Multi-Stage On-Demand Program Slicing for Modular Analysis of Multi-Threaded Programs

ANONYMOUS AUTHOR(S)

Precise analysis of multi-threaded programs requires combining flow-sensitive pointer analysis (FSPTA) with interleaving and lock analysis (ILA) to reason about cross-thread value flows under feasible concurrent executions. ILA computes may-happen-in-parallel (MHP) relations and lock-release spans to determine when shared accesses can occur concurrently. Unfortunately, these analyses are both expensive and tightly coupled: FSPTA needs ILA to rule out infeasible inter-thread def-use relations, while ILA needs alias information to identify interference-relevant interactions. As a result, whole-program analyses often spend most of their time on code that is irrelevant to the client query. We present MSLI, an on-demand slicing framework for modular analysis of multi-threaded programs. It extracts compact, query-relevant program slices while preserving the answers of downstream analyses. Unlike single-pass slicing over a unified dependence graph, MSLI performs multi-stage slicing with analysis-specific criteria. Concretely, a lightweight pre-analysis establishes an over-approximation of inter-thread value flows and performs ILA slicing source extraction to identify the MHP and lock-span queries required later for ILA slicing. The refined main-phase ILA results then enable reconstruction of a thread-aware value-flow graph to guide FSPTA slicing, supporting modular analysis and downstream clients. We implement MSLI in SVF and evaluate it on ten large real-world projects with data race detection as a representative client. Compared with the unsliced baseline (FSAM), MSLI reduces the analyzed ICFG to 5.4% (ILA) and 25.7% (FSPTA), reduces ILA/FSPTA runtimes to 4.7%/18.3%, and cuts total analysis time to 20.8% on average, while producing identical query outcomes and race alarms.

1 Introduction

Precise analysis of multi-threaded programs requires not only identifying memory locations that may be shared across threads, but also determining when such interactions can occur. Flow-sensitive pointer analysis [26] is therefore essential, as it explicitly captures the ordering of memory updates and enables reasoning about the feasibility of cross-thread data flows. Sparse flow-sensitive pointer analysis [10, 26, 42] further improves scalability by restricting the propagation of data-flow facts (points-to set) along relevant def-use chains, thereby avoiding redundant propagation along control flows that do not affect pointer relationships.

In the presence of concurrency, constructing accurate def-use chains additionally requires *interleaving and lock analysis* (ILA) to determine which statements from different threads may execute concurrently. ILA comprises two components: (i) *interleaving analysis*, which computes may-happen-in-parallel (MHP) relations to identify statements that can execute simultaneously across threads, and (ii) *lock analysis*, which derives lock-release spans to determine when shared accesses are protected by common locks. FSAM [41] addresses this challenge by integrating sparse, flow-sensitive pointer analysis (FSPTA) with ILA results [19]. In particular, FSAM leverages MHP relations and lock spans to conservatively construct a value-flow graph that captures feasible inter-thread def-use relationships, while pruning spurious connections arising from infeasible thread interleavings or mutually exclusive lock-protected regions. Subsequent work has focused on improving the scalability of flow-sensitive pointer analysis [10, 44] and interleaving and lock analysis [55]. For example, Li et al. [10] enhance pointer analysis efficiency by merging similar pointer abstractions, whereas Zhou et al. [55] propose an MHP analysis based on vector clocks. Despite these advances, whole-program analyses often spend most of their time on code that is irrelevant to the client query. This motivates the need for techniques that can reduce the analysis scope while preserving precision for queries of interest.

Our goal is **on-demand program slicing for modular analysis of multi-threaded programs**: given a client query (e.g., whether two memory access can race), we want to retain only the

code relevant to that query, i.e., preserving (i) inter-thread value flows enabled by pointer/alias information from FSPTA and (ii) feasible interleavings and synchronization constraints captured by ILA. This enables modular analysis: instead of running whole-program FSPTA and ILA over the entire codebase, we reduce the analyzed ICFG/value-flow graph to what is strictly necessary for the module or query at hand, without sacrificing precision where it matters.

An intuitive approach is to apply program slicing to **eliminate statements irrelevant to the downstream client queries**, thereby accelerating both FSPTA and ILA while preserving precision. Prior work on slicing multi-threaded programs primarily contributes to the concept of cross-thread interference/synchronization dependencies (referred to as synchronization dependence) to model the effects of thread interleaving [30, 34]. They do *not* propose a dedicated slicing strategy for jointly accelerating FSPTA and ILA. Nevertheless, a straightforward baseline is to construct a unified dependence graph that combines data dependences for FSPTA with synchronization dependences for ILA, and then perform a single slicing pass over the combined graph. We refer to this baseline as *single-pass slicing*.

While this single-pass slicing strategy safely over-approximates the dependencies needed by both analyses, it often yields slices that are larger than necessary. This is because the unified dependency graph conservatively combines all dependencies required by FSPTA and ILA, thereby collapsing analysis-specific dependencies into a single solution. As a result, dependencies that are required only under either inter-thread or value-flow conditions are treated as universally necessary during slicing. This loss of distinction forces the slicing process to retain statements that are not simultaneously required for both analyses under the same execution context, leading to slices that are larger than necessary. Consequently, the resulting slice may still contain extraneous code that does not impact the specific queries of either analysis, thereby limiting the potential efficiency gains. This observation motivates the need for a more refined slicing strategy that can **tailor program slices to the distinct requirements of FSPTA and ILA**, thereby producing smaller and more effective slices.

However, separately slicing the program for FSPTA and ILA is non-trivial due to their differing dependency requirements and mutual interdependence. FSPTA depends on ILA results to determine feasible inter-thread def-use relations: MHP information identifies which stores and loads may execute concurrently, while lock spans prune infeasible value flows between mutually exclusive regions. Conversely, slicing for ILA must account for inter-thread aliasing identified by pointer analysis in order to accurately determine which statements may interfere and, consequently, to identify appropriate slicing criteria. Ignoring these interdependencies, for example, by conservatively assuming that all statements from different threads may interleave, would introduce excessive imprecision, resulting in overly conservative value-flow connections. This, in turn, degrades the effectiveness of the slicing for FSPTA and undermines the scalability of the subsequent main analysis phases.

To address these challenges, we introduce MSli, a novel slicing technique for on-demand modular analysis of multi-threaded programs. The key idea is to perform a multi-stage slicing, in which **slicing is carried out incrementally using analysis-specific criteria** rather than producing a single slice that simultaneously satisfies the requirements of both FSPTA and ILA. Specifically, to relax the tight interdependence between the two analyses, MSli first applies an imprecise but fast interleaving and lock analysis as a preliminary stage. This initial analysis, together with a pre-pointer analysis (e.g., Andersen's pointer analysis [8]), identifies an over-approximation of inter-thread value flows, which is sufficient to determine the slicing criteria for the subsequent ILA slicing phase. The refined ILA then provides precise MHP relations and lock spans that enable the construction of a more accurate inter-thread value-flow graph, which guides the FSPTA slicing phase, ensuring that only statements relevant to feasible inter-thread interactions are retained.

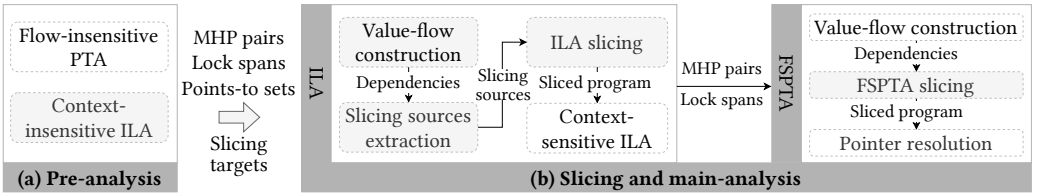


Fig. 1. Framework overview of multi-staged slicing.

Figure 1 presents the pre-analysis phase (Figure 1(a)) and the slicing and main-analysis phase (Figure 1(b)) of our on-demand multi-stage slicing framework. The framework takes LLVM intermediate representation (LLVM IR) as input and leverages a static analysis tool, SVF [42], to construct an interprocedural control-flow graph and a call graph. It then performs a lightweight pre-analysis consisting of flow-insensitive Andersen-style pointer analysis and context-insensitive interleaving and lock analysis, together with the extraction of data dependences (value-flow edges) and synchronization dependences. Based on the resulting dependency information, the framework conducts an ILA slicing phase that incorporates both data and synchronization dependences. Prior to this slicing step, the preliminary pointer and interleaving analyses construct an over-approximation of inter-thread value flows and perform ILA slicing source extraction to identify the MHP and lock-span queries required by subsequent analyses. This ensures that statements necessary for resolving these queries are preserved during slicing, even before precise inter-thread value-flow construction. Finally, the framework performs FSPTA slicing guided by data dependences derived from a reconstructed, thread-aware value-flow graph, which is built using refined, context-sensitive MHP relations and lock spans obtained from ILA. Sparse, flow-sensitive pointer analysis is then executed on the resulting sliced value-flow graph.

In summary, this paper makes the following contributions:

- We propose MSli, an on-demand, multi-stage slicing framework for modular analysis of multi-threaded programs. It decouples the tight interdependence between flow-sensitive pointer analysis (FSPTA) and interleaving and lock analysis (ILA) by progressively refining analysis-specific slicing criteria, avoiding the excessive conservatism of single-pass slicing while preserving downstream query results.
- We formalize slicing correctness as preservation of downstream-relevant ILA predicates (MHP relations and lock spans) and FSPTA alias answers, and show that our sliced ICFG reconstruction preserves required reachability and contexts.
- We implement MSli in SVF [42] and evaluate it on ten large projects, demonstrating that the resulting slices accelerate modular FSPTA and preserve client results with data race detection as a representative client. Compared with the unsliced baseline (FSAM), MSli reduces the analyzed ICFG to 5.4% (ILA) and 25.7% (FSPTA), reduces ILA/FSPTA runtimes to 4.7%/18.3%, and cuts total analysis time to 20.8% on average, while producing identical query outcomes and race alarms.

2 Preliminaries and Problem Formulation

This section provides the necessary background on the FSAM framework [41], a representative flow-sensitive pointer analysis (FSPTA) framework for multi-threaded program analysis. Building upon this foundation, we then present a formal problem formulation that characterizes the objectives of our approach.

2.1 FSAM Overview

FSAM begins by performing a flow-insensitive pointer analysis to compute preliminary points-to information. This information is then used to perform ILA (§2.3), which derives concurrency

$$\text{[T-SPAWN]}\ \frac{t \xrightarrow{(c,fk_i)} t' \quad t' \xrightarrow{(c',fk_{i'})} t''}{t \xrightarrow{(c,fk_i)} t''} \qquad \text{[T-SYNC]}\ \frac{t \xleftarrow{(c,jn_i)} t' \quad t' \xleftarrow[full]{(c',jn_{i'})} t''}{t \xleftarrow{(c,jn_i)} t''}$$

$$\text{[T-SIBLING]}\ \frac{t'' \xrightarrow{(c,fk_i)} t \quad t'' \xrightarrow{(c',fk_{i'})} t' \quad (c,fk_i) \neq (c',fk_{i'})}{t \bowtie t'}$$

Fig. 2. Inference rules for deriving thread relations.

information about thread interactions. Subsequently, both the points-to and concurrency results are leveraged to construct thread-aware value-flow graphs, which enable the execution of FSPTA (§2.4). It is important to note that FSAM does not incorporate program slicing; instead, it directly applies both FSPTA and ILA to the entire program, analyzing all statements and control-flow paths without reduction.

2.2 Abstract Thread Model

This section presents the abstract thread model employed by FSAM. We formalize the notions of abstract threads, intra-thread control-flow graphs, and the thread-relation model that underpin the analysis.

Abstract threads. We use *abstract thread* to approximate *concrete thread* during actual program execution. An abstract thread t represents either the initial entry thread originating from `main` or a library entry point, or a context-sensitive invocation of `pthread_create` at a fork site. Each abstract thread may correspond to one or multiple runtime thread instances. When an abstract thread corresponds to multiple spawning events, for instance, under different calling contexts or due to loops or recursion, it is classified as a *multi-forked thread*.

Intra-thread control-flow graph ($ICFG_t$). ILA is formulated as a data-flow analysis operating over the collection of $ICFG_t$. Each abstract thread t is associated with a corresponding $ICFG_t$. Intra-procedural edges, denoted $s \xrightarrow{intra} s'$, capture sequential control flow within a procedure. Inter-procedural edges model procedure calls and returns: call edges $s \xrightarrow{call_i} s'$ connect a call node s to the entry statement s' of the callee at callsite i, while return edges $s \xrightarrow{ret_i} s'$ connect the exit statement s of the callee to the corresponding return node s' at the same callsite. Callsites to `pthread_create` and `pthread_join` connect their call nodes directly to the corresponding return nodes via intra-procedural edges

Thread relation model. Figure 2 formalizes three fundamental relations over abstract threads. The *spawning relation* $t \xrightarrow{(c,fk_i)} t'$ indicates that thread t spawns thread t' at fork site fk_i under call string context c, either directly or transitively. The *synchronization relation* $t \xleftarrow{(c,jn_i)} t'$ indicates that spawned thread t' joins with its spawner t at join site (c, jn_i), provided t' is not multi-forked. Synchronization may be direct or transitive, but transitive synchronization requires all intermediate joins to be full joins. Two threads t and t' are *siblings*, written $t \bowtie t'$, if they share a common spawner and neither spawns the other. A *happens-before* relation $t \succ t'$ holds when the fork site of t' is post-dominated by a join site of t.

2.3 Interleaving and Lock Analysis (ILA)

ILA comprises two components: interleaving analysis, which computes may-happen-in-parallel (MHP) information for statements, and lock analysis, which derives lock spans.

$$[\texttt{I-SPAWN}]\ \frac{t \xrightarrow{(c,fk_i)} t' \quad (t,c,fk_i) \xrightarrow{intra} (t,c,s) \quad (c',s') = Entry(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t,c,s) \quad \{t\} \subseteq \mathcal{I}(t',c',s')}$$

$$[\texttt{I-SIBLING}]\ \frac{t \bowtie t' \quad (c,s) = Entry(\mathcal{S}_t) \quad (c',s') = Entry(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t',c',s') \quad \{t'\} \subseteq \mathcal{I}(t,c,s)}$$

$$[\texttt{I-SYNC}]\ \frac{t \xleftarrow{(c,jn_i)} t'}{\mathcal{I}(t,c,jn_i) = \mathcal{I}(t,c,jn_i) \backslash \{t'\}} \qquad [\texttt{I-CALL}]\ \frac{(t,c,s) \xrightarrow{call_i} (t,c',s') \quad c' = c.push(i)}{\mathcal{I}(t,c,s) \subseteq \mathcal{I}(t,c',s')}$$

$$[\texttt{I-INTRA}]\ \frac{(t,c,s) \xrightarrow{intra} (t,c,s')}{\mathcal{I}(t,c,s) \subseteq \mathcal{I}(t,c,s')} \qquad [\texttt{I-RET}]\ \frac{(t,c,s) \xrightarrow{ret_i} (t,c',s') \quad i = c.top() \quad c' = c.pop()}{\mathcal{I}(t,c,s) \subseteq \mathcal{I}(t,c',s')}$$

Fig. 3. Inference rules for interleaving analysis [41].

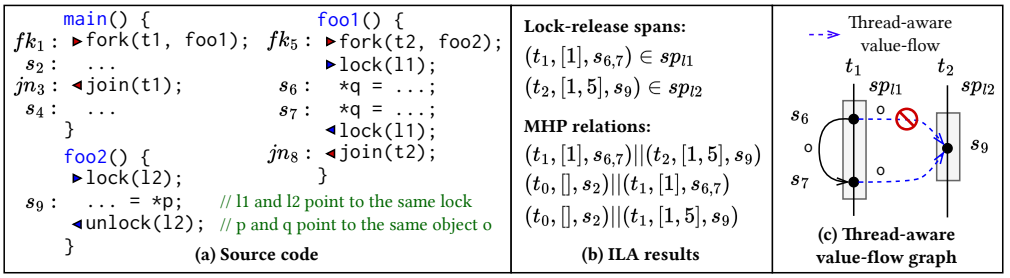


Fig. 4. An example of ILA and FSPTA.

Interleaving analysis. Interleaving analysis is formulated as a data-flow analysis that operates flow- and context-sensitively over the $ICFG_t$s of all threads. For a statement s in thread t's $ICFG_t$, the analysis approximates which threads may execute in parallel with t at the program point where s is executed. We denote this interleaving state as $\mathcal{I}(t,c,s)$, where c is the calling context that distinguishes the specific instance of s when its enclosing method is invoked under context c.

Figure 3 presents the inference rules for interleaving analysis. Each thread t forked at (c, fk_i) is associated with a start procedure $\mathcal{S}_t$, whose entry statement is given by $Entry(\mathcal{S}_t) = (c', s)$ with context $c' = c.push(i)$. Rule [I-SPAWN] adds interleaving states for the statement following the fork site and the spawned thread's entry. Rule [I-SIBLING] adds interleaving states to sibling threads' entry statements. Rule [I-SYNC] removes interleaving states after join sites. Rules [I-CALL], [I-INTRA], and [I-RET] propagate interleaving states along intra-thread control flows. We write $(t_1, c_1, s_1) \parallel (t_2, c_2, s_2)$ to denote that statement s_1 in thread t_1 under context c_1 may happen in parallel with statement s_2 in thread t_2 under context c_2. This holds if either (1) $t_1 \neq t_2$ and $t_2 \in \mathcal{I}(t_1, c_1, s_1) \wedge t_1 \in \mathcal{I}(t_2, c_2, s_2)$, or (2) $t_1 = t_2$ and t_1 is multi-forked.

Lock analysis. Lock analysis computes lock-release spans (Definition 1) for context-sensitive lock sites by performing a flow- and context-sensitive analysis over lock/unlock operations, guided by the points-to information obtained from pre-analysis.

Definition 1 (Lock-Release Spans)**.** A lock-release span sp_l at a lock site $(t, c, lock(l))$ is defined as the set of statements in $ICFG_t$ reachable from $(c, lock(l))$ to its matching release site $(c', unlock(l'))$, computed via forward reachability where calls and returns are matched context-sensitively. The span is considered valid if and only if l and l' point to the same unique runtime lock object, denoted $l \equiv l'$.

Example 1. Figure 4 illustrates both interleaving analysis and lock analysis. Two lock-release spans, sp_{l1} and sp_{l2}, are protected by a common lock, since $l1$ and $l2$ are identified as must-aliases. The program exhibits two immediate fork relations: $t_0 \xRightarrow{([],fk_1)} t_1$ and $t_1 \xRightarrow{([2],fk_5)} t_2$. By transitivity, we obtain the spawning relation $t_0 \xRightarrow{([],fk_1)} t_2$. Consequently, the MHP relation $(t_1, [], s_2) \parallel (t_2, [1,5], s_9)$ holds. Since the join site jn_8 is a full join, a synchronization relation $t_2 \xLeftarrow{([],jn_3)} t_0$ is established, which ensures that $(t_0, [], s_4) \nparallel (t_2, [1,5], s_9)$.

2.4 Sparse Flow-Sensitive Pointer Analysis (FSPTA)

Building upon the ILA results and the pre-analysis points-to information, we construct a thread-aware value-flow graph and subsequently perform sparse pointer resolution over this graph.

Thread-aware value-flow construction. The construction of value flows proceeds in two stages. In the first stage, thread-oblivious value flows are established using only the points-to information obtained from pre-analysis [41]. In the second stage, we incorporate *thread-aware* value flows by leveraging both points-to and concurrency information. Specifically, for each pair of MHP store-load or store-store statements $(t, c, s) \parallel (t', c', s')$, we introduce a thread-aware value-flow edge $s \overset{o}{\hookrightarrow} s'$ for each object $o \in AS(p, q)$, where $AS(p, q)$ denotes the intersection of the points-to sets derived from pre-analysis. This construction is formalized by the following rule:r

$$[\texttt{THREAD-VF}]\ \frac{\begin{array}{c} s : *p = \_ \quad s' : \_ = *q \quad \text{or} \quad s' : *q = \_ \\ (t, c, s) \parallel (t', c', s') \quad o \in AS(p, q) \end{array}}{s \overset{o}{\hookrightarrow} s'}$$

Subsequently, we compute *non-interference lock pairs* to refine the [THREAD-VF] rule by excluding value flows whose endpoints form non-interference lock pairs under all contexts.

Definition 2 (Non-Interference Lock Pairs)**.** For an object $o \in \mathcal{A}$ and a lock span sp_l, we define the *span head* and *span tail* as follows:

$$HD(sp_l, o) = \{(t, c, s) \in sp_l \mid \nexists (t, c', s') \in sp_l : s' \overset{o}{\hookrightarrow} s\},$$

$$TL(sp_l, o) = \{(t, c, s) \in sp_l \mid s \text{ is a store} \wedge \nexists (t, c', s') \in sp_l : (s' \text{ is a store} \wedge s \overset{o}{\hookrightarrow} s')\}.$$

Let $(t, c, s) \parallel (t', c', s')$ be an MHP statement pair, where s is a store. Suppose both statements are protected by at least one common lock, i.e., there exist locks l and l' such that $(t, c, s) \in sp_l$, $(t', c', s') \in sp_{l'}$, and $l \equiv l'$. We say that the pair forms a *non-interference lock pair* if $(t, c, s) \notin TL(sp_l, o)$ or $(t', c', s') \notin HD(sp_{l'}, o)$ holds.

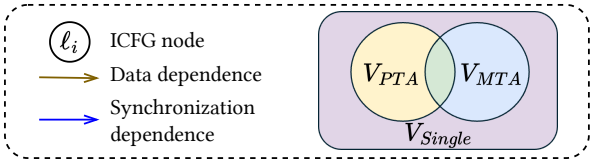
Example 2. We revisit the example in Figure 4 to illustrate the thread-aware value-flow construction process. Figure 4(c) depicts the resulting graph. Since $(t_1, [1], s_{6,7}) \parallel (t_2, [1,5], s_9)$ and $o \in AS(p, q)$, two thread-oblivious value-flow edges, $s_6 \overset{o}{\hookrightarrow} s_9$ and $s_7 \overset{o}{\hookrightarrow} s_9$, are initially introduced. However, according to lock analysis, $(t_1, [1], s_6) \notin TL(sp_{l1}, o)$, which implies that $(t_1, [1], s_6)$ and $(t_2, [1,5], s_9)$ constitute a non-interference lock pair. Consequently, the value flow $s_6 \overset{o}{\hookrightarrow} s_9$ is removed. In contrast, since $(t_1, [1], s_7) \in TL(sp_{l1}, o)$, the corresponding value flow $s_7 \overset{o}{\hookrightarrow} s_9$ is preserved.

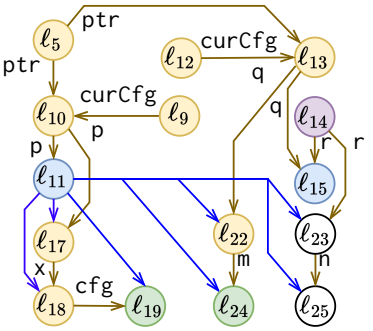
Sparse pointer resolution. We apply the Andersen-style sparse pointer resolution method for sequential programs [41] to the constructed thread-aware value-flow graph to derive over-approximated points-to results.

```c
1  int main() {
2    Config *curCfg;
3    Config cfgA, cfgB, backupCfg;
4
5    Config **ptr = &curCfg;
6
7    pthread_t t;
8    ...
9    *ptr = &cfgA;
10   Config *p = *ptr;
11   pthread_create(&t,NULL,readCfg,p);
12   *ptr = &cfgB;
13   Config *q = *ptr;
14   Config *r = &backupCfg;
15   applyCfgUpdate(q, r);
16 }
```

```c
17 void *readCfg(void *x) {
18   Config *cfg = (Config *)x;
19   int v = cfg->ver; // Spurious race
20   ...
21 }
22 void applyCfgUpdate(Config *m,
23                     Config *n) {
24   m->ver = ...;     // Spurious race
25   n->checksum = ...;
26 }
```

(a) Source code





(b) Data and synchronization dependence graph

Fig. 5. A motivating example that illustrates the slice results of single-pass slicing compared to our multi-stage slicing approach.

2.5 Problem Formulation

Definition 3 (Query Preservation for ILA and FSPTA). Let P'_{ILA} and P'_{FSPTA} be the sliced programs for ILA and FSPTA, respectively. Let $[\![\varphi]\!]_P$ denote the answer to query φ on program P. The downstream task specifies a set of context-sensitive statement instances $CSS_{ILA} \subseteq T \times C \times S$ for ILA queries and a set of variables VAR_{PTA} for alias queries. Slicing preserves ILA and FSPTA queries if (i) for any $(t_1, c_1, s_1), (t_2, c_2, s_2) \in CSS_{ILA}$, $[\![(t_1, c_1, s_1) \parallel (t_2, c_2, s_2)]\!]_{P'_{ILA}} = [\![(t_1, c_1, s_1) \parallel (t_2, c_2, s_2)]\!]_P$, (ii) for any $(t, c, s) \in CSS_{ILA}$ and any $o \in \mathcal{A}$, $[\![(t, c, s) \in sp_l]\!]_{P'_{ILA}} = [\![(t, c, s) \in sp_l]\!]_P$, and (iii) for any $v_1, v_2 \in VAR_{PTA}$, $[\![AS(v_1, v_2)]\!]_{P'_{FSPTA}} = [\![AS(v_1, v_2)]\!]_P$.

We formalize the goal of our slicing approach as query preservation between the original program and its sliced counterparts:

> Given a multi-threaded program P and a downstream analysis task (e.g., data race detection) that requires both ILA and FSPTA results, our goal is to compute two sliced programs P'_{ILA} and P'_{FSPTA} such that both slices preserve all query results required by the downstream task, as formalized in Definition 3, and the combined cost of performing ILA on P'_{ILA} and FSPTA on P'_{FSPTA} is substantially lower than performing these analyses on the original program P.

3 A Motivating Example

Figure 5(a) presents a real-world configuration-management scenario involving concurrent reads and updates. The program comprises a main thread that spawns a worker thread to execute `readCfg`. Prior to spawning, the main thread assigns the address of `cfgA` to `curCfg` at ℓ_9 and passes this address as the thread argument via pointer `p`. Within `readCfg`, the worker thread (created at ℓ_{11}) reads the `ver` field of `cfgA` at ℓ_{19}. Subsequently, after spawning the worker thread, the main thread updates `curCfg` to point to `cfgB` at ℓ_{12} and invokes `applyCfgUpdate`, passing the addresses of

cfgB and backupCfg via pointers p and r, respectively. This function then writes to both cfgB and backupCfg at ℓ_{24} and ℓ_{25}, respectively.

The program does not exhibit any data race [12, 18, 21], as the main thread and worker thread operate on two distinct objects, cfgB and cfgA, respectively. However, a flow-insensitive pointer analysis cannot distinguish between the two assignments to curCfg at ℓ_9 and ℓ_{12}. Consequently, it conservatively concludes that p and q may alias and potentially refer to either cfgA or cfgB. This imprecision leads to a spurious race warning between the read at ℓ_{19} and the write at ℓ_{24}. In contrast, a flow-sensitive pointer analysis correctly infers that p points exclusively to cfgA while q points exclusively to cfgB, thereby eliminating the false positive.

Motivation. While sparse flow-sensitive pointer analysis achieves high precision, it remains computationally expensive for large-scale codebases [42]. For downstream clients such as race detection, a large portion of the program is *irrelevant to the analysis query*: analyzing these statements does not influence the final result. For instance, statements such as ℓ_{23} and ℓ_{25} do not contribute to determining the race detection outcome, as the write to backupCfg at ℓ_{25} does not race with any statement in other threads. This observation motivates the application of program slicing to eliminate irrelevant code, thereby accelerating the subsequent interleaving and lock analysis (ILA) and flow-sensitive pointer analysis (FSPTA) stages.

Single-Pass Slicing. A straightforward baseline approach constructs both data dependences and synchronization dependences [30, 34], then computes a single-pass slice over the unified dependence graph. Figure 5(b) illustrates this combined dependence graph for the example program, where data dependences are denoted as $\longrightarrow$ and synchronization dependences as $\longrightarrow$. The region labeled V_{Single} in the figure represents the result of this single-pass slicing method. However, ILA and FSPTA require distinct program subsets to achieve both precision and soundness. Specifically, certain statements are essential for ILA but irrelevant to FSPTA, while others are crucial for FSPTA but unnecessary for ILA. For instance, ℓ_{11} serves as a synchronization primitive that enables ILA to determine the may-happen-in-parallel relation between ℓ_{19} and ℓ_{24}, and ℓ_{15} is essential for context management; however, neither statement contributes to pointer resolution for cfg at ℓ_{19} or m at ℓ_{24}, rendering them irrelevant to FSPTA. Conversely, statements such as ℓ_{10} and ℓ_{13} are critical for propagating pointer information in FSPTA but do not affect the concurrency analysis performed by ILA. Furthermore, merging these dependencies into a single slice V_{Single} introduces additional statements (colored purple) that are not required by either analysis (e.g., ℓ_{14}). This over-approximation leads to inefficiencies in both analysis stages. These observations motivate the design of a multi-stage slicing technique that separately accelerates ILA and FSPTA by computing tailored slices for each analysis stage.

Our Approach. Rather than computing a single slice over the combined dependence graph, we propose a multi-stage slicing approach that decouples the analysis into two distinct stages, each tailored to its specific dependence requirements. We identify target statements that require exclusively ILA results and those that require exclusively FSPTA results, then compute slices based on their respective dependences: data dependence for FSPTA, and synchronization and call dependences for ILA. In this example, the race statements necessitate ILA information to determine concurrency relations, while the pointer values accessed by these statements require FSPTA information for precise alias resolution. Consequently, we generate two distinct slices: V_{ILA}, indicated by blue and green regions in the figure, representing the ILA slice, and V_{PTA}, indicated by yellow and green regions, representing the FSPTA slice.

Our multi-stage slicing approach yields tighter slices for both ILA and FSPTA compared to the single-pass approach. First, each stage operates on a smaller subgraph than the single-pass

slice ($V_{ILA} \subseteq V_{Single}$ and $V_{PTA} \subseteq V_{Single}$), as each slice retains only the statements relevant to its respective analysis. Second, statements that are irrelevant to both analyses (e.g., ℓ_{15}) are eliminated entirely, thereby reducing the overall volume of code that must be analyzed across all stages.

4 Approach

MSLI begins with a lightweight pre-analysis to identify *slicing targets*, then perform ILA slicing (§4.2) to retain statements affecting concurrency analysis, followed by FSPTA slicing (§4.3) to retain statements affecting pointer resolution in the main-phase analyses.

4.1 Pre-Analysis

Pre-analysis incorporates flow-insensitive PTA [8] and context-insensitive ILA (details in §2.3) to provide points-to and concurrency information for the main phase, respectively. It also identifies potential slicing targets. For instance, in data race detection, race pairs constitute the slicing targets. We denote a potential race pair as $(s, s') \in RacePairs_{pre}$, where s and s' are may-happen-in-parallel memory access statements (with at least one being a write) that access the same memory location without proper synchronization in at least one thread pair.

4.2 ILA Slicing

ILA slicing aims to retain a minimal set of statements such that the ILA results for these statements remain unchanged. The process proceeds in three steps: (i) extracting slicing sources that may be queried by downstream analysis, (ii) performing synchronization-dependence slicing to retain relevant synchronization primitives, and (iii) performing call-dependence slicing to recover the calling contexts of the retained statements. Finally, we reconstruct a sliced ICFG by removing non-target nodes and adding bridge edges to preserve reachability among the remaining nodes.

ILA slicing source extraction. The slicing sources comprise not only the race statements identified during pre-analysis, but also statements that contribute to determining *thread-aware* value flows during value-flow construction in the subsequent FSPTA, thereby requiring concurrency information. Figure 6 summarizes the extraction rules.

$$[\texttt{INIT}]\ \frac{(s, s') \in RacePairs_{pre}}{\{s, s'\} \subseteq V_{ILA}} \qquad [\texttt{THREAD-VF}]\ \frac{s \overset{o}{\hookrightarrow} s' \in ThreadVF(G'_{VFG})}{Query(s \overset{o}{\hookrightarrow} s') \subseteq V_{ILA}}$$

Fig. 6. Inference rules for ILA slicing source extraction.

Rule [INIT] directly includes all statements in $RaceStmt_{pre}$, as these statements may be queried during the final race detection phase. Rule [THREAD-VF] extracts the statements that will be queried for ILA information during thread-aware value-flow construction in the subsequent FSPTA (see Section 2.4). To construct thread-aware value flows, we first build a thread-aware value-flow graph VFG_{pre} using the ILA information obtained from pre-analysis. Since VFG_{pre} may contain value flows that are ultimately unused in the subsequent FSPTA (after FSPTA slicing is applied), we extract a sliced graph VFG'_{pre} based on the same criterion employed in FSPTA slicing (Section 4.3). We denote by $ThreadVF(VFG'_{pre})$ the set of thread-aware value-flow edges constructed on VFG'_{pre}. For each remaining thread-aware value-flow edge $s \overset{o}{\hookrightarrow} s'$, we define $Query(s \overset{o}{\hookrightarrow} s')$ as the set of statements that must be queried to determine the validity of this edge:

$$Query(s \overset{o}{\hookrightarrow} s') \;=\; \{s, s'\} \cup \bigcup_{((t,s),(t',s')) \in InstPair(s,s')} \pi\Big(Query_o\big((t, s), (t', s')\big)\Big),$$

$$\text{[D-SPAWN]}\ \frac{t \overset{fk_i}{\Longrightarrow} t' \quad (t, fk_i) \xrightarrow{intra} (t, s) \quad s' = Entry(\mathcal{S}_{t'}) \quad FKS = Prim(t \overset{fk_i}{\Longrightarrow} t')}{FKS \subseteq \mathcal{D}(s) \quad FKS \subseteq \mathcal{D}(s')}$$

$$\text{[D-SIBLING]}\ \frac{t \bowtie t' \quad s = Entry(\mathcal{S}_t) \quad s' = Entry(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t \quad FKS = Prim(t \bowtie t')}{FKS \subseteq \mathcal{D}(s) \quad FKS \subseteq \mathcal{D}(s')}$$

$$\text{[D-SYNC]}\ \frac{t \overset{c,jn_i}{\Longleftarrow} t' \quad JNS = Prim(t \overset{c,jn_i}{\Longleftarrow} t')}{JNS \subseteq \mathcal{D}(jn_i)} \qquad \text{[D-INTRA-TD]}\ \frac{(t,s) \rightarrow (t,s')}{\mathcal{D}(s) \subseteq \mathcal{D}(s')}$$

Fig. 7. Rules for deriving synchronization dependencies in interleaving analysis.

where $InstPair(s, s') = \{((t, s), (t', s')) \mid (t, s) \parallel (t', s')\}$ denotes the set of may-happen-in-parallel instance pairs identified during pre-analysis, with each instance being a pair consisting of a thread and a statement. The projection $\pi(X) = \{x \mid \exists t : (t, x) \in X\}$ extracts the statement component from each instance.

For each instance pair, we collect all relevant witness instances within the corresponding lock-release spans:

$$Query_o\big((t,s),(t',s')\big) = \bigcup_{(t',s') \in sp_{l'}} \big(Pred_{sp_{l'}}(t',s',o)\big) \cup \bigcup_{(t,s) \in sp_l} \big(Succ_{sp_l}(t,s,o)\big)$$

where

$$Pred_{sp_{l'}}(t',s',o) = \left\{ (t',x) \in sp_{l'} \;\middle|\; x \overset{o}{\hookrightarrow} s' \right\},$$
$$Succ_{sp_l}(t,s,o) = \left\{ (t,x) \in sp_l \;\middle|\; x \text{ is a store } \wedge\ s \overset{o}{\hookrightarrow} x \right\}.$$

Intuitively, $Pred_{sp_{l'}}(t', s', o)$ and $Succ_{sp_l}(t, s, o)$ provide the intra-thread witnesses necessary to determine membership in $HD(sp_{l'}, o)$ and $TL(sp_l, o)$, respectively. These membership tests, in turn, determine the outcome of the non-interference lock-pair test and consequently establish whether the thread-aware value flow exists. To ensure soundness, lock-release spans are computed conservatively (i.e., as may-spans), thereby accounting for all potential non-interference cases.

Synchronization-dependence slicing. Given the slicing sources, we perform synchronization-dependence slicing to retain the relevant *synchronization primitives* in the slice, that is, concurrency-related statements (including `fork`/`join` for interleaving analysis and `lock`/`unlock` for lock analysis) that may affect the interleaving behavior of other statements.

Figure 7 presents the rules for deriving synchronization primitives for interleaving analysis. These rules mirror those in Figure 3, but instead of computing interleaving information, they collect the synchronization primitives on which the derivation depends. We summarize the collected dependencies using a function $\mathcal{D} : S \rightarrow 2^S$, where $\mathcal{D}(s)$ denotes the set of synchronization primitives on which statement s depends. Rules `[D-SPAWN]` and `[D-SIBLING]` extract the fork sites that witness spawning and sibling relationships among threads, while Rule `[D-SYNC]` collects join sites required to establish potential joining relations. Together, these rules capture the synchronization primitives that affect the interleaving behavior of the involved statements. Rule `[D-INTRA-TD]` propagates these dependencies along intra-thread control-flow edges.

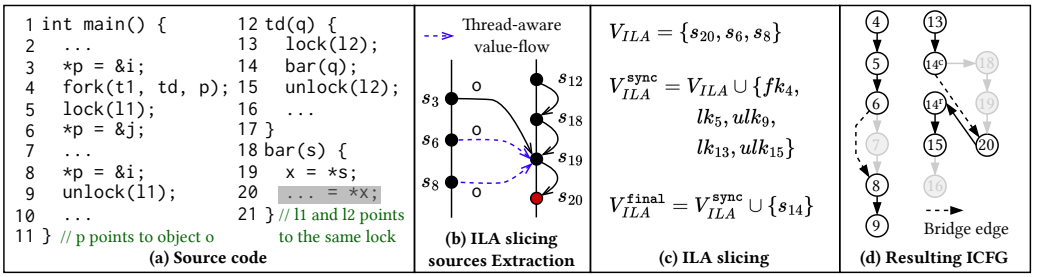


Fig. 8. ILA slicing example.

The function $Prim(t\,R\,t')$ extracts the synchronization primitives that witness a thread relation $R \in \{\overset{fk_i}{\Longrightarrow}, \overset{jn_i}{\Longleftarrow}, \bowtie\}$ (Section 2.2) between abstract threads t and t'. *Prim* is defined as follows:

$$Prim(t\,R\,t') = \begin{cases} \{fk_i\} \cup \{fk_{i'} \mid \exists\, t'' :\ t \overset{fk_i}{\Longrightarrow} t'' \wedge t'' \overset{fk_{i'}}{\Longrightarrow} t'\} & R = \overset{fk_i}{\Longrightarrow}, \\ \{jn_i\} \cup \{jn_{i'} \mid \exists\, t'' :\ t \overset{jn_i}{\Longleftarrow} t'' \wedge t'' \overset{jn_{i'}}{\underset{full}{\Longleftarrow}} t'\} & R = \overset{jn_i}{\Longleftarrow}, \\ Prim(t'' \overset{fk_i}{\Longrightarrow} t') \cup Prim(t'' \overset{fk_{i'}}{\Longrightarrow} t) & R = \bowtie,\ \exists\, t'' :\ t'' \overset{fk_i}{\Longrightarrow} t' \wedge t'' \overset{fk_{i'}}{\Longrightarrow} t. \end{cases}$$

Synchronization primitives for lock analysis are obtained directly from pre-analysis results. Based on the may-lock results, we derive synchronization primitives by including all relevant `lock` statements and their matching `unlock` statements. Formally, for any statement $s \in S$, if s may be protected by a lock-release span sp_l, then we include the corresponding `lock` and `unlock` sites in its dependency set:

$$\forall s \in S,\ \forall l, l' :\ \big(\exists t :\ (t, s) \in sp_l\big) \wedge l \equiv l' \ \Rightarrow\ lock(l) \in \mathcal{D}(s)\ \wedge\ unlock(l') \in \mathcal{D}(s).$$

Given the dependence function $\mathcal{D}$, we extend the slicing sources with their synchronization-dependence witnesses to derive the augmented slice:

$$V_{ILA}^{\text{sync}} = V_{ILA} \cup \{s_d \mid s \in V_{ILA} \wedge s_d \in \mathcal{D}(s)\}.$$

Call-dependence slicing. Finally, we perform call-dependence slicing to recover the call sites necessary for preserving the calling contexts of the involved statements. We compute backward reachability on the call graph, starting from functions that contain statements in V_{ILA}^{sync}, to collect all call sites along the reachable paths. Let $CS(V_{ILA}^{\text{sync}})$ denote the set of such call sites. For each call site $i \in CS(V_{ILA}^{\text{sync}})$, we include both the corresponding call node $call_i$ and return node ret_i in the slice. The final ILA slice is then defined as:

$$V_{ILA}^{\text{final}} = V_{ILA}^{\text{sync}} \cup \{\, call_i,\ ret_i \mid i \in CS(V_{ILA}^{\text{sync}}) \,\}.$$

Sliced ICFG construction. Given the complete set of nodes to be retained, we construct a sliced ICFG for performing ILA. Algorithm 1 presents the construction procedure. For each node to be deleted, the algorithm inserts bridge edges to preserve reachability among the remaining nodes (Line 13). These bridge edges are added to the sliced ICFG only if both their source and destination nodes are retained (Line 11). Consequently, the construction preserves both the connectivity among originally reachable nodes and the post-dominance relations [22] among retained nodes, thereby maintaining the partial join/lock semantics necessary for sound ILA.

Theorem 1 (Correctness of ILA Slicing)**.** Let G be the original ICFG and G' be the sliced ICFG produced by ILA slicing source extraction and ICFG reconstruction. For any ILA query whose

Algorithm 1: Sliced ICFG construction algorithm.

Input: $G(V, E)$: Original ICFG of the program
$V_{\text{ILA}}^{\text{final}}$: Final ILA slice of the program
Output: $G'(V', E')$: Sliced ICFG of the program

```
1  Function Slice(G(V, E), V_ILA^final):
2      V' := V_ILA^final; E' := ∅;
3      V_del := ∅; E_br := ∅;
4      // Construct bridge edges to preserve control flow
5      foreach v_del ∈ V \ V_ILA^final do
6          foreach v' → v_del ∈ E ∪ E_br ∪ E' ∧ v' ∉ V_del do
7              foreach v_del → v'' ∈ E ∪ E_br ∪ E' ∧ v'' ∉ V_del do
8                  if v' ∈ V_ILA^final ∧ v'' ∈ V_ILA^final then
9                      E'.insert((v', v''));
10                 else
11                     E_br.insert((v', v''));
12         V_del.insert(v_del);
13     // Retain edges between nodes in the slice
14     foreach e ∈ E do
15         if e.src ∈ V_ILA^final ∧ e.dst ∈ V_ILA^final then
16             E'.insert(e);
17     return G'(V', E');
```

operands are in V_{ILA}, the ILA result on G' is identical to that on G, satisfying the query preservation property for ILA (Definition 3).

Proof Sketch. ILA slicing constructs the final slice V_{ILA}^{final} by closing the initial slicing sources V_{ILA} under two types of dependences: (i) synchronization dependences, which retain all synchronization primitives (`fork`/`join` and `lock`/`unlock`) that may affect interleaving states and lock-release spans, and (ii) call dependences, which retain the call/return nodes necessary to preserve the calling contexts of the retained primitives and target statements. This closure ensures that every synchronization event and calling-context element that may influence the ILA results for statements in V_{ILA} is included in the slice.

The sliced ICFG G' removes nodes not in V_{ILA}^{final} and inserts bridge edges to preserve reachability and post-dominance relationships among retained nodes, maintaining the partial join/lock semantics required for sound ILA. Since all relevant synchronization primitives, calling contexts, and reachability relationships are preserved in G', the ILA results for retained statements remain identical to those on the original ICFG G, satisfying the query preservation property (Definition 3). □

Example 3. Figure 8 illustrates the ILA slicing process. The `main` function spawns a thread executing `td` and passes pointer `p` as an argument. The thread `td` invokes `bar` within sp_{l2}, passing `q` as an argument. Inside `bar`, statement s_{19} loads the pointer argument, and statement s_{20} dereferences it to access the referenced memory. Pre-analysis identifies s_{20} as a potential race statement. We first construct a thread-aware value-flow graph and slice it according to the criterion in Section 4.3, as shown in Figure 8(b). This retains statements $\{s_3, s_6, s_8, s_{12}, s_{18}, s_{19}, s_{20}\}$, inducing two

thread-aware value-flow edges: $s_6 \overset{o}{\hookrightarrow} s_{19}$ and $s_8 \overset{o}{\hookrightarrow} s_{19}$. ILA slicing proceeds in three steps. First, source extraction selects s_{20} via [INIT] and includes s_6 and s_8 via [THREAD-VF] for concurrency queries. Second, synchronization-dependence slicing adds relevant synchronization primitives: $\{fk_4, lk_5, ulk_9, lk_{13}, ulk_{15}\}$. Third, call-dependence slicing recovers calling contexts by adding corresponding call and return nodes (e.g., (14^c) and (14^r)). The sliced ICFG retains selected nodes and inserts bridge edges (e.g., ⑥ ⇢ ⑧) to preserve reachability. Context-sensitive ILA on the sliced graph yields the required concurrency information while preserving results.

4.3 FSPTA Slicing

FSPTA slicing reduces the analysis codebase by retaining only statements that affect the points-to information of target pointers (e.g., those accessed in potential race statements), thereby preserving query results for these variables (Definition 3). This is achieved through data-dependence slicing.

Data dependence slicing. Figure 9 presents the inference rules for data-dependence slicing. Given the slicing targets $RacePair_{pre}$, the [INIT] rule initializes the set V_{FSPTA} with statements that define the pointers accessed by statements in $RacePair_{pre}$. Here, $Def(v)$ denotes the statement that defines variable v, while $Ptr(s)$ denotes the pointer accessed in statement s. Subsequently, the [DATA] rule collects all statements that may influence the final pointer analysis outcome by performing a backward traversal along the data-dependence chain, thereby ensuring that all relevant dependencies are captured.

$$[\texttt{INIT}]\ \frac{(s, s') \in RacePairs_{pre} \quad v = Ptr(s) \quad v' = Ptr(s')}{Def(v) \cup Def(v') \subseteq V_{FSPTA}} \qquad [\texttt{DATA}]\ \frac{s \in V_{FSPTA} \quad (s_d, s) \in E_d}{s_d \in V_{FSPTA}}$$

Fig. 9. Inference rules for data dependence slicing.

Definition 4 (Data Dependence)**.** Data dependence is a binary relation E_d over program statements: $(s, s') \in E_d$ if and only if a value defined in statement s is used in statement s'. These data dependencies are inherently represented in the thread-aware value-flow graph, which is constructed using the context-sensitive ILA results from the main analysis phase and the pre-analysis points-to results. Only value flows retained in VFG'_{pre} are considered in the construction of E_d.

Theorem 2 (Correctness of FSPTA Slicing)**.** FSPTA on the sliced program yields the same points-to results for the target variables as on the original program, thereby satisfying the query preservation property for FSPTA (Definition 3).

Proof Sketch. FSPTA relies on thread-aware value flows, which depend on ILA results. ILA slicing source extraction (Section 4.2) preserves all statements required for constructing thread-aware value flows: Rule [INIT] includes statements in $RaceStmt_{pre}$, while Rule [THREAD-VF] includes statements queried for ILA information during value-flow construction.

Let VFG denote the original value-flow graph and VFG' the sliced graph. Since VFG_{pre} uses pre-analysis ILA results that over-approximate main-phase results, we have $ThreadVF(VFG') \subseteq ThreadVF(VFG'_{pre})$, ensuring VFG'_{pre} contains all useful thread-aware value flows. For each retained edge $s \overset{o}{\hookrightarrow} s' \in ThreadVF(VFG')$, the set $Query(s \overset{o}{\hookrightarrow} s')$ contains endpoints and intra-span witnesses needed to query ILA results. By Theorem 1, these ILA results are preserved.

Data-dependence slicing constructs V_{FSPTA} by retaining all statements reachable along value flows from the defining statements of target pointers. Since all statements contributing to the points-to relation of target variables are included in V_{FSPTA}, sparse FSPTA on the sliced graph yields identical points-to results, satisfying the query preservation property. □

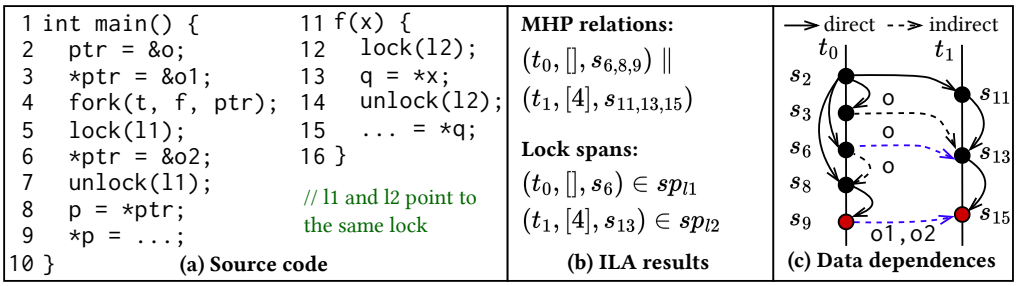


Fig. 10. FSPTA slicing example.

Example 4. Consider the example in Figure 10, where `main` spawns a thread executing `f` at fk_4 with pointer `ptr`. Pre-analysis identifies a potential race pair $(s_6, s_9) \in RacePairs_{pre}$. Using pre-analysis points-to results and context-sensitive ILA results (Figure 10(b)), we construct a thread-aware value-flow graph inducing the data dependencies in Figure 10(c). These dependencies include: (i) direct dependencies (e.g., $s_2 \rightarrow s_3$) from def-use chains of top-level variables, (ii) indirect dependencies (e.g., $s_3 \overset{o}{\dashrightarrow} s_{13}$) from def-use chains of address-taken variables, and (iii) thread-aware def-use dependencies (e.g., $s_6 \overset{o}{\dashrightarrow} s_{13}$) capturing inter-thread value flows.

Data-dependence slicing proceeds from the identified race pair as follows. The `[INIT]` rule initializes V_{FSPTA} with statements defining the accessed pointers (i.e., s_8 and s_{13} for `p` and `q`). The `[DATA]` rule then extends V_{FSPTA} by backward traversal along data-dependence paths to include all transitively influencing statements. The resulting slice $V_{FSPTA} = \{s_2, s_3, s_6, s_8, s_{11}, s_{13}\}$ is necessary and sufficient for deriving the points-to results of p and q in the FSPTA phase.

5 Evaluation

In this section, we evaluate our multi-stage differential slicing approach, focusing on its effectiveness in reducing both the analysis scope and computational cost, while verifying that it preserves precision and soundness. To demonstrate the advantages of our design, we also implement and compare against a straightforward baseline, single-pass slicing, which merges ILA slicing source extraction and FSPTA slicing into a unified pass.

Our evaluation aims to address the following research questions:

- RQ1 **Slicing effectiveness:** To what extent does slicing reduce the analysis scope and computational cost for both ILA and FSPTA?
- RQ2 **Precision and soundness preservation:** Does slicing preserve the correctness of ILA results (including interleaving analysis and lock analysis) and FSPTA results (alias queries) that are essential for downstream bug detection?
- RQ3 **Benefit of differential slicing:** What are the additional advantages of differential slicing over single-pass slicing in terms of analysis scope reduction and overall efficiency?

5.1 Datasets and Implementation

Datasets. We evaluate MSli on ten real-world projects spanning diverse application domains. These include five large applications from the `PARSEC-3.0` benchmark suite [11]: `radiosity` (graphics rendering), `ferret` (content-based similarity search server), `bodytrack` (computer vision-based body tracking), `raytrace` (physically-based ray tracing), and `x264` (H.264/MPEG-4 AVC video encoding). Additionally, we evaluate five widely-used open-source applications: `httpd` [4] (Apache HTTP server), `darknet` [3] (neural network framework for deep learning), `teeworlds` [2] (online multiplayer game engine), `NanoMQ` [1] (lightweight MQTT broker for IoT edge platforms),

Table 1. Statistics of 10 open-source projects. *#LOI* denotes the number of lines of LLVM instructions. *#Method*, *#Call*, *#Ptr*, and *#Obj* are the numbers of functions, method calls, pointers, and memory objects, respectively. $|V|$ and $|E|$ are the numbers of ICFG nodes and edges.

Project	*#LOI*	*#Method*	*#Call*	*#Ptr*	*#Obj*	$\|V\|$	$\|E\|$
radiosity	14,873	53	631	11,853	163	7,459	8,374
ferret	18,490	58	579	12,528	187	7,938	8,323
bodytrack	21,358	81	627	16,034	212	8,930	9,005
raytrace	106,473	391	4,879	93,617	617	53,762	54,483
x264	147,110	481	5,440	143,325	985	125,571	128,374
httpd	182,833	2,526	7,124	191,051	13,147	172,095	162,743
darknet	213,858	987	9,920	195,684	2,568	135,169	148,925
teeworlds	272,443	2,334	20,590	328,211	5,813	228,395	215,926
NanoMQ	385,737	3,235	39,415	382,746	31,736	313,571	441,396
redis	469,883	6,304	41,097	549,641	9,817	415,349	457,513
Total	1,833,058	16,450	130,302	1,924,690	65,245	1,468,239	1,635,062

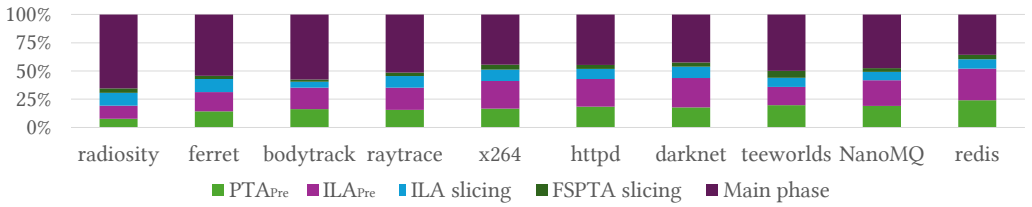


Fig. 11. Time breakdown of MSli across analysis phases.

and `redis` [5] (in-memory key-value database). This diverse selection enables us to assess the effectiveness of MSli across realistic and heterogeneous concurrent software systems.

Implementation. All experiments are conducted on an Ubuntu 22.04 server equipped with an 8-core 2.60GHz Intel Xeon CPU and 128 GB of memory. We construct interprocedural control-flow graphs (ICFGs) and value-flow graphs (VFGs) from LLVM IR (LLVM 16.0.0) using the SVF library [42] (v3.2). Our slicing implementation is built on top of FSAM, the flow-sensitive analysis module provided by the SVF framework. FSAM implements both flow-insensitive and flow-sensitive pointer analysis (FSPTA), as well as interleaving and lock analysis (ILA), for both the pre-analysis and main analysis phases.

5.2 RQ1: Effectiveness in Accelerating Analysis

In this section, we evaluate the extent to which MSli reduces the analysis scope and accelerates the overall analysis pipeline. Using ten large-scale projects, we first quantify the reduction in analysis scale and the resulting speedups. We then examine the time breakdown across analysis phases to assess the overhead introduced by slicing.

Reduction in analysis scope and cost. Table 2 compares MSli against FSAM with respect to ICFG size (measured by node and edge counts) and main-phase analysis time, decomposed into ILA, FSPTA, and total execution time. Overall, slicing reduces the ICFG node count to 5.4% for ILA and 25.7% for FSPTA on average, with proportional reductions observed in edge counts. Correspondingly, the runtime for ILA decreases to 4.7% of the baseline, while FSPTA runtime decreases to 18.3%, resulting in an overall reduction in total analysis time to 20.8% (achieving as low as 12.3% on `redis`).

Table 2. Reduction in analysis scope and speedup achieved by MSLI. ICFG size (node/edge counts) and analysis time are reported for FSAM (without slicing) and MSLI (with slicing), broken down by ILA, FSPTA, and total analysis cost.

Project	FSAM					MSLI						
	Code size		Time (sec)			Code size				Time (sec)		
	$\|V\|$	$\|E\|$	ILA	FSPTA	Tot.	$\|V_{ILA}\|$	$\|E_{ILA}\|$	$\|V_{PTA}\|$	$\|E_{PTA}\|$	ILA	FSPTA	Tot.
radiosity	7,459	8,374	2.4	2.7	5.3	832	917	4,324	4,389	0.2	1.5	2.6
ferret	7,938	8,323	4.3	3.1	7.9	938	991	3,716	3,861	0.6	1.3	3.5
bodytrack	8,930	9,005	26.3	34.4	62.3	1,465	1,493	4,280	4,331	1.9	3.8	9.9
raytrace	53,762	54,483	84.5	96.3	184.9	2,354	2,404	15,233	15,415	2.7	10.7	26.1
x264	125,571	128,374	153.8	121.5	281.8	3,152	3,387	17,482	17,570	3.1	14.3	39.1
httpd	172,095	162,743	169.4	172.0	351.1	2,164	2,204	20,154	20,648	5.2	18.1	52.3
darknet	135,169	148,925	241.9	191.1	443.8	3,979	4,185	15,628	16,012	7.3	18.5	60.9
teeworlds	228,395	215,926	267.6	318.8	602.7	3,195	3,278	34,254	35,024	6.5	34.6	82.3
NanoMQ	313,571	441,396	304.8	374.5	701.7	3,914	4,116	42,938	43,153	8.4	47.3	117.3
redis	415,349	457,513	671.4	538.4	1,246.7	5,312	5,440	43,841	44,107	10.9	43.5	152.8
Total	1,468,239	1,635,062	1,926.4	1,852.8	3,888.2	27,305	28,415	201,850	204,510	46.8	193.6	546.8

Table 3. Analysis results before and after slicing. #IA, #LA, and #FSPTA denote the number of positive queries for interleaving analysis, lock analysis, and FSPTA, respectively. #Alarms denotes the number of reported race statements.

Project	*Results (FSAM)*				*Results (MSLI)*			
	#IA	#LA	#FSPTA	#Alarms	#IA	#LA	#FSPTA	#Alarms
radiosity	52	17	26	10	52	17	26	10
ferret	63	21	31	13	63	21	31	13
bodytrack	86	28	42	15	86	28	42	15
raytrace	210	65	72	20	210	65	72	20
x264	198	61	70	19	198	61	70	19
httpd	349	128	139	30	349	128	139	30
darknet	1045	275	253	43	1045	275	253	43
teeworlds	1176	308	294	46	1176	308	294	46
NanoMQ	1643	418	389	54	1643	418	389	54
redis	1620	410	385	53	1620	410	385	53
Total	6442	1731	1701	303	6442	1731	1701	303

Time breakdown and slicing overhead. Figure 11 presents a detailed breakdown of the execution time of MSLI across different phases, including pre-analysis (PTA_{Pre} and ILA_{Pre}), ILA/FSPTA slicing, and the main-phase analysis. The combined overhead introduced by MSLI (comprising ILA_{Pre} and ILA/FSPTA slicing) accounts for 33.7% of the total analysis time on average, reaching at most 40.2% on `radiosity`. In contrast, the main phase accounts for 49.3% on average, which is comparable to the pre-analysis phase. This balance is attributable to the substantial acceleration achieved through slicing. These results demonstrate that our approach delivers significant speedup while incurring only modest overhead.

5.3 RQ2: Precision and Soundness

This section evaluates whether MSLI satisfies the query preservation property for ILA and FSPTA (Definition 3). We compare the fine-grained analysis outcomes produced by MSLI against those of FSAM, measuring the number of positive queries in interleaving analysis (#IA), lock analysis (#LA), and flow-sensitive pointer analysis (#FSPTA), as well as the number of distinct reported

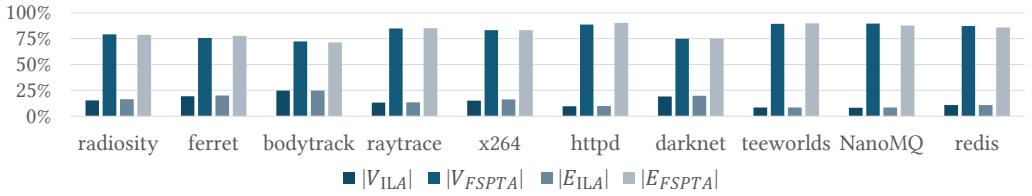


Fig. 12. ICFG size for ILA and FSPTA slicing under MSLI compared to single-pass slicing (normalized).

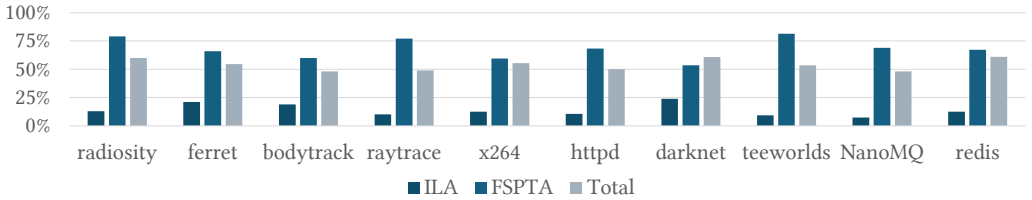


Fig. 13. Analysis time (ILA, FSPTA, and total) under MSLI compared to single-pass slicing (normalized).

race statements (#Alarms). For clarity, we count only queries issued during the final race-detection phase, excluding those used to construct thread-aware value flows, as their effects are already reflected in the final FSPTA results. When the same statement is queried multiple times, we count it only once. Table 3 presents the results. Across all benchmarks, MSLI produces exactly the same query outcomes and race alarms as FSAM, thereby confirming that slicing preserves both ILA and FSPTA results with full precision and soundness.

5.4 RQ3: Ablation Analysis of Differential Slicing

This section evaluates the effectiveness of our multi-stage differential slicing strategy by comparing it against a straightforward baseline. The baseline merges ILA slicing source extraction and FSPTA slicing into a unified pass by computing the transitive closure of all target statements under the combined dependence graph, which we refer to as *single-pass slicing*. We compare the two approaches with respect to slice sizes and the resulting analysis time, measured both for the main analysis phase and for the overall execution.

Figure 12 compares the number of retained nodes under our multi-stage differential slicing approach and the single-pass baseline. Overall, differential slicing reduces the ILA slice to 14.2% of the single-pass baseline on average, while the FSPTA slice is reduced to 82.5%. Figure 13 reports the corresponding speedups in analysis time. Compared to single-pass slicing, differential slicing reduces the main-phase runtime to 18.9% for ILA and 68.1% for FSPTA on average, and achieves an overall reduction in total runtime to 54.0% (reaching as low as 48.1% on NanoMQ). These results demonstrate that the differential slicing strategy yields substantial additional reductions in both slice size and analysis time, thereby confirming the effectiveness of our multi-stage design.

6 Related Work

Static analysis for multi-threaded programs. Flow-sensitive pointer analysis [10, 13, 25, 29, 37, 39, 42] typically achieves higher precision than flow-insensitive alternatives by tracking how pointer values evolve along different program control points. Sparse analyses [25, 26, 42] accelerate data-flow resolution by avoiding redundant propagation, with further speedups achieved through object versioning [10] and saturation techniques [49]. In multi-threaded settings, flow sensitivity introduces the additional challenge of reasoning about inter-thread interference. Rugina and Rinard [37] proposed an interprocedural, flow- and context-sensitive pointer analysis for multi-threaded programs. Sui et al. [41] extended sparse FSPTA to multi-threaded C programs, while Cai

et al. [13, 52] presented Canary, which leverages thread-modular value-flow analysis and SMT-based constraint solving for concurrency bug detection.

Concurrency analysis for multi-threaded programs [12, 18, 19, 21, 35, 55] underpins downstream tasks such as data race detection [14, 24, 28, 31]. Key advances in May-Happen-in-Parallel (MHP) analysis include region-based approaches for Pthreads [19] and static vector clocks [55]. Additional research addresses language-specific concurrency modeling [23, 43], downstream analysis improvements [29, 32, 38], and hybrid static-dynamic techniques [9]. While prior efforts primarily focus on algorithmic advances for scalability and precision, our work pursues a complementary direction: reducing analysis scope via program slicing. Our approach is agnostic to the specific ILA or FSPTA algorithm employed and serves as a front end that reduces runtime overhead while preserving the precision and soundness of target query results.

Program slicing and staged analysis. Program slicing [27, 45, 48, 50, 51] is a fundamental technique in software engineering and program analysis, widely used for debugging, testing, and static analysis. Considerable effort has been devoted to precise dynamic slicing [7, 47, 54], but dynamic slicing is inherently execution-dependent and thus does not provide the sound, all-path guarantees typically desired in static settings [16, 27, 36]. In contrast, static slicing is defined over program dependences, most notably data and control dependences [27, 45, 48, 50, 53], and has been extensively studied as a foundation for analysis reduction.

Slicing concurrent programs introduces additional challenges due to interference across threads. Nanda et al. [33, 34] address this by proposing context-sensitive interference dependences, and subsequent work (e.g., by Krinke [30]) further advances concurrency-aware slicing. These approaches primarily target general-purpose slicing criteria, rather than explicitly guaranteeing the preservation of specific static-analysis results (e.g., MHP/lock predicates or alias queries) for downstream analyses. Staged analysis [15, 17, 41, 46] uses lightweight pre-analysis to guide subsequent expensive analysis via program reduction or slicing [15, 17, 20, 40]. For instance, Cheng et al. [17] use path-insensitive results to accelerate path-sensitive analysis, while Chebaro et al. [15] leverage static analysis to speed up testing. Unlike prior work that computes a single slice for one subsequent analysis, our approach performs differential slicing tailored to the distinct requirements of ILA and FSPTA, preserving precision and soundness for downstream tasks.

7 Conclusion

This paper presents MSli, a multi-stage differential slicing framework that improves the scalability of precise analysis for concurrent programs by tailoring program slices to the distinct needs of ILA and FSPTA while preserving downstream query results. MSli uses lightweight pre-analysis to derive over-approximate inter-thread value flows and perform ILA slicing source extraction, then leverages refined ILA results to reconstruct thread-aware value-flow graphs that guide FSPTA-oriented slicing. On ten large real-world projects, MSli reduces the ICFG to 5.4% for ILA and 25.7% for FSPTA, cuts total runtime to 20.8% on average, and matches the baseline's ILA/PTA query outcomes and race alarms.

Acknowledgments

The authors used generative AI tools (ChatGPT and Claude) exclusively for linguistic refinement. All scientific content, technical contributions, interpretations, and claims remain the sole responsibility of the authors.

Data Availability Statement

We have made our artifact including implementation and dataset publicly available [6].

References

[1] 2023. NanoMQ: An ultra-lightweight and blazing-fast MQTT broker for IoT edge. https://github.com/emqx/nanomq.

[2] 2024. Teeworlds: A retro multiplayer shooter. https://teeworlds.com/.

[3] 2025. Darknet: Open source neural networks in C. https://github.com/pjreddie/darknet.

[4] 2025. httpd: A powerful and flexible HTTP/1.1 compliant web server. https://github.com/apache/httpd.git.

[5] 2025. redis: A preferred, fastest, and most feature-rich cache, data structure server, and document and vector query engine. https://github.com/redis/redis.git.

[6] 2026. MSli Implementation. https://anonymous.4open.science/r/MSli. Anonymous GitHub.

[7] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. 1991. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Canada) *(TAV4)*. Association for Computing Machinery, New York, NY, USA, 60–73. doi:10.1145/120807.120813

[8] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen* (1994).

[9] Jia-Ju Bai, Qiu-Liang Chen, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. 2021. Hybrid static-dynamic analysis of data races caused by inconsistent locking discipline in device drivers. *IEEE Transactions on Software Engineering* 48, 12 (2021), 5120–5135.

[10] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object versioning for flow-sensitive pointer analysis. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 222–235. doi:10.1109/CGO51591.2021.9370334

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. doi:10.1145/1454115.1454128

[12] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. doi:10.1145/3276514

[13] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1126–1140.

[14] Yuandao Cai, Chengfeng Ye, Qingkai Shi, and Charles Zhang. 2022. Peahen: fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 784–796. doi:10.1145/3540250.3549110

[15] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2012. Program slicing enhances a verification technique combining static and dynamic analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. 1284–1291.

[16] Zhenqiang Chen and Baowen Xu. 2001. Slicing object-oriented Java programs. *ACM Sigplan Notices* 36, 4 (2001), 33–40.

[17] Xiao Cheng, Jiawei Ren, and Yulei Sui. 2024. Fast graph simplification for path-sensitive typestate analysis through tempo-spatial multi-point slicing. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 494–516.

[18] Tomáš Dacík and Tomáš Vojnar. 2025. RacerF: Lightweight Static Data Race Detection for C Code. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 37:1–37:19. doi:10.4230/LIPIcs.ECOOP.2025.37

[19] Peng Di, Yulei Sui, Ding Ye, and Jingling Xue. 2015. Region-Based May-Happen-in-Parallel Analysis for C Programs. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP) (ICPP '15)*. IEEE Computer Society, USA, 889–898. doi:10.1109/ICPP.2015.98

[20] Zhijun Ding, Shuo Li, Cheng Chen, and Cong He. 2025. Program Dependence Net and on-demand slicing for property verification of concurrent system and software. *Journal of Systems and Software* 219 (2025), 112221.

[21] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. doi:10.1145/945445.945468

[22] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. doi:10.1145/24039.24041

[23] Fengjuan Gao, Mumu Zhang, Zixiao Zhao, Yu Wang, and Xuandong Li. 2025. Modeling Go Concurrency: A Static Analysis Approach to Data Race Detection. In *Proceedings of the 16th International Conference on Internetware*. 209–219.

[24] Fangming Gu, Qingli Guo, Lian Li, Zhiniang Peng, Wei Lin, Xiaobo Yang, and Xiaorui Gong. 2022. COMRace: Detecting Data Race Vulnerabilities in COM Objects. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX

Association, Boston, MA, 3019–3036. https://www.usenix.org/conference/usenixsecurity22/presentation/gu-fangming

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

[26] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 289–298.

[27] Mark Harman and Robert Hierons. 2001. An overview of program slicing. *software focus* 2, 3 (2001), 85–92.

[28] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.

[29] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and accurate static data-race detection for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 226–239.

[30] Jens Krinke. 2003. Context-sensitive slicing of concurrent programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Helsinki, Finland) *(ESEC/FSE-11)*. Association for Computing Machinery, New York, NY, USA, 178–187. doi:10.1145/940071.940096

[31] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 316–325. doi:10.1145/2594291.2594311

[32] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.

[33] Mangala Gowri Nanda and S. Ramesh. 2000. Slicing concurrent programs. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Portland, Oregon, USA) *(ISSTA '00)*. Association for Computing Machinery, New York, NY, USA, 180–190. doi:10.1145/347324.349121

[34] Mangala Gowri Nanda and S. Ramesh. 2006. Interprocedural slicing of multithreaded programs with applications to Java. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 1088–1144. doi:10.1145/1186632.1186636

[35] Robert O'callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 167–178.

[36] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript static analysis via dynamic shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1129–1140. doi:10.1145/3468264.3468556

[37] Radu Rugina and Martin Rinard. 1999. Pointer analysis for multithreaded programs. *ACM SIGPLAN Notices* 34, 5 (1999), 77–90.

[38] Gabriel Ryan, Abhishek Shah, Dongdong She, and Suman Jana. 2023. Precise Detection of Kernel Data Races with Probabilistic Lockset Analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2086–2103. doi:10.1109/SP46215.2023.10179366

[39] Alexandru Salcianu and Martin Rinard. 2001. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices* 36, 7 (2001), 12–23.

[40] Mifta Sintaha, Noor Nashid, and Ali Mesbah. 2023. Katana: Dual Slicing Based Context for Learning Bug Fixes. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 100 (May 2023), 27 pages. doi:10.1145/3579640

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

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

[43] Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. 2020. Ompracer: A scalable and precise static race detector for openmp programs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[44] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 278–291. doi:10.1145/3062341.3062360

[45] Frank Tip. 1994. *A Survey of Program Slicing Techniques.* Technical Report. NLD.

[46] Sarah Verbelen, Bram Vandenbogaerde, Jens Van der Plas, Noah Van Es, and Coen De Roover. 2024. *Abstract Slicing To Improve The Speed Of Static Program Analysis* (23 ed.). CEUR Workshop Proceedings, Vol. 3941. CEUR Workshop Proceedings, 134–145. Publisher Copyright: © 2024 Copyright for this paper by its authors..

[47] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. 2010. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 253–264. doi:10.1145/1831708.1831740

[48] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 439–449.

[49] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 990–1013.

[50] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (March 2005), 1–36. doi:10.1145/1050849.1050865

[51] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024. A Learning-Based Approach to Static Program Slicing. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 97 (April 2024), 27 pages. doi:10.1145/3649814

[52] Chengfeng Ye, Yuandao Cai, and Charles Zhang. 2024. When Threads Meet Interrupts: Effective Static Detection of Interrupt-Based Deadlocks in Linux. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 6167–6184. https://www.usenix.org/conference/usenixsecurity24/presentation/ye

[53] Xiao Yu and Guoliang Jin. 2018. Dataflow tunneling: mining inter-request data dependencies for request-based applications. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 586–597. doi:10.1145/3180155.3180171

[54] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, USA, 319–329.

[55] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. 2018. May-happen-in-parallel analysis with static vector clocks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO '18)*. Association for Computing Machinery, New York, NY, USA, 228–240. doi:10.1145/3168813