttt{o} \text{ is singleton} \\ \{\texttt{o} \mapsto \_ \mid \texttt{o} \in \mathcal{O}\} & \text{if } \sigma_{\overline{\ell}}(\texttt{p}) = \langle \top, \varnothing \rangle \\ \varnothing & \text{otherwise} \end{cases}$$

Figure 4.7: Rules for interpreting each node during abstract interpretation (Line 11 in Algorithm 4). $\ell@\texttt{caller} \hookrightarrow \ell'@\texttt{callee}$ and $\ell@\texttt{caller} \hookleftarrow \ell'@\texttt{callee}$ 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`. $\ell \overset{\texttt{cond}}{\hookrightarrow} \ell'$ denotes an intraprocedural control flow with path condition `cond`, whose value is null for unconditional flows. $a \sqsupseteq b$ represents $a := a \sqcup b$. $\texttt{f}^{\#}$ is the abstract operator concerning the concrete operator `f`. $\alpha(\texttt{cond}(\texttt{p}_\texttt{i}))$ abstracts the concrete values of $\texttt{p}_\texttt{i}$ restricted by sequence condition `cond` based on Definition 4. $\texttt{o.fld}_\texttt{j}$ represents the j^{th} field of the base object `o`.



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



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

Figure 4.8: Illustration of the CALLFLOW rule for deriving pre-abstract state of ℓ_{11} and ℓ_5 by revisiting the example in Figure 4.6.

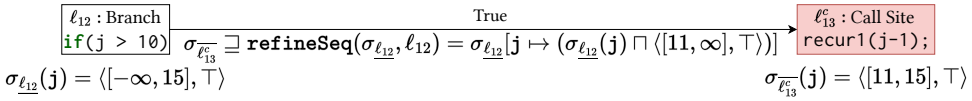


Figure 4.9: Illustration of the SEQUENCE rule for deriving the pre-abstract state $\sigma_{\overline{\ell_{13}^c}}$ by revisiting the example in Figure 4.6.

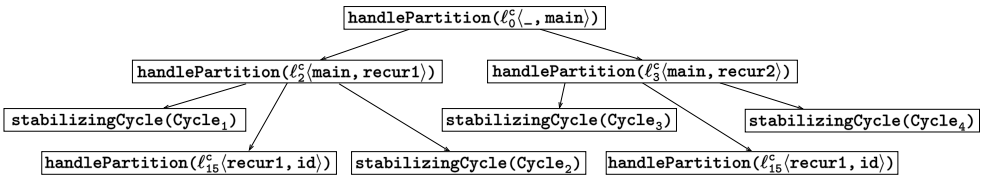


Figure 4.10: Recursion tree of algorithm 4 executing on the program presented in Figure 4.5

```c
int mc91(int p) {
    if(p > 100) {
        return p - 10;
    } else {
        int p1 = p + 11;
        int p2 = mc91(p1);
        int res = mc91(p2);
        return res;
    }
}
int main() {
    char buf[200];
    int idx = mc91(100);
    buf[idx] = 0;
}
```

(a) Recursive function.

```c
int sum = 0;
void adder(int num) {
    sum += num;
}
int main() {
    char buf[10];

    // call adder twice
    int m = 1;
    adder(m);
    int n = 2;
    adder(n);

    buf[sum] = 0;
}
```

(b) Multiple call sites.

Figure 4.11: False buffer overflows eliminated by RecTopo.

```c
int modNb(int n) {
    if (n - 9 >= 0)
        return n;
    else
        return modNb(n + 1);
}
int main() {
    char buf[10];
    int idx = modNb(0);
    buf[idx] = 0;
}
```

(a) Complex path condition.

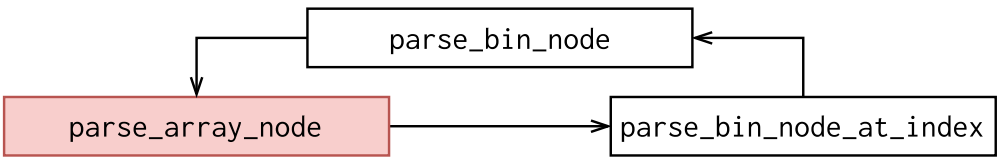
```c
int sum(int n, int m) {
    if (n <= 0)
        return m + n;
    else
        return sum(n - 1,m + 1);
}
int main() {
    char buf[10];
    int idx = sum(9, 0);
    buf[idx] = 0;
}
```

(b) Variable correlations.

Figure 4.12: False positives of RecTopo.

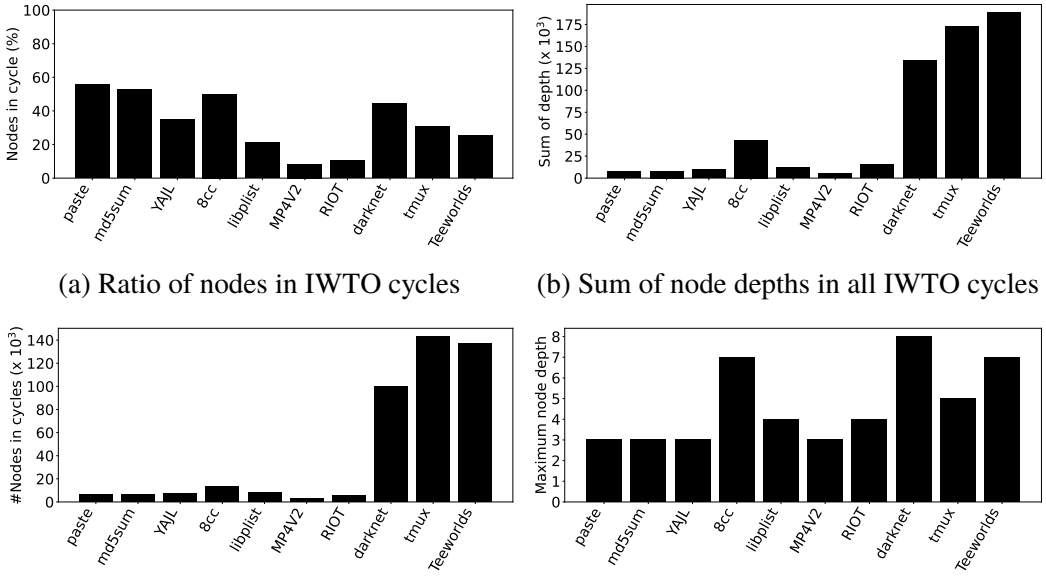
```
1  static plist_t parse_array_node(struct bplist_data *bplist,
2                                  const char** bnode, uint64_t size)
3  {
4      plist_data_t data = plist_new_plist_data();
5      const char *index1_ptr = NULL;
6      ...
7      for (uint64_t j = 0; j < size; j++) {
8          index1_ptr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

Patch



Call Graph

Figure 4.13: The patch and call graph of a real-world buffer overflow in `libplist` [50] found by our tool.



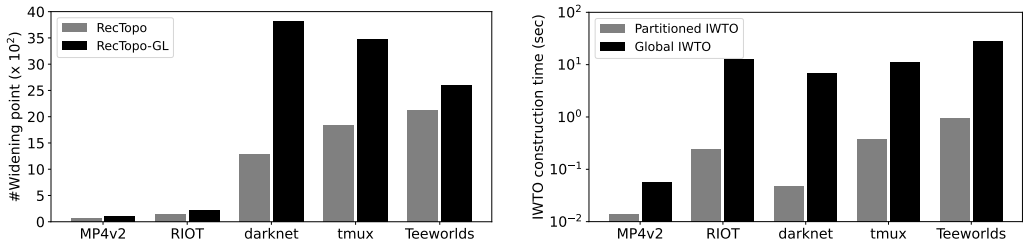
(a) Ratio of nodes in IWTO cycles

(b) Sum of node depths in all IWTO cycles

(c) Number of nodes in IWTO cycles

(d) Maximum node depth in the IWTO

Figure 4.14: Statistics of IWTO in real-world programs.



(a)

(b)

Figure 4.15: (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.

Chapter 5

Conclusions and Suggestions for Future Research

5.1 Conclusions

This thesis enhances the precision and scalability of points-to analysis for race detection. Chapter 3 proposes SLIRD, which utilizes cross-domain abstract interpretation to accurately model points-to information for compound data structures and introduces a race detection-targeted slicing methods to narrow the analysis scope and enhance performance. Experimental data demonstrates that SLIRD achieves an average precision improvement of 89.1% over the baseline tool SVFRD on large-scale industrial programs, highlighting the substantial benefits of more precise points-to analysis. Furthermore, the slicing method substantially enhances the scalability of abstract interpretation, enabling successful analysis of three additional real-world programs with millions of lines of code. Notably, this advancement is realized without compromising soundness. While SLIRD requires on aver-

age 4.8× more execution time, the considerable precision improvement justifies the adoption of more precise points-to analysis. Further more, the speedup observed relative to CROSSRD underscores the effectiveness of slicing in extending complex points-to analyses to large-scale programs. Collectively, these results inspire future investigations into more granular slicing techniques. Chapter 4 introduce RECTOPO to facilitate abstract interpretation in handling recursion, improving its precision and supporting downstream bug detection tasks. Experiments show that RECTOPO achieves a significant precision improvement of 46.51% and an increased bug detection recall rate of 32.98% on real-world recursive benchmarks.

Both works advance the precision of points-to analysis from distinct perspectives and demonstrate the substantial benefit of reducing false positives in downstream bug detection tasks. To further address the computational overhead introduced by the complexity inherent in more precise points-to analyses, this thesis introduces a novel slicing approach that markedly improves the scalability of points-to analysis on large industrial codebases. This contribution enables complex points-to analysis techniques to be applied more effectively to real-world programs, thereby rendering their benefits more accessible and practically attainable.

5.2 Suggestions for Future Research

SLIRD improves precision of points-to information while maintaining scalability by applying cross-domain abstract interpretation to sliced code. However, the downstream race detection process (including MHP analysis and lockset analysis) still uses traditional approaches, which perform standalone analysis without scalar information. Since abstract interpretation is a unified framework with the potential

to replace traditional analyses [31] for thread analysis, a promising future direction would be improving the precision of thread analysis using cross-domain abstract interpretation.

The slicing methods introduced in SLIRD enhances the scalability of points-to analysis for data race detection tasks. However, the slicing methods are restricted to race detection tasks. Given the wide range of static analysis tasks and various downstream applications, achieving a balance between scalability and precision remains a significant challenge. Developing new slicing methods to achieve a better balance between precision and scalability in diverse scenarios would be beneficial.

Another avenue for improving precision and efficiency is the design of fine-grained context sensitivity strategies for functions within recursions. In RECTOPO, we adopt context insensitivity for functions within recursion, which may result in imprecision. However, the proposed methodology is not restricted to a specific context sensitivity strategy. Future exploration of fine-grained context sensitivity strategies would provide a valuable opportunity to balance analysis precision and resource consumption.

References

[1] *8cc: C compiler*. https://github.com/rui314/8cc. 2024. URL: https://github.com/rui314/8cc.

[2] *A buffer overflow patch in libplist*. https://github.com/libimobile-device/libplist/commit/4765d9a60ca4248a8f89289271ac69cbff-cc29bc. 2024.

[3] Temitayo Adefemi. *What Every Computer Scientist Needs To Know About Parallelization*. 2025. arXiv: 2504.03647 [cs.DC]. URL: https://arxiv.org/abs/2504.03647.

[4] Hiralal Agrawal and Joseph R. Horgan. "Dynamic program slicing". In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. PLDI '90. White Plains, New York, USA: Association for Computing Machinery, 1990, pp. 246–256. ISBN: 0897-913647. DOI: 10.1145/93542.93576. URL: https://doi.org/10.1145/93542.93576.

[5] Gianluca Amato and Francesca Scozzari. "Localizing widening and narrowing". In: *International Static Analysis Symposium*. Springer. 2013, pp. 25–42. DOI: 10.1007/978-3-642-38856-9_4.

[6] Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. "Efficiently intertwining widening and narrowing". In: *Science of Computer Programming* 120 (2016), pp. 1–24. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2015.12.005. URL: https://www.sciencedirect.com/science/article/pii/S01676-42315004165.

[7] Lars Ole Andersen. "Program analysis and specialization for the C programming language". In: (1994).

[8] David F. Bacon and Peter F. Sweeney. "Fast static analysis of C++ virtual function calls". In: *SIGPLAN Not.* 31.10 (Oct. 1996), pp. 324–341. ISSN: 0362-1340. DOI: 10.1145/236338.236371. URL: https://doi.org/10.1145/236338.236371.

[9] 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.

[10] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. "RacerD: compositional static race detection". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–28.

[11] 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*. Springer. 2005, pp. 128–141. DOI: 10.1007/BFb0039704.

[12] 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). Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2023, pp. 5–1.

[13] 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*. Springer. 2014, pp. 271–277. DOI: 10.1007/978-3-319-10431-7_20.

[14] 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. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[15] Marek Chalupa and Jan Strejček. "Evaluation of program slicing in software verification". In: *International Conference on Integrated Formal Methods*. Springer. 2019, pp. 101–119.

[16] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. "Interval polyhedra: An abstract domain to infer interval linear relationships". In: *International Static Analysis Symposium*. Springer. 2009, pp. 309–325. DOI: 10.1007/978-3-642-03237-0_21.

[17] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. "DTaint: detecting the taint-style vulnerability in embedded device firmware". In: *2018 48th Annual IEEE/IFIP International*

Conference on Dependable Systems and Networks (DSN). IEEE. 2018, pp. 430–441.

[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*. https://github.com/seahorn/clam. 2024.

[20] Patrick Cousot. *Asynchronous iterative methods for solving a fixpoint system of monotone equations*. Tech. rep. 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.

[22] Patrick Cousot. "Semantic foundations of program analysis". In: *Program flow analysis: theory and applications*. Prentice Hall, 1981, pp. 303–342.

[23] Patrick Cousot and Radhia Cousot. "Abstract interpretation frameworks". In: *Journal of logic and computation* 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511.

[24] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation

of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[25] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "Combination of abstractions in the Astrée static analyzer". In: *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*. Ed. by Mitsu Okada and Ichiro Satoh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 272–300. ISBN: 978-3-540-77505-8. DOI: 10.1007/978-3-540-77505-8_23.

[26] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The Astrée analyzer". In: *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14*. Springer. 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[27] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. "A²I: Abstract² Interpretation". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290355.

[28] *CWE-121: Stack-based buffer overflow*. https://cwe.mitre.org/data/definitions/121.html. 2024.

[29] *CWE-122: Heap-based buffer overflow*. https://cwe.mitre.org/data/definitions/122.html. 2024.

[30] *CWE-126: buffer over-read*. https://cwe.mitre.org/data/definitions/126.html. 2024.

[31] Tomáš Dacík and Tomáš Vojnar. "RacerF: Lightweight Static Data Race Detection for C Code". In: *39th European Conference on Object-Oriented Programming (ECOOP 2025)*. Ed. by Jonathan Aldrich and Alexandra Silva. Vol. 333. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 37:1–37:19. ISBN: 978-3-95977-373-7. DOI: 10.4230/LIPIcs.ECOOP.2025.37. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2025.37.

[32] *Darknet: Open source neural networks in C*. https://github.com/pjreddie/darknet. 2023. URL: https://github.com/pjreddie/darknet.

[33] Manuvir Das, Sorin Lerner, and Mark Seigle. "ESP: Path-sensitive program verification in polynomial time". In: PLDI '02. Berlin, Germany: Association for Computing Machinery, 2002, pp. 57–68. ISBN: 1581134630. DOI: 10.1145/512529.512538.

[34] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. "Static analysis and compiler design for idempotent processing". In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 2012, pp. 475–486.

[35] Matt Elder. "Bourdoncle Components". In: (2010).

[36] Dawson Engler and Ken Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct.

2003), pp. 237–252. ISSN: 0163-5980. DOI: 10.1145/1165389.945468. URL: https://doi.org/10.1145/1165389.945468.

[37] Sukhpal Singh Gill, Huaming Wu, Panos Patros, Carlo Ottaviani, Priyansh Arora, Victor Casamayor Pujol, David Haunschild, Ajith Kumar Parlikad, Oktay Cetinkaya, Hanan Lutfiyya, Vlado Stankovski, Ruidong Li, Yuemin Ding, Junaid Qadir, Ajith Abraham, Soumya K. Ghosh, Houbing Herbert Song, Rizos Sakellariou, Omer Rana, Joel J.P.C. Rodrigues, Salil S. Kanhere, Schahram Dustdar, Steve Uhlig, Kotagiri Ramamohanarao, and Rajkumar Buyya. "Modern computing: Vision and challenges". In: *Telematics and Informatics Reports* 13 (2024), p. 100116. ISSN: 2772-5030. DOI: https://doi.org/10.1016/j.teler.2024.100116. URL: https://www.sciencedirect.com/science/article/pii/S277250302400-0021.

[38] Arie Gurfinkel and Jorge A. Navas. "Abstract interpretation of LLVM with a region-based memory model". In: *Software Verification: 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18–19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18–19, 2021, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 122–144. ISBN: 978-3-030-95560-1. DOI: 10.1007/978-3-030-95561-8_8. URL: https://doi.org/10.1007/978-3-030-95561-8_8.

[39] Ben Hardekopf and Calvin Lin. "Flow-sensitive pointer analysis for millions of lines of code". In: *International Symposium on Code Generation*

and Optimization (CGO 2011). IEEE. 2011, pp. 289–298. DOI: 10.1109/CGO.2011.5764696.

[40] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[41] Rishabh Iyer, Katerina Argyraki, and George Candea. "Automatically Reasoning About How Systems Code Uses the {CPU} Cache". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 581–598.

[42] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. "Fast and accurate static data-race detection for concurrent programs". In: *Proceedings of the 19th International Conference on Computer Aided Verification*. CAV'07. Berlin, Germany: Springer-Verlag, 2007, pp. 226–239. ISBN: 9783540733676.

[43] Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival, et al. "Astrée: Proving the absence of runtime errors". In: *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)* 9 (2010).

[44] Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. "Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters". In: *Proceedings of the ACM on Programming Languages* 7.ICFP (2023), pp. 955–981. DOI: 10.1145/3607863.

[45] Sol Kim, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. "Widening with thresholds via binary search". In: *Software: Practice and Experience* 46.10

(2016), pp. 1317–1328. DOI: https://doi.org/10.1002/spe.2381. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2381. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2381.

[46] Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. "Deterministic Parallel Fixpoint Computation". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371082. URL: https://doi.org/10.1145/3371082.

[47] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Form. Asp. Comput.* 27.3 (May 2015), pp. 573–609. ISSN: 0934-5043. DOI: 10.1007/s00165-014-0326-7. URL: https://doi.org/10.1007/s00165-014-0326-7.

[48] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[49] Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. "A dual number abstraction for static analysis of Clarke Jacobians". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498718.

[50] *libplist: A small portable C library to handle Apple Property List files in binary, XML, JSON, or OpenStep format.* https://github.com/libimobile-device/libplist. 2024. URL: https://github.com/libimobiledevice/libplist.

[51] Kangjie Lu and Hong Hu. "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis". In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK, 2019.

[52] M. S. Lydia, I. Srinath, and J. Gyani. "Static and dynamic attribute slicing tool for object-oriented programs". In: *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*. ICWET '10. Mumbai, Maharashtra, India: Association for Computing Machinery, 2010, pp. 717–722. ISBN: 9781605588124. DOI: 10.1145/1741906.1742073. URL: https://doi.org/10.1145/1741906.1742073.

[53] Zohar Manna. *Mathematical theory of computation*. Dover Publications, Inc., 2003.

[54] *Memcached: A high-performance, distributed memory object caching system*. https://memcached.org/. 2025. URL: https://memcached.org/.

[55] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[56] *MP4v2: A C/C++ library to create, modify and read MP4 files*. https://github.com/enzo1982/mp4v2/. 2023.

[57] *MySQL: A widely adopted relational database management system known for its scalability and reliability in data storage*. https://www.mysql.com/. 2025. URL: https://www.mysql.com/.

[58] Mayur Naik, Alex Aiken, and John Whaley. "Effective static race detection for Java". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 308–319. ISBN: 1595933204. DOI: 10.1145/1133981.1134018. URL: https://doi.org/10.1145/1133981.1134018.

[59] *NIST datasets*. https://samate.nist.gov/SARD/test-suites/116. 2023.

[60] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. "Design and Implementation of Sparse Global Analyses for C-like Languages". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: Association for Computing Machinery, 2012, pp. 229–238. ISBN: 9781450312059. DOI: 10.1145/2254064.2254092.

[61] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. *The Sparrow static analyzer*. https://opam.ocaml.org/packages/sparrow/. 2012.

[62] Hakjoo Oh and Kwangkeun Yi. "An algorithmic mitigation of large spurious interprocedural cycles in static analysis". In: *Software: Practice and Experience* 40.8 (2010), pp. 585–603. DOI: 10.1002/spe.969.

[63] *OpenBLAS: An optimized BLAS library based on GotoBLAS2*. https://www.openblas.net/. 2025. URL: https://www.openblas.net/.

[64] *OpenCV: A leading computer vision and machine learning software library, providing tools for image and video processing, feature detection,*

and real-time applications. https://opencv.org/. 2025. URL: https://opencv.org/.

[65] Komal Pathade and Uday P. Khedker. "Computing partially path-sensitive MFP solutions in data flow analyses". In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 37–47. DOI: 10.1145/3178372.3179497.

[66] Komal Pathade and Uday P. Khedker. "Path sensitive MFP solutions in presence of intersecting infeasible control flow path segments". In: *Proceedings of the 28th International Conference on Compiler Construction*. CC 2019. Washington, DC, USA: Association for Computing Machinery, 2019, pp. 159–169. ISBN: 9781450362771. DOI: 10.1145/3302516.3307349.

[67] D.J. Pearce, P.H.J. Kelly, and C. Hankin. "Efficient field-sensitive pointer analysis of C". In: *ACM TOPLAS* 30.1 (2007), 4–es. DOI: 10.1145/1290520.1290524.

[68] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. "LOCKSMITH: Practical static race detection for C". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.1 (2011), pp. 1–55.

[69] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. "LOCKSMITH: context-sensitive correlation analysis for race detection". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 320–331. ISBN: 1595933204. DOI: 10.

1145/1133981.1134019. URL: https://doi.org/10.1145/1133981.1134019.

[70] *Property list*. https://en.wikipedia.org/wiki/Property_list. 2024.

[71] *Qt: A cross-platform application framework for developing GUIs and multi-platform applications*. https://www.qt.io/. 2025. URL: https://www.qt.io/.

[72] Thomas Rauber and Gudula Rünger. *Parallel programming*. Springer, 2013.

[73] *RecTopo Source Code*. https://github.com/JoelYYoung/RecTopo. 2025.

[74] *RIOT: The friendly OS for IoT*. https://github.com/RIOT-OS/RIOT. 2023.

[75] Xavier Rival and Bor-Yuh Evan Chang. "Calling context abstraction with shapes". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 173–186. ISSN: 0362-1340. DOI: 10.1145/1925844.1926406. URL: https://doi.org/10.1145/1925844.1926406.

[76] Emerson Sales, Omar Inverso, and Emilio Tuosto. "Accurate Static Data Race Detection for C". In: *International Symposium on Formal Methods*. Springer. 2024, pp. 443–462.

[77] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice". In: *Proceedings of the workshop on binary instrumentation and applications*. 2009, pp. 62–71.

[78] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. "Pinpoint: Fast and precise sparse value flow analysis for million lines of code". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 693–706. DOI: 10.1145/3192366.3192418.

[79] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. "Path-Sensitive Sparse Analysis without Path Conditions". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 930–943. ISBN: 9781450383912. DOI: 10.1145/3453483.3454086.

[80] Johannes Späth, Karim Ali, and Eric Bodden. "Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29. DOI: 10.1145/3290361.

[81] Bjarne Steensgaard. "Points-to analysis in almost linear time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 32–41. ISBN: 0897917693. DOI: 10.1145/237721.237727. URL: https://doi.org/10.1145/237721.237727.

[82] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. "Interactive abstract interpretation with demanded summarization". In: *ACM Transac-*

tions on Programming Languages and Systems 46.1 (2024), pp. 1–40. DOI: 10.1145/3648441.

[83] Yulei Sui and Jingling Xue. "SVF: interprocedural static value-flow analysis in LLVM". In: *Proceedings of the 25th international conference on compiler construction*. ACM. 2016, pp. 265–266. DOI: 10.1145/2892208.2892235.

[84] Yulei Sui, Ding Ye, and Jingling Xue. "Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis." In: *IEEE Trans. Software Eng (TSE '14)*. 40.2 (2014), pp. 107–122. DOI: 10.1109/TSE.2014.2302311.

[85] Yulei Sui, Ding Ye, and Jingling Xue. "Static memory leak detection using full-sparse value-flow analysis". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA '12. ACM, 2012, pp. 254–264. DOI: 10.1145/2338965.2336784.

[86] Wuliang Sun, Benoit Combemale, Robert B France, Arnaud Blouin, Benoit Baudry, and Indrakshi Ray. "Using slicing to improve the performance of model invariant checking". In: *The Journal of Object Technology* (2015), p. 28.

[87] Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. "OMPRacer: a scalable and precise static race detector for OpenMP programs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[88] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[89] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: (1955).

[90] *Teeworlds: A retro multiplayer shooter*. https://teeworlds.com/. 2024. URL: https://teeworlds.com/.

[91] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[92] *Tmux: Tmux source code*. https://github.com/tmux/tmux. 2023.

[93] Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. "Detecting Blocking Errors in Go Programs Using Localized Abstract Interpretation". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3561154.

[94] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. "Static race detection for device drivers: the Goblint approach". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE '16. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 391–402. ISBN: 978145033-8455. DOI: 10.1145/2970276.2970337. URL: https://doi.org/10.1145/2970276.2970337.

[95] Mark Weiser. “Program slicing”. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0897911466.

[96] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. “Database-Backed Program Analysis for Scalable Error Propagation”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 586–597. DOI: 10.1109/ICSE.2015.75.

[97] *YAJL: A fast streaming JSON parsing library in C*. https://github.com/lloyd/yajl. 2023.

[98] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. “Program Analysis via Efficient Symbolic Abstraction”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485495.

[99] Misun Yu, Seung-Min Park, Ingeol Chun, and Doo-Hwan Bae. “Experimental Performance Comparison of Dynamic Data Race Detection Techniques”. In: *ETRI Journal* 39.1 (2017), pp. 124–134. DOI: https://doi.org/10.4218/etrij.17.0115.1027. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.4218/etrij.17.0115.1027. URL: https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.17.0115.1027.

[100] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. “Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code”. In: *Proceedings of the Fourteenth EuroSys Conference*

2019. EuroSys '19. Dresden, Germany: ACM, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303972.