Efficient Strong Updates For Path Sensitive Data Dependence Analysis

Yiyuan Guo
The Hong Kong University of Science and Technology
Hong Kong, China
yguoaz@cse.ust.hk

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

Abstract

Path-sensitive data dependence analysis is a powerful technique widely used in static vulnerability detection. One of the central challenges is how to resolve indirect data dependencies induced by pointer operations: the value loaded from a memory location may depend on different values stored before. Resolving indirect data dependencies in a path-sensitive manner significantly improves the analysis precision, but also induces high overhead that limits its scalability.

We observe that much of the computation effort in path-sensitive data dependence analysis is spent on performing strong updates during load-store matching: a stored value propagates to a load statement only if it is not overwritten by other values stored to the same memory location during the propagation. Answering this question path-sensitively is extremely challenging and often leads to a state explosion that precludes efficient static analysis.

To improve the efficiency for performing strong updates in path-sensitive data dependence analysis, our key insight is that the relation among multiple store statements could be determined in stages: most of the easy cases are handled efficiently by inferring a must-kill relation among the heap store statements, reserving the computationally expensive path-sensitive analysis for the rest. We design a tree-like data structure to encode both the control flow and alias information, which incrementally updates the relation during the analysis. Experiments have shown significant speed-ups and improved state coverage in static analysis through the algorithmic improvements of path-sensitive strong updates.

CCS Concepts

• **Software and its engineering → Software verification and validation**.

Keywords

Static analysis, data dependence analysis.

ACM Reference Format:
Yiyuan Guo and Charles Zhang. 2026. Efficient Strong Updates For Path Sensitive Data Dependence Analysis. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773183

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773183

1 Introduction

Data dependence analysis [15], which tracks the def-use relation among program variables, underpins a great many static analysis techniques for bug finding, such as memory leak, use after free, and null pointer dereference [26, 33, 39, 43]. The central challenge in data dependence analysis is to resolve the indirect dependencies induced by pointer operations. For instance, in the code `*p = k; t=*q`, the variable `t` is *data dependent on* `k` (or equivalently, the value of `k` *may flow to* `t`) if `p` and `q` refer to the same memory location.

For statically detecting bugs in large and realistic systems, the analyzer is often expected to track complex value flows involving deep calling contexts, extensive memory operations, and sophisticated path correlations [5, 33]. Path-sensitive data dependence analysis [6, 28, 33, 40, 43, 44] is a promising direction to achieve this goal and recent advancements [33, 44] have demonstrated significant improvements in terms of both precision and analysis efficiency.

1.1 Inefficient Path-sensitive Strong Updates

To resolve the indirect dependencies precisely, a path-sensitive data dependence analysis tracks the condition for how value flows into and out of the heap. Moreover, to precisely infer heap contents, strong updates [35] are enabled by inferring the condition φ under which storing to a heap location may overwrite its old containing value [44]. While path-sensitivity and strong updates together bring better precision, they also lead to significant scalability challenges.

Falcon [44], the state-of-the-art approach, pairs path-sensitivity with a modular design for better efficiency. Specifically, a bottom-up, path-sensitive pointer analysis is carried out to resolve local indirect dependencies and function side-effects, where the more complex inter-procedural path conditions are discovered on-demand during the bug detection phase. However, we observe that the other axis, i.e., strong updates, still constitutes a major performance bottleneck in path-sensitive data dependence analysis, especially for the realistic programs where sheer number of load / store statements and complex points-to relations are present.

Consider the code example shown in Fig. 1a (ignore Line 13 for now). To resolve indirect dependencies, the analysis needs to match the three load statements at Line 15 (`x1`), 18 (`x2`), and 21 (`x3`) with the three store statements at Line 10 (`q1`), 12 (`q2`), and 17 (`q3`). A conservative answer is any of $\{\texttt{q1}, \texttt{q2}, \texttt{q3}\}$ may flow into [1] any of $\{\texttt{x1}, \texttt{x2}, \texttt{x3}\}$, which is highly imprecise. In contrast, a path-sensitive data dependence analysis will deduce the condition for the value stored at Line i (l_i) to flow into the value loaded at Line j (l_j). For

[1] In this paper, we consider the SSA form [10] of the program, where variables and their corresponding values can be used interchangeably.

instance, the table cell at the column l_{12} and the row l_{21} of Fig. 1b indicates that q2 flows to x3 under the condition $\neg$c2 $\wedge$ $\neg$c3.

However, the precise result for load-store matching comes at a price. To support strong updates, the candidate store statements providing the incoming values need to be enumerated exhaustively for performing case analysis, which leads to a blowup of conditions that throttles the performance. For instance, the statement at l_{18} of Fig. 1a loads the pointer p that could point to either o_4 or o_6 (o_i denotes the memory object allocated at Line i). As shown by Fig. 1c, the conditions for q3, q2, and q1 to be first stored into o_4 and later loaded into x2 are as follows (denoted by $\mathtt{cond}(\mathtt{q}_i \xrightarrow{o_4} \mathtt{x2})$):

- $\mathtt{cond}(\mathtt{q3} \xrightarrow{o_4} \mathtt{x2})$: The condition equals c1 $\wedge$ c3, where c1 is the condition for n to point to o_4 (n equals p due to l_8 and l_{14}) and c3 is the condition guarding l_{17}.
- $\mathtt{cond}(\mathtt{q2} \xrightarrow{o_4} \mathtt{x2})$: The condition includes (1) c1 $\wedge$ $\neg$c2 $\wedge$ c3 and (2) $\neg\mathtt{cond}(\mathtt{q3} \xrightarrow{o_4} \mathtt{x2})$. The first part contains c1 for p to point to o_4, $\neg$c2 and c3 for the execution to reach l_{12} and l_{18} respectively. The second part is a **blocking condition**: for q2 to flow into x2, the effect of storing q3 at l_{17} must be blocked.
- $\mathtt{cond}(\mathtt{q1} \xrightarrow{o_4} \mathtt{x2})$: Similarly, for q1 to flow into x2 via o_4, c1 $\wedge$ c2 $\wedge$ c3 must be satisfied and the effects of l_{17} and l_{12} need to be blocked (the blocking conditions $\neg\mathtt{cond}(\mathtt{q3} \xrightarrow{o_4} \mathtt{x2})$ and $\neg\mathtt{cond}(\mathtt{q2} \xrightarrow{o_4} \mathtt{x2})$).

Essentially, the analysis enables strong updates through path-sensitive reasoning by blocking the effects of other store statements that could interfere with the result of the particular load. In Fig. 1c, conjoining the blocking conditions makes $\mathtt{cond}(\mathtt{q2} \xrightarrow{o_4} \mathtt{x2})$ and $\mathtt{cond}(\mathtt{q1} \xrightarrow{o_4} \mathtt{x2})$ unsatisfiable, meaning that only q3 can flow to x2 via o_4. While this improves the precision, it could easily produce a blow-up of blocking conditions and significantly hurts the efficiency: to match the load statement l_j with the store statement l_i, all the other store statement l_k lying between l_i and l_j need to be "blocked", leading to a linear number of extra blocking conditions, each of which might itself be a complex formula.

1.2 Our Solution

Our goal is to maintain the high precision benefits of path-sensitivity and strong updates for data dependence analysis, while greatly improving the analysis efficiency. Our key insight is that instead of solely relying on the determination of path conditions to enable strong updates, we can infer a must-kill relation among the store statements based on a synergy of *must-alias analysis* and *control flow analysis*. The must-kill relation is computed efficiently and can help to discover the strong update opportunities for the majority of cases, reserving the computationally expensive path-sensitive analysis only for the rest.

For resolving the indirect dependencies of l_{18} in Fig. 1a, our approach will infer that the store at l_{17} must kill the stores at l_{10} and l_{12} based on the following reasons:

(1) **Control Flow Dominance**: Any execution path from l_{10} or l_{12} to l_{18} must go through l_{17}.
(2) **Must Aliasing**: The stored pointer n at l_{17} must alias the stored pointer p at l_{10} and l_{12} (p flows to n according to l_8 and l_{14}).

```
 1  void foo(int ***m) {
 2      int **p, **n;
 3      if (c1)
 4          p = malloc(int*);
 5      else
 6          p = malloc(int*);
 7
 8      *m = p;
 9      if (c2)
10          *p = q1;
11      else
12          *p = q2;
13      // *p = q0;
14      n = *m;
15      int *x1 = *p;
16      if (c3) {
17          *n = q3;
18          int *x2 = *p;
19      }
20
21      int *x3 = *p;
22
23  }
```

(a) Code example.

	From		
To	l_{10} : q1	l_{12} : q2	l_{17} : q3
l_{15} : x1	c2	$\neg$ c2	NA
l_{18} : x2	false	false	c3
l_{21} : x3	c2 $\wedge$ $\neg$ c3	$\neg$ c2 $\wedge$ $\neg$ c3	c3

(b) Final results for load-store match.

$$\mathtt{cond}(\mathtt{q3} \xrightarrow{o_4} \mathtt{x2}) = \mathtt{c1} \wedge \mathtt{c3}$$

$$\begin{aligned}\mathtt{cond}(\mathtt{q2} \xrightarrow{o_4} \mathtt{x2}) &= \mathtt{c1} \wedge \neg\mathtt{c2} \wedge \mathtt{c3} \wedge \\ &\quad \neg\mathtt{cond}(\mathtt{q3} \xrightarrow{o_4} \mathtt{x2}) \\ &= \mathtt{false}\end{aligned}$$

$$\begin{aligned}\mathtt{cond}(\mathtt{q1} \xrightarrow{o_4} \mathtt{x2}) &= \mathtt{c1} \wedge \mathtt{c2} \wedge \mathtt{c3} \wedge \\ &\quad \neg\mathtt{cond}(\mathtt{q3} \xrightarrow{o_4} \mathtt{x2}) \wedge \\ &\quad \neg\mathtt{cond}(\mathtt{q2} \xrightarrow{o_4} \mathtt{x2}) \wedge \\ &= \mathtt{false}\end{aligned}$$

(c) Deriving the conditions for q3, q2, and q1 to flow into x2 via o_4. They can also flow into x2 via o_6, with similar derivations.

Figure 1: Illustration of Path-sensitive data dependence analysis. l_i denotes the statement at Line i and o_i represents the memory object allocated at l_i.

Therefore, for resolving the indirect dependencies of x2, we could early terminate the analysis after discovering the strong update at l_{17}, thereby avoiding the costly path-sensitive reasoning for matching against the values stored in l_{10} or l_{12} as demonstrated in Fig. 1c.

Intuitive as the idea may seem, there are two major challenges for implementing our approach:

(1) Inferring must-aliases in a path-sensitive setting is hard. While control flow dominance could be decided efficiently, determining whether two pointers p and q are must-aliases in a path-sensitive setting is hard because p and q may point to different memory locations under different conditions. This is the inherent complexity of path-sensitive analysis that we intend to avoid in the first place.
(2) Lengthy history of store operations need to be considered. During the load-store matching, a load statement could have multiple preceding store statements as candidates. Constructing the must-kill relation among the candidates needs to check the must-aliasing and control flow dominance relation for times quadratic to the number of candidates, further threatening the performance.

To tackle these challenges, we design an algorithm based on syntactical equivalence checking to efficiently compute an approximation of the must-aliasing results. Further, we adopt a tree-like data structure to maintain and incrementally update the must-kill relation during the analysis to avoid the quadratic behavior.

In this paper, we propose Tuna, a technique that boosts the performance of resolving indirect dependencies in path-sensitive data dependence analysis by enabling efficient strong updates. Tuna

$$\begin{array}{lcl}
\textit{Program } P & ::= & F^{+} \\
\textit{Func } F & ::= & \textit{define } f(v_1, ..., v_n) = \{S\} \\
\textit{Statement } S & ::= & v := e \mid v := \&a \\
& \mid & *v := k \mid k := *v \\
& \mid & \textit{assume}(\gamma) \\
& \mid & v := \textit{ite}(\gamma, v_1, v_2) \\
& \mid & f(a_1, \cdots a_n) \\
& \mid & S_1; S_2 \mid \textit{nondet}(S_1, S_2) \mid \text{return } v
\end{array}$$

Figure 2: A simple programming language.

limits the scope of the path-sensitive reasoning by identifying most strong update opportunities based on a must-kill relation among the heap store statements. We implemented Tuna and evaluated it on 20 real world programs. Comparing with the state-of-the-art approach Falcon [44], Tuna achieves a speedup ranging from 1.2x to 31.5x, reduces the peak memory usage to around 39.5%–62.9%, and is able to finish on targets where Falcon fails due to resource constraints. The enhancement in performance of Tuna also transfers to improved state coverage in static analysis.

In summary, this paper makes the following contributions:

- We identify the efficiency problem of performing strong updates in path-sensitive data dependence analysis for resolving indirect dependencies.
- We infer a must-kill relation among the heap store statements to efficiently enable strong updates by limiting the scope of the path-sensitive analysis. We design algorithms and data structures to optimize the performance of both the must-kill relation computation and the path-sensitive analysis.
- We implement the idea and demonstrate that our approach improves over the state-of-the-art significantly in terms of run time, peak memory and state coverage.

2 Preliminaries

In this section, we give some preliminary definitions and formally state the problem of path-sensitive data dependence analysis. We demonstrate our approach using the language in Fig. 2. We use *Loc* to denote the set of program locations ($l : S$ means a statement S at the program location l) and $\gamma \in Cond$ to denote conditions (or guards). Most of the statements are standard. We represent memory allocation (such as `malloc`) using the address-of operator (`v:=&a`). We use the *assume* statement to model branch conditions and the *nondet* statement to perform a non-deterministic choice.

The program is assumed to be in SSA-form [10] where def-use relations among the top-level variables are made explicit. The statement $v := ite(\gamma, v_1, v_2)$ indicates that v is assigned v_1 if γ holds and is assigned v_2 otherwise (mirroring the ϕ-statement in SSA). In this work, we inherit the same assumptions from previous works [33, 44] that the program is loop free and no aliasing exists among the accessed heap locations from the function's environment.

The goal of path-sensitive data dependence analysis is to construct the sparse value flow graphs (SVFG), where nodes represent values and edges denote how value flows:

Definition 2.1. A path-sensitive data dependence analysis computes a set of SVFGs $\mathcal{G} = (\bigcup_{f \in Funcs} G_f, E_{inter}, L_E)$, where each function `f` has its own SVFG denoted by $G_f(N_f, E_f)$. Each node $n \in N_f$ of G_f is either a program variable `v` (recall our use of SSA) or a memory access path $(*\texttt{p})_l$ denoting the value contained in `p` at the program location l. Each edge $e = (v_1, v_2) \in E_f$ is labelled with a condition $\gamma = L_E(e) \in Cond$ indicating that v_1 may flow to v_2 under the condition, which we write as $v_1 \xrightarrow{\gamma} v_2$.

To facilitate inter-procedural analysis, interface variables [42] are created for a function's SVFG, including its parameters, return value, and the memory access paths from its environment that are either read or written inside the function. Then the value flow across function boundaries are encoded in E_{inter}, which contains edges connecting a function's interface variables to the matching values at the function's call site. For instance, suppose `f` calls `g` as in $l : g(a_1, \cdots a_n)$ and par_i denote the ith parameter of g, we model regular parameter passing as $a_i \xrightarrow{(_l} par_i \in E_{inter}, a_i \in N_f, par_i \in N_g$. Notice that edges from E_{inter} are labelled with open (for calls) or close parentheses (for returns) corresponding to the call site l, modeling context-sensitivity with CFL reachability [31].

An inter-procedural value flow path $\pi = v_1 \rightarrow ...v_k$ is feasible if $\bigwedge_{(v_i \xrightarrow{\gamma_i} v_j) \in \pi} \gamma_i$ is satisfiable and the sequence of edges in π from E_{inter} constitutes an inter-procedural realizable path [31].

The goal of path-sensitive data dependence analysis is to construct SVFGs that resolve the indirect dependencies:

Definition 2.2. The SVFGs $\mathcal{G}$ soundly resolve the indirect dependencies if for any concrete execution of the program in which $t = *u$ fetches the value of k stored before in $*v := k$, there exists a feasible inter-procedural value flow path $\pi = k \rightarrow \cdots \rightarrow t$ in $\mathcal{G}$.

We assume that the domain of conditions, *Cond*, and the decision procedures for checking the satisfiability or validity of these conditions are provided. In this work, we follow Falcon [44] to construct the conditions based on a propositional abstraction of the program and reuse the lightweight semi-decision procedures of Falcon. As opposed to a full-fledged SMT solver, these procedures could decide restricted class of condition formulas in quasi-linear time but may also conservatively classify unsatisfiable conditions as satisfiable.

3 Tuna in a nutshell

In this section, we illustrate how Tuna boosts the performance of resolving indirect dependencies for constructing the SVFGs defined in 2.1. First, we discuss the design of existing works and highlight why they encounter scalability challenges in performing strong updates path-sensitively. Then we introduce our idea of staged resolving of indirect dependencies based on the inference of the must-kill relation for performing strong updates efficiently. Finally, we demonstrate algorithmic optimizations for computing the must-kill relation and performing the path-sensitive analysis.

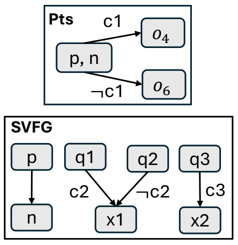
3.1 The Scalability Challenge

Path-sensitive data dependence analysis such as Falcon qualifies the analysis domain with path conditions, which are constructed, propagated, and solved to prune infeasible value flows. As shown in Fig. 3, in order to resolve the indirect dependencies of `x2` at l_{18} of Fig. 1a, Falcon computes:

- A points-to map *Pts*: `p` and `n` point to o_4 and o_6 under the condition `c1` and `¬c1` respectively (Fig. 3b).

l_{10}	o_4	i_0 = (q1, c1 ∧ c2)
	o_6	i_1 = (q1, ¬c1 ∧ c2)
l_{12}	o_4	i_2 = (q2, c1 ∧ ¬ c2)
	o_6	i_3 = (q2, ¬c1 ∧ ¬ c2)
l_{17}	o_4	$i_0{}'$ = (q1, c1 ∧ c2 ∧ **c3**) $i_2{}'$ = (q2, c1 ∧ ¬ c2∧ **c3**) i_4 = (q3, c1 ∧ c3)
	o_6	$i_1{}'$ = (q1, ¬c1 ∧ c2 ∧ **c3**) $i_3{}'$ = (q2, ¬c1 ∧ ¬ c2 ∧ **c3**) i_5 = (q3, ¬c1 ∧ c3)

(a) For location l_i and memory object o_j, the list of reachable store items $(val, cond)$ is maintained.



(b) Points-to map and the constructed SVFG. Edges are labelled with their guarding conditions.

Item	M_i	B_i	$M_i \wedge B_i$ satisfiable?
i_5	M_0 = ¬c1 ∧ c3	B_0 = true	Yes
$i_3{}'$	M_1 = ¬c1 ∧ ¬ c2 ∧ c3	$B_1 = \neg M_0$	No
$i_1{}'$	M_2 = ¬c1 ∧ c2 ∧ c3	$B_2 = \neg M_0 \wedge \neg M_1$	No

(c) The process of load-store matching by traversing the store item list associated with l_{17} and o_6 backwards.

Figure 3: The internals of the existing method on Fig. 1a. We highlight the step for handling the load statement at l_{18}.

- An abstract heap H (Fig. 3a): Given a program point and a memory object, an ordered list of reachable "store items" is computed. For instance, i_0 = (q1, c1 ∧ c2) in the list associated with l_{10} and o_4 indicates that o_4 is stored the value q1 under c1 ∧ c2. Notice that i_0–i_3 originate from the store statements at l_{10} and l_{12}, but "reach" (i.e., the stored values may be loaded back at) l_{17}, leading to the items i'_0–i'_3 (the guarding condition c3 for l_{17} is conjoined).

To resolve indirect dependencies at a particular load statement, the points-to objects of the loaded pointer will be iterated, and for each pointed object, the reachable value items are queried to compute the value flow conditions. Notably, the possible interference among the store statements are explicitly encoded as blocking conditions in order to support strong updates.

Example 3.1. For l_{18} : `x2 = *p` in Fig. 1a, the analysis iterates over each memory object o_i that p points to (i.e., o_4 and o_6 according to Pts), and visits the store list associated with l_{17} and o_i backwards.

For instance, $\{i_5, i'_3, i'_1\}$ is visited in order for the points to target o_6 (Fig. 3c). The first item i_5 = (q3, ¬c1 ∧ c3) is matched under its contained condition M_0 = ¬c1 ∧ c3, indicating that q3 may flow into x2 through the memory object o6 if M_0 holds. Then i'_3 is matched while blocking i_5 (matching condition M_1 and blocking condition $B_1 = \neg M_0$), i'_1 is matched while blocking i_5 and i'_3 (matching condition M_2 and blocking condition $B_2 = \neg M_0 \wedge \neg M_1$).

Due to the blocking conditions, it is inferred that x2 can match with i_5 but cannot match with i'_3 ($M_1 \wedge B_1 \equiv$ `false`) or i'_1 ($M_2 \wedge B_2 \equiv$ `false`). A similar process is carried out for the other points-to target o_4 and infers that i_4 is the only matched value item. Thus, the analysis deduces q3 to be the only incoming value for x2, as shown by the SVFG constructed in Fig. 3b.

Scalability Challenge. Relying on the path-sensitive reasoning for performing strong updates can be expensive because of the large number of store candidates to consider and the explosive accumulation of blocking conditions. For instance, the blocking condition formulated when matching with the first store item in the list such as i'_1 involves all the other items i_5 and i'_3, which could quickly become intractable.

Practical Considerations. Due to the inherent complexity of the path-sensitive analysis, a static analyzer often enforces limitations on its state space exploration, e.g., a path-sensitive data dependence analysis could limit the number of points-to targets that are tracked path-sensitively [44]. Our work shares the same theoretical limit of path-sensitive analysis but aims to significantly improve the performance over the existing method such that a larger state space could be explored more efficiently.

3.2 Identify Strong Updates Opportunities

As introduced in §1, our key improvement lies in a staged design for resolving indirect dependencies in path-sensitive data dependence analysis: strong updates are mostly enabled by exploiting the control flow and must-alias information, limiting the scope of the expensive path-sensitive analysis for better efficiency. To achieve this, we compute a *must-kill relation* among the heap store statements:

Definition 3.1. Given a load statement $l_0 : t = *u$ and a pair of store statements $l_1 : *v_1 = s_1$, $l_2 : *v_2 = s_2$, the store at l_1 must kill the store at l_2 w.r.t l_0 (denoted by $mustKill(l_1, l_2, l_0)$) if:

(1) Any execution path from l_2 to l_0 must go through l_1.
(2) The stored pointers v_1 and v_2 are must-aliases.

The graphical illustration of the definition and the application of it to l_{18} of Fig. 1a are shown in Fig. 4a and Fig. 4b.

A direct consequence of Definition 3.1 is given by the following theorem:

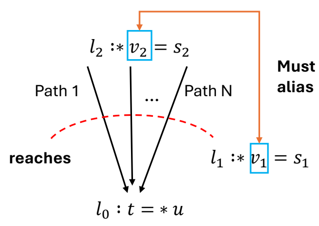
Theorem 3.2. *If $mustKill(l_1, l_2, l_0)$ as per Definition 3.1, then t is not dependent on s_2 in any concrete execution of the program.*

From Theorem 3.2, we are able to achieve a straightforward optimization over the existing approach: If $mustKill(l_1, l_2, l_0)$, we can safely skip l_2 when resolving the indirect dependencies of l_0 because the effect of l_2 is guaranteed to be invalidated by l_1. In other words, we have identified the opportunity to perform a *strong update* at l_1 w.r.t l_2.

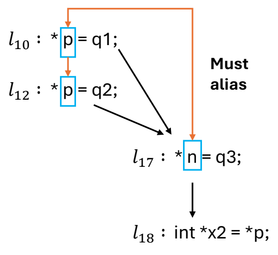
Example 3.3. Fig. 4c illustrates how the must-kill relation in Fig. 4b can help to enable efficient strong updates. Because of the must-kill relations $mustKill(l_{17}, l_{10}, l_{18})$ and $mustKill(l_{17}, l_{12}, l_{18})$, l_{17} is discovered to be the only incoming indirect dependency for l_{18}, allowing the analysis to skip matching between l_{18} and l_{12} or between l_{18} and l_{10}. Meanwhile, we focus the path-sensitive reasoning to places where the killing relation is conditional, e.g., matching l_{21} with l_{12} or l_{10} needs to add the blocking condition for l_{17}.

3.3 Algorithmic Optimizations

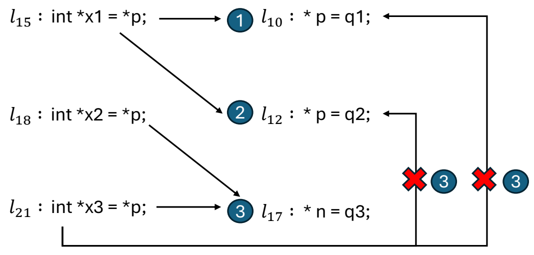
Computing Must Aliases. According to Definition 3.1, a key challenge in computing the must-kill relation lies in the identification of must-aliases. Higher precision in inferring must-aliasing could



(a) Graphical illustration of Definition 3.1



(b) The must-kill relation $mustKill(l_{17}, l_{10}, l_{18})$ and $mustKill(l_{17}, l_{12}, l_{18})$ identified for Fig. 1a.



(c) Utilize the must-kill relation to perform strong updates. Edges connect loads to matched stores and are labeled with stores that should be blocked.

Figure 4: Efficient strong updates through inferring must kill relation among the heap store statements.

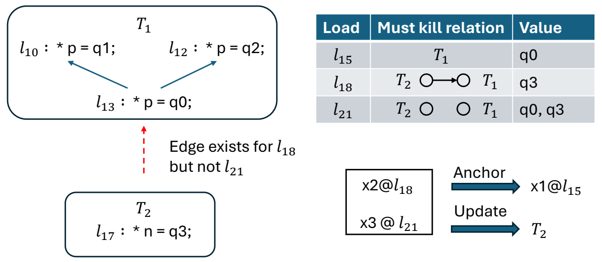
lead to the discovery of more must-alias pairs, which in turn causes the condition in Definition 3.1 to be satisfied more often, allowing more strong updates to be performed efficiently.

Specifically, we pursue path-sensitivity in must-aliasing inference for better precision. Two pointers p1 and p2 can be determined to be must-aliases if they always point to the same target under the logically equivalent condition. For instance, according to Fig. 3 (a), the pointer analysis result suggests that both p and n point to o_4 under c1 and o_6 under ¬c1. They can be determined to be must-aliases because (c1 ∧ c1) ∨ (¬c1 ∧ ¬c1) ≡ true. However, inferring must-aliasing by computing the exact points-to condition and checking for logical equivalence suffers from the inherent complexity of path-sensitive analysis (e.g., there may exist a large number of points-to targets and complex points-to conditions), which we aim to avoid in the first place.

Our idea is to approximate the computation of must-aliases based on the following observation: If p and q always point to the same target under the *syntactically equivalent* condition, they are must-aliases (notice that the converse is not true). Our approximate procedure can determine p and n in Fig. 3b to be must-aliases, but may fail in other cases. Moreover, the syntactic equivalence checking could be efficiently implemented by computing a hash value for each node in the points-to map, which we will detail in §4.

Incrementally Update The Must-Kill Relation. As introduced in §1, computing the must-kill relation in Definition 3.1 leads to quadratic behaviors because all pairs of interfering store statements need to be enumerated for computing must-aliasing and control flow dominance. We propose to optimize the computation by exploiting the fact that the must-kill relation is transitive. Indeed, for a given load statement, the must-kill relation can be represented as a *must-kill forest* (i.e., a set of trees), which we incrementally update.

Specifically, to construct the must-kill relation for a load at l_i, we trace back to its "anchor point" l_j (a preceding load statement) and fetch the must-kill forest $\mathcal{F}$ previously constructed at l_j. $\mathcal{F}$ is then incrementally updated to denote the must-kill relation at l_i by considering the store statements that are present between l_j and l_i. We defer the formal definition of "anchor point" to §4.



Load	Must kill relation	Value
l_{15}	T_1	q0
l_{18}	T_2 ○⟶○ T_1	q3
l_{21}	T_2 ○ ○ T_1	q0, q3

Figure 5: The kill forests and matched incoming values for l_{15}, l_{18}, and l_{21} in Fig. 1a (with l_{13} uncommented).

Example 3.4. The kill forest constructed for Fig. 1a (with Line 13 uncommented) is shown in Fig. 5. After processing the load at l_{15}, we construct tree T_1 to encode that l_{13} must kill l_{10} and l_{12}. When computing the must-kill relation at l_{18} or l_{21}, the analysis first traces back to the "anchor point" l_{15} of both statements and incrementally updates T_1 by T_2. T_2 consists of the only store statement l_{17} lying between l_{15} and l_{18} (or l_{21}), allowing us to avoid processing the entire history of memory stores.

Because $mustKill(l_{17}, l_{13}, l_{18})$ holds but $mustKill(l_{17}, l_{13}, l_{21})$ does not hold, only the must-kill forest constructed for l_{18} has an edge that connects the root of T_2 to the root of T_1. As a result, the analysis concludes that x2 only takes the incoming value from q3, while x3 either takes the value of q3 or q0 (with extra condition ¬c3 that blocks the effect of l_{17}).

Optimized Path-sensitive Reasoning. When the must-kill relation is absent, we fall back to using blocking conditions for enabling strong updates. Existing method computes the condition for load-store matching by exhaustively iterating over the possible points-to targets, e.g., the store item list at l_{17} of Fig. 3 splits over o_4 and o_6, which could be expensive for large points-to set.

We optimize this process by directly computing the "matching condition" between a load and a store statement regardless of the passed through memory objects (detailed in §4).

4 Analysis algorithm

In this section, we detail the algorithm of Tuna for SVFG construction. We first demonstrate how Tuna constructs and incrementally updates the must-kill relation among the store statements using a forest data structure. We then show how the kill forest boosts the performance of performing strong updates in resolving indirect dependencies. Finally, we elaborate on the computation of both

Locations	$l \in Loc$
Objects	$obj \in V_A$
Variables	$v \in V_P \cup V_I \cup V_A$
Guard $\psi \in Cond$	$::= true \mid false \mid C_b \mid \psi_0 \wedge \psi_1 \mid \neg\psi$
`stOp` $\in$ *Stores*	$::= (l_1, p_1, v_1) \in Loc \times V \times V$
`ldOp` $\in$ *Loads*	$::= (l_2, p_2, v_2) \in Loc \times V \times V$
SVFG $\mathbb{G}$	$::= V \longrightarrow 2^{\langle \psi, V\rangle}$

Figure 6: Analysis domain of Tuna. V_P, V_I and V_A denote variables. C_b represents an atomic boolean guard.

$$\frac{}{(\mathbb{G}, \psi) \vdash v := \&a : (\mathbb{G}[o_a \mapsto (\psi, v)], \_)}\ [Alloc]$$

$$\frac{\psi_1 = \psi \wedge \gamma, \quad \psi_2 = \psi \wedge \neg\gamma}{(\mathbb{G}, \psi) \vdash v := ite(\gamma, v_1, v_2) : (\mathbb{G}[v_1 \mapsto (\psi_1, v), v_2 \mapsto (\psi_2, v)], \_)}\ [Ite]$$

$$\frac{\texttt{stOp} = (l, v, k)}{Stores \vdash (l : *v = k) : Stores \cup \{\texttt{stOp}\}}\ [Store]$$

$$\frac{\texttt{ldOp} = (l, v, k), Loads' = Loads \cup \{\texttt{ldOp}\} \quad \mathbb{G}' = \mathbb{G}[val \mapsto (\phi \wedge \psi, k)]_{(val,\phi) \in match(\texttt{ldOp})}}{(\mathbb{G}, \psi, Loads) \vdash (l : k = *v) : (\mathbb{G}', \psi, Loads')}\ [Load]$$

Figure 7: Inference rules of Tuna for path-sensitive data dependence analysis.

must-aliases and may-alias condition path-sensitively, which are crucial to the analysis performance.

4.1 Efficient Strong Updates Via Kill Relation

The core of our analysis domain (c.f. Fig. 6) is the SVFG $\mathbb{G}$: Each node corresponds to either a program variable (V_P), an interface variable (V_I), or a memory allocation site (V_A), and each edge $v_1 \xrightarrow{\gamma} v_2$ indicates a value flow from v_1 to v_2 under the condition γ (c.f. Definition 2.1). For each store operation $l_1 : *p_1 = v_1$ in the program, we record it as $\texttt{stOp} = (l_1, p_1, v_1) \in Stores$. Similarly, each load operation $l_2 : v_2 = *p_2$ is recorded as $\texttt{ldOp} = (l_2, p_2, v_2) \in Loads$.

By processing the program statements following the topological order of the control flow graph (CFG) and applying the inference rules in Fig. 7, Tuna gradually constructs the SVFGs during the path-sensitive data dependence analysis. Specifically, Rule *Alloc* adds an edge $o_a \xrightarrow{\psi} v$ to $\mathbb{G}$ from the allocation site to the variable, Rule *Ite* adds the edges $v_1 \xrightarrow{\psi_1} v$ and $v_2 \xrightarrow{\psi_2} v$ where the conditions ψ_1 and ψ_2 update the precondition ψ with γ, and Rule *Store* simply adds the new store operation `stOp` to *Stores*.

Rule *Load* performs load-store matching to resolve the indirect dependencies and is central to the analysis. By invoking $match(l_1 : k = *v)$, the possible incoming values for k (val) and their associated conditions (ϕ) are obtained, leading to the new value flow edges $val \xrightarrow{\phi \wedge \psi} k$ (Note that the precondition ψ is also conjoined).

The implementation of *match* as shown in Algorithm 1 consists of two stages: the construction of must-kill forest (Lines 2–7) and the computation of matching stores (Lines 8–15).

Must-Kill Forest Construction. As discussed in §3, Tuna utilizes the must-kill relation to spot opportunities for performing strong

Algorithm 1: Load Store Matching In Tuna

```
   Input: Load operation ldOp corresponding to $l_1 : k = *v$
   Global: Map $M$ that associates program location with the
           must kill forest
   Output: A set of value-condition pairs $(val, \phi)$ that may
           flow into $k$
1  Procedure match($l_1 : k = *v$)
2      $l_0 \leftarrow$ getImmAnchorOrEntry($l_1, v$);
3      $F_0 \leftarrow M.at(l_0)$;
4      $S \leftarrow$ getReachableStore($l_0, l_1$);
5      $F_S \leftarrow$ constructKillForest($S, l_1$);
6      $F_1 \leftarrow$ merge($F_S, F_0, l_1$);
7      $M$.set($l_1, F_1$);
8      $TreeRoots \leftarrow$ getTreeRoots($F_1$);
9      Sort $TreeRoots$ using reverse topological CFG order;
10     $c_b \leftarrow$ true, $Res \leftarrow \emptyset$;
11     for $Node\ N\ in\ TreeRoots$ do
12         $(l_x : *v_x = val) \leftarrow N$;
13         $c_m \leftarrow$ getMatchCond($l_1 : k = *v,\ l_x : *v_x = val$);
14         $Res \leftarrow Res \cup \{(val, c_m \wedge c_b)\}$;
15         $c_b \leftarrow c_b \wedge \neg c_m$ ;
16     return $Res$;
17 Procedure constructKillForest ($S, l_1$)
18     $F_S \leftarrow \emptyset$ ;
19     for $(l_2, v_2, val_2) \in S, (l_3, v_3, val_3) \in S, l_2 \neq l_3$ do
20         $N_1 \leftarrow$ addNode($F_S, l_2$);
21         $N_2 \leftarrow$ addNode($F_S, l_3$);
22         if $mustKill(l_2, l_3, l_1)$ then
23             addEdge($F_S, N_1 \rightarrow N_2$);
24     return $F_S$;
25 Procedure merge ($F_S, F_0, l_1$)
26     $F_1 \leftarrow F_S \cup F_0$ ;
27     for $N_1 = (l_2 : *v_2 = \_)$ in getTreeLeaves($F_S$) do
28         for $N_2 = (l_3 : *v_3 = \_)$ in getTreeRoots($F_0$) do
29             if $mustKill(l_2, l_3, l_1)$ then
30                 addEdge($F_1, N_1 \rightarrow N_2$);
31     return $F_1$;
32 Procedure getMatchCond($l_1 : k = *v, l_x : *v_x = val$)
33     return $\gamma(l_x, l_1) \wedge$ aliasCond($v, v_x$);
34 Procedure getReachableStore ($l_0, l_1 : k = *v$)
35     $S \leftarrow \{(l_x, v_x, val) \mid l_x : *v_x = val, reachable(l_0, l_x, l_1),$
36             aliasCond($v, v_x$) $\not\equiv$ false$\}$;
```

updates efficiently. We encode the must-kill relation in a forest data structure that is incrementally updated. Given a load operation at l_1, we seek to find its "immediate anchor point" l_0 such that the kill forest F_0 at l_0 equal the kill forest F_1 at l_1 as long as no store statements are present between l_0 and l_1:

Definition 4.1. The load $l_0 : k_0 = *v_0$ is an anchor point of the load $l_1 : k = *v$ if (1) $l_0 \neq l_1$, l_0 dominates l_1 (2) v_0 and v are must-aliases. l_0 is an immediate anchor point of l_1 if for any anchor point l_3 of l_1 where $l_3 \neq l_0$, l_3 is also an anchor point of l_0.

Based on this intuition, Lines 2–6 of Algorithm 1 construct the kill forest F_1 at the given load statement l_1 by updating the forest F_0 at l_0, the immediate anchor point of l_1 (in case the immediate anchor point does not exist, l_0 will be the function entry and F_0 will be the empty forest).

The update from F_0 to F_1 is conducted by considering the set S of new store operations that could interfere with the load at l_1 (Line 4 and Lines 35–36): Each operation $l_x : *v_x = val$ in S should lie in a control flow path from l_0 to l_1 ($reachable(l_0, l_x, l_1)$), and the stored pointer v_x may alias v at l_1 ($\textbf{aliasCond}(v, v_x) \not\equiv \texttt{false}$, which we detail in §4.2). The relation among the store operations in S is computed and encoded in the forest F_S (Line 5), which further merges with F_0 to produce the final forest F_1 at l_1 (Lines 6–7).

Example 4.1. For the example in Fig. 5, consider applying Algorithm 1 to l_{18} : `x2 = *p`. First, according to Definition 4.1, Line 2 of Algorithm 1 will return l_{15} as the immediate anchor point of l_{18}. The kill forest associated with l_{15} is T_1 in Fig. 5 (obtained by Line 3 of Algorithm 1), $getReachableStore(l_{15}, l_{18})$ returns $S = \{(l_{17}, \texttt{n}, \texttt{q3})\}$, and $constructKillForest(S, l_{18})$ gives rise to T_2 in Fig. 5. Merging T_2, T_1 will add an edge that connects them (i.e., the dotted edges shown in Fig. 5), leading to the final must-kill forest at l_{18}.

The steps to construct a kill forest from a set of store operations and the steps to incrementally update a kill forest by merging are shown in Lines 17–24 and Lines 25–31 of Algorithm 1 respectively. We mitigate the quadratic performance issue in constructing the must-kill relation by only enumerating pairs of store operations within the reachable store set S (Line 21), as opposed to the entire history of store statements. Moreover, merging F_S with F_0 can be achieved efficiently by examining the must-kill relation between leaf nodes of F_S and root nodes of F_0 and adding edges when such relation exists (Lines 27–30).

Perform Load-Store Matching. With the kill forest F_1 constructed, strong updates are enabled readily: Only the store operations at the root nodes of F_1 may provide the incoming value for k, while other nodes are killed by the roots of their containing trees (c.f. Theorem 3.2). Therefore, we have successfully reduced the scope of path-sensitive reasoning from all possible store operations to only the root nodes of F_1.

Lines 11–15 of Algorithm 1 iterate over each root node N (associated with a store $l_x : *v_x = val$) in reverse topological order of the control flow graph to perform load-store matching. The value flow condition for val to reach k consists of two parts. First, as shown by Lines 13, 32–33 of Algorithm 2, a *matching condition* c_m is computed to enforce that the control flow can transfer from l_x to l_1 (denoted by the condition $\gamma(l_x, l_1)$) and the stored pointer should alias the loaded pointer (denoted by the condition $aliasCond(v, v_x)$). Second, a blocking condition c_b is conjoined to block the effects of other root nodes visited before N by taking the negation of their matching conditions (Line 15).

Example 4.2. For the example in Fig. 5, consider applying Algorithm 1 to l_{21} : `x3 = *p`. The immediate anchor point found for l_{21} is still l_{15} but the final kill forest constructed for l_{21} consists of two separate trees $\{T_1, T_2\}$ (c.f. Fig. 5, note that the dotted edge is not present for the forest of l_{21}). Lines 11–15 of Algorithm 1 visit the root node of T_2 (l_{17} : `*n=q3`) and the root node of T_1 (l_{13} : `*p=q0`) in order and add the new value flow edges $\texttt{q3} \xrightarrow{\texttt{c3}} \texttt{x3}$ and $\texttt{q0} \xrightarrow{\neg\texttt{c3}} \texttt{x3}$ (T_2 is blocked when matching the root of T_1).

Algorithm 2: Computation of Path-sensitive Alias Information

```
1  Procedure getPts($\mathbb{G}$, $v$)
2      if $v \in V_A$ then
3          return $\{(v, \texttt{true})\}$;
4      $Pts \leftarrow \emptyset$;
5      for $src \xrightarrow{\phi} v$ in $\mathbb{G}$ do
6          $SrcPts \leftarrow$ getPts($\mathbb{G}, src$);
7          $UpdPts \leftarrow \{(o, \psi \wedge \phi) \mid (o, \psi) \in SrcPts\}$;
8          $Pts \leftarrow (Pts \setminus UpdPts) \cup (UpdPts \setminus Pts) \cup \{(o, \psi_1 \vee \psi_2) \mid (o, \psi_1) \in Pts, (o, \psi_2) \in UpdPts\}$;
9      return $Pts$
10 Procedure getHash($\mathbb{G}$, $v$)
11     if $v \in V_A$ then
12         return hash($\{(v, \texttt{true})\}$);
13     $Vals \leftarrow$ [hash($v$)];
14     for $src \xrightarrow{\phi} v$ in $\mathbb{G}$ do
15         $H_1 \leftarrow$ hash($\phi$);
16         $H_2 \leftarrow$ getHash($\mathbb{G}, src$);
17         $Vals \leftarrow Vals + [H_1, H_2]$;
18     return hashCombine($Vals$);
19 Procedure isMustAlias($\mathbb{G}$, $p$, $q$)
20     return getHash($\mathbb{G}, p$) == getHash($\mathbb{G}, q$);
21 Procedure aliasCond ($v_1, v_2$)
22     $S_1 \leftarrow$ getPts($v_1$);
23     $S_2 \leftarrow$ getPts($v_2$);
24     return $\bigvee\{\phi_1 \wedge \phi_2 \mid (o, \psi_1) \in S_1, (o, \psi_2) \in S_2\}$;
```

4.2 Optimize Path-sensitive Alias Reasoning

Algorithm 1 presented in §4.1 necessitates the computation of both must-aliasing and may-aliasing:

(1) Must-alias information is required in determining the must-kill relation (Definition 3.1) and finding the anchor point of a load operation (Definition 4.1).
(2) May-alias information is required in finding the store candidates for a given load operation (Line 4 of Algorithm 1) and computing the matching condition between a load and a store (Line 33 of Algorithm 1).

To determine whether two pointers may or must point to the same memory location, a path-sensitive analysis normally tracks the related information in *the guarded points-to sets.* For a given variable v and its guarded points to set Pts, $(o, \psi) \in Pts$ indicates that v points to o under the condition ψ. Our goal is to compute both must-alias and may-alias information in a precise, path-sensitive manner, while mitigating the performance issues normally seen in a path-sensitive analysis.

Must Alias Inference Based On Syntactical Equivalence. By definition, `p` and `q` are must-aliases if they always point to the same target under *the logically equivalent* condition, which can be verified by computing and comparing their guarded points-to sets.

However, this straightforward method induces high overhead because it is expensive to compute the guarded points-to set due to the inherent complexity of a path-sensitive analysis. As shown by the procedure getPts of Algorithm 2, to compute Pts for v, we stop at the base case when v is an allocation site (Lines 2–3) and recursively apply the procedure to each incoming edge $src \xrightarrow{\phi} v$ in $\mathbb{G}$. Pts is updated by conjoining the condition ϕ to the guarded points-to set $SrcPts$ of src (Line 7) and merging with the existing result (Line 8). As $\mathbb{G}$ gets larger, both the size of Pts and the condition formula may become explosive, making it challenging to infer must-aliases.

Our approach addresses this challenge by performing an approximate must-alias analysis: p and q are must-aliases if they always point to the same target under *the syntactically equivalent* condition. Moreover, we boost the checking of syntactical equivalence by computing a hash value for each node in $\mathbb{G}$ based on a Merkle hash scheme.

As shown by the procedure getHash in Algorithm 2, the hash value of a node v is aggregated from the hash of the current node (Line 13), the hash of the condition ϕ labeling the incoming edge of v (Line 15), and the hash recursively computed for the incoming node src of v (Line 16). Notice how the structure of getHash mirrors that of getPts, but instead of keeping the points-to targets separate, the hash value compactly aggregates the points-to structure information. Checking for must-aliasing is then easily done by comparing the hash values (Lines 19–20).

Directly Computing The May-Alias Condition. When the must-kill relation does not contribute to enabling strong updates, Tuna falls back to using blocking conditions to support path-sensitive strong updates (Lines 11–15 of Algorithm 1). We aim to further improve the performance of load-store matching in this stage.

In Tuna, we directly compute the set of candidate store statements for a given load (Line 4 of Algorithm 1) and the aliasing condition required for the specific load-store match (Line 33 of Algorithm 1). The computation of the aliasing condition is done by aggregating the conditions from the guarded points-to sets of both variables, as shown in Lines 21–24 of Algorithm 2. By directly computing the may-alias condition, we avoid the repetitive store list traversal for different points-to targets in the existing work (c.f. §3.1).

5 Evaluation

We implement Tuna based on LLVM [22]. Following existing literature on path-sensitive data dependence analysis [3, 14, 33, 34, 39], Tuna replaces each loop in the control flow graph and call graph of the program with a finite unrolling of its body.

Our evaluation aims to answer the following research questions:

- **RQ1**: How much can Tuna boost the performance of indirect dependencies resolving in path-sensitive data dependence analysis? Does the improved performance of Tuna lead to increased state coverage in static analysis?
- **RQ2**: How often are strong updates enabled in Tuna by utilizing the must-kill relation?
- **RQ3**: How do the design choices of Tuna affect its performance?

ID	Program	Size (KLOC)	#Funcs	#Loads	#Stores
0	x264	96	0.8K	28.2K	10.1K
1	omnetpp	134	15.7K	66.9K	21.8K
2	povray	170	2.1K	57.6K	14.2K
3	cactusBSSN	257	3.6K	325.6K	98.5K
4	perlbench	362	5.3K	178.5K	58.5K
5	cam4	407	5.1K	390.9K	174.5K
6	parest	427	66.2K	302.3K	116.3K
7	xalancbmk	520	38.8K	139.7K	49.5K
8	gcc	1304	26.9K	434.5K	108.9K
9	blender	1577	56.8K	538.4K	168.3K
10	libz	41	0.1K	3.2K	1.8K
11	librdkafka	167	2.6K	74.2K	43.1K
12	libtorrent	213	86.2K	83.1K	62K
13	curl	297	6.8K	35.1K	15.7K
14	redis	325	5.7K	75.2K	43.9K
15	bitcoin	716	16.9K	108.1K	85.2K
16	php	1012	18.2K	310K	130.8K
17	ffmpeg	1346	23.2K	584.1K	316.5K
18	qemu	1809	24.4K	160.6K	75.5K
19	mysql	2030	100.5K	719.1K	458.5K

Table 1: Subjects for evaluation. Subjects 0–9 are from the SPEC CPU@2017 benchmark, while Subjects 10–19 are well-known open-source software at their latest versions.

Baseline. We compare Tuna with Falcon [44], the state-of-the-art in path-sensitive data dependence analysis. We obtain the source code of Falcon from its authors and modify it so that we could tune its limitation parameters (detailed later).

Subjects. As shown in Table 1, we perform an evaluation on 10 programs (ID 0–9) from the standard benchmark SPEC CPU@2017 as well as 10 open-source projects (ID 10–19). These subjects are widely used in the evaluation of previous works, cover diverse application domains, and pose significant challenges to static analysis due to their large size (up to several million lines of code).

We also list the number of functions and the number of load / store instructions in the LLVM bitcode representation in Table 1. For instance, to handle *mysql*, over 100K SVFGs will need to be constructed (one for each function) and the load-store matching process needs to be performed over 719K times (for every load instruction), posing significant challenges to the static analyzer.

Environment. All the experiments were performed on a computer with dual 20-core processors Intel(R) Xeon(R) CPU E5-2698 v4@2.20GHz and 500GB physical memory, running Ubuntu-20.04. In the experiments, each static analyzer is run under a single-threaded mode, where the computation resource is limited to four hours of time and 300GB of memory.

Configurations. As discussed in §3.1, in practice, path-sensitive data dependence analysis often sacrifices soundness by limiting its state space exploration. In our experiments, we evaluate the performance of Falcon and Tuna under the same default parameters. Also, we examine whether Tuna can contribute to relaxing the limiting parameters such that the static analyzer could explore a larger state space. Specifically, we will tune the following two parameters:

- `--pts-limit`: The maximum number of points-to targets that are tracked path-sensitively.

- `--vals-limit`: The maximum number of incoming values that could propagate to a load statement.

For all experiments, we set up both Falcon and Tuna to unroll each loop in the program's control flow graphs once and to directly break the back edges in the call graphs. We discuss the effects of varying the loop unrolling factor in §5.3.1.

5.1 Effectiveness of Tuna

To demonstrate how Tuna improves the performance over Falcon, we run them on the selected subjects for the task of SVFG construction. Both of them perform path-sensitive data dependence analysis for resolving indirect dependencies, but differ in how strong updates are enabled.

Fig. 8 demonstrates both the elapsed time and peak memory of Falcon and Tuna under the default configuration (`--pts-limit` is set to 128 and `--vals-limit` is set to 1000). On average, Tuna achieves a 8.0x speed up over Falcon, consumes only 43.3% of memory, and successfully analyzes four subjects where Falcon fails due to out of memory (OOM) issues.

We further run Falcon and Tuna under a set of nine (ID 0–8) configurations `(X, Y)` that are increasingly relaxed, where `X` and `Y` denote the value for `--pts-limit` and `--vals-limit` respectively. We set the values of `(X, Y)` by doubling or halving the default values $(128, 1000)$ (the default configuration is given the ID 4), leading to the more relaxing configurations (ID 5–8) and the more restrictive configurations (ID 0–3). Thus, each tool performs 180 analysis runs for the 20 subjects in total.

The result is shown in Fig. 9. Compared with Falcon, Tuna achieves a speedup ranging from 1.2x to 31.5x, while the peak memory usage is around 39.5%–62.9%. As the figure shows, the improvement of Tuna becomes increasingly significant as the configuration becomes more relaxed. In summary, Falcon fails to complete 38 analysis runs due to timeouts or out-of-memory (OOM) errors, while Tuna fails in only four runs, all of which are also among the runs where Falcon fails.

The capability of Tuna to run path-sensitive data dependence analysis in a more relaxed configuration leads to its better state coverage. Table 2 demonstrates how the size of the constructed SVFGs increases as the configuration is relaxed (we measure the total count of nodes and edges in the SVFG as a proxy for its size). In summary, relaxing the configuration leads to 2.2x more nodes and edges in the SVFGs on average (maximum increase is 13.6x), thus capturing more value flows in the program.

Answer to RQ1: Tuna significantly improves the performance over Falcon in both run time and memory usage, and enables the path-sensitive data dependence analysis to explore a larger state space more efficiently.

The improvements of Tuna lies in its ability to perform strong updates early by utilizing the must-kill relation to avoid the costly path-sensitive analysis. We further study how often such strong updates can be enabled. For this purpose, we measure the number of tree root nodes N_1 traversed in Lines 11–15 of Algorithm 1 versus the total number of store operation candidates N_2 (i.e., the total number of nodes contained in F_1 at Line 6 of Algorithm 1). Our approach reduces the scope of path-sensitive reasoning from

ID	Max num	Delta num	ID	Max num	Delta num
0	1213.2K	11.8K	10	15511.5K	14375.0K
1	5700.3K	1972.3K	11	11429.1K	3720.1K
2	3561.1K	62.2K	12	4994.1K	1394
3	18838.6K	784.8K	13	3587.8K	810.2K
4	68578.4K	54557.6K	14	33784.6K	16640.7K
5	14039.0K	139.4K	15	53378.6K	16596.4K
6	13466.2K	657.7K	16	77480.1K	53135.8K
7	8411.1K	346.8K	17	227345.8K	163744.3K
8	48064.1K	8799.3K	18	23637.3K	442.4K
9	24512.1K	943.5K	19	132747.2K	29657.5K

Table 2: Statistics variation for the SVFGs constructed under the nine different configurations. For the 20 programs (IDs 0 to 19), the column labeled "Max num" represents the maximum value of the total count of nodes and edges in the SVFGs across all configurations, calculated as $max(\Sigma(|N|+|E|))$. Similarly, the column labeled "Delta num" represents the difference between the maximum and minimum total counts of nodes and edges in the SVFGs across all configurations, calculated as $max(\Sigma(|N|+|E|)) - min(\Sigma(|N|+|E|))$.

N_2 nodes to N_1 nodes, and hence the smaller the ratio $\frac{N_1}{N_2}$, the more strong updates are enabled efficiently based on the must-kill relation.

As shown by Fig. 9, averaged across the nine configurations, Tuna only needs to traverse 27.8% of the store candidates in Lines 11–15 of Algorithm 1, indicating that most of the strong updates are efficiently enabled by the must-kill relation in our approach.

Answer to RQ2: Most of the strong updates opportunities are enabled by the must-kill relation in Tuna, reducing the path-sensitive reasoning to only 27.8% of the store candidates on average.

5.2 Design choices of Tuna

In this section, we study the contribution of the key designs of Tuna: must-aliasing computation and incremental kill forest construction.

5.2.1 *Must Aliasing Computation.*

We compute must-aliases based on syntactical equivalence checking of the points-to structure. We compare this choice with the standard approach, i.e., determining must-aliasing by checking the logical equivalence of the guarded points-to sets, which we use the variant Tuna_1 to denote. Across the nine configurations, We found that Tuna_1 is 1.3 to 33.3 times slower (average slowdown is 9.3x) and consumes 1.7 to 6.2 times (average increase is 3.1x) of peak memory compared with Tuna.

5.2.2 *Incremental Kill Forest Construction.*

For a given load operation, we construct the must-kill forest incrementally from the existing forest of its immediate anchor point. We compare with the naive approach where each time the kill forest is constructed from scratch accounting for all reachable store operations, which we use the variant Tuna_2 to denote. We found that Tuna_2 is 1.1 to 2.5 times slower (average slowdown is 1.6x) compared with Tuna (the memory consumption is similar).

Answer to RQ3: Both the designs in must-alias computation and incremental kill forest construction are critical to the performance of Tuna.

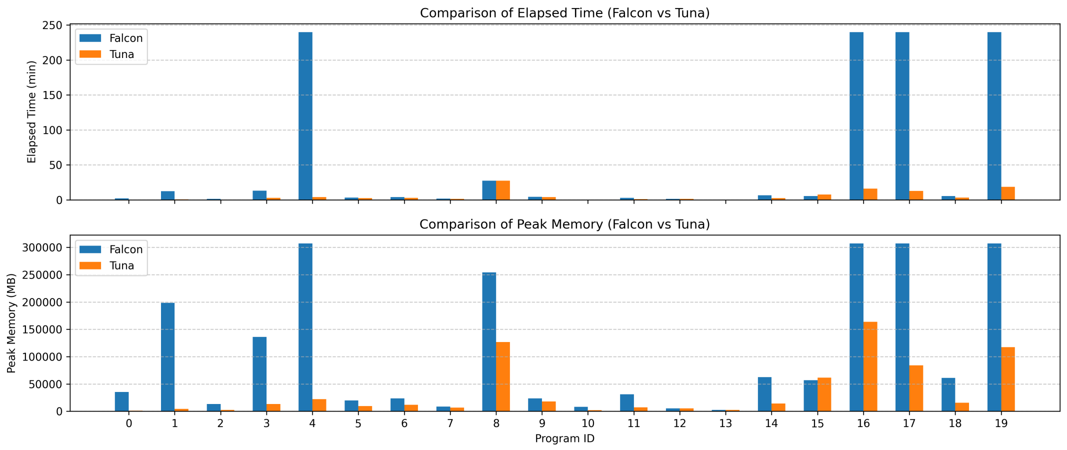


Figure 8: Run time and memory comparison between Falcon and Tuna under the default configuration.

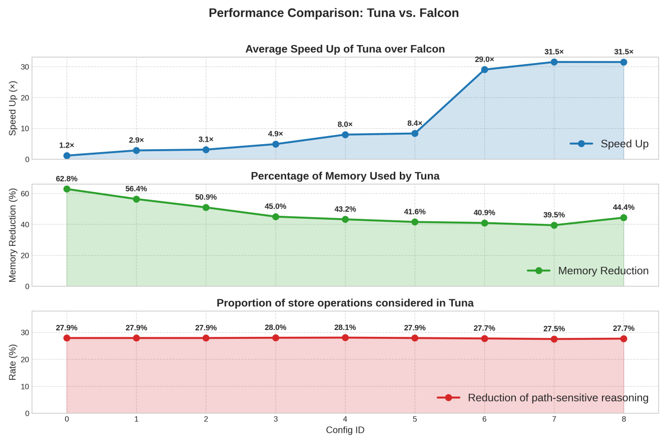


Figure 9: Illustration of the improvements of Tuna over Falcon across a set of nine configurations. We set `-pts-limit` to $[8, 16, 32, 64, 128, 256, 512, 1024, 2048]$ **and `-vals-limit` to** $[62, 125, 250, 500, 1000, 2000, 4000, 8000, 16000]$ **for Config ID 0–8 respectively. The top plot shows the speed up, the middle plot shows the percentage of peak memory usage, and the bottom plot shows proportion of store operation candidates left for path-sensitive reasoning. The data is averaged across the 20 program subjects.**

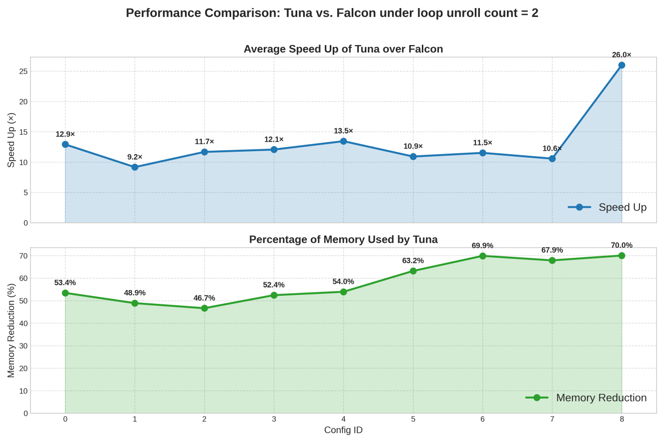


Figure 10: Performance comparison between Tuna and Falcon over the nine configurations when the loop unroll count is set to 2. The configurations are the same as Fig. 9.

5.3 Discussion

5.3.1 The Effects for Loop Unrolling. In the original paper [44], Falcon also directly breaks the back edges in the call graphs, [2] but it sets the loop unroll count to 2. Additionally, the values for other limitation parameters are set to a relatively low number (`--pts-limit` is set to 3 and `--vals-limit` is set to 10 [3]).

We have attempted to rerun all our experiments under the loop unroll count 2 for both Falcon and Tuna. Out of the 180 analysis runs (recall that we have tested nine configurations for each of the 20 subjects), Falcon fails to complete on 111 (61.7%) runs due to either timeout or OOM, while Tuna fails to complete on 58 (32.2%) runs. Tuna still achieves a speedup ranging from 9.2x to 26.0x, while the peak memory usage is around 46.7%–70.0% (c.f. Fig. 10).

Increasing the loop unroll count hinders path-sensitive data dependence analysis by heightening code complexity, which we identify as an orthogonal problem for future investigation.

5.3.2 Can Falcon Benefit from Syntactical Equivalence Checking? Although Tuna uses syntactical equivalence checking, this approach is unsuitable for improving Falcon's performance:

- Unlike our method, Falcon does not infer must-aliases (it computes conditional points-to results through satisfiability checking, c.f. Fig. 3). Thus, the syntactical equivalence checking as in Tuna could not be directly applied.
- We tested an alternative to Falcon's standard satisfiability checking by using a simpler syntactical equivalence check. This method treats any condition as satisfiable unless it is syntactically equivalent to "false." However, this approach was counterproductive,

[2]This was confirmed with the authors of Falcon.
[3]We discover such numbers from the implementation code.

slowing performance by an average of 2.5x because it conservatively approved too many conditions, leading to a significant increase in the number of paths needing analysis.

5.3.3 Effects of Relaxing the Limitation Parameters. In §5.1, we demonstrated that Tuna efficiently performs path-sensitive data dependence analysis in a more relaxed configuration, resulting in value flow graphs that are more "complete" (c.f. Table 2). However, It is important to note that discovering more value flows does not automatically lead to detecting more bugs. Designing an effective bug detector from a value flow graph is a separate and complex challenge, as practical detectors often rely on trade-offs like under-approximate algorithms [18] and require sophisticated search strategies [21] to be effective.

6 Related Work

Data dependence analysis. Data dependence analysis [15] (also termed value flow analysis [9, 33, 38]) is a powerful technique that tracks value propagation in the program by sparsely following the def-use relations and skipping the irrelevant statements. Due to the efficiency benefits, data dependence analysis is widely adopted for statically finding bugs in large and realistic software [26, 33, 39, 43].

Reasoning about pointers and resolving indirect memory dependencies is a major bottleneck in data dependence analysis. Most existing works adopt a layered approach: A global, conservative points-to analysis approximates the def-use information and the subsequent analysis attempts to recover the precision (such as path-sensitivity) [6, 9, 26, 36, 39, 41, 43]. However, they suffer from the spurious value flow propagation induced by the imprecise pre-analysis [33]. Falcon [44] breaks the layered design by fusing path-sensitivity in a modular analysis: An intra-procedurally path-sensitive pointer analysis resolves local dependencies and function side-effects, where the discovery of inter-procedural path conditions is piggybacked on the client (e.g., the specific bug detection task). In this work, Tuna further improves over the state-of-the-art [44] by tackling the problem of inefficient path-sensitive strong updates during the resolving of indirect dependencies.

Path-sensitive static analysis. Path-sensitive static analysis [6, 11, 14, 24, 26, 42, 44] comes in many flavours depending on (1) What analysis facts are tracked in separation (e.g., the path-sensitivity could be added during pointer analysis [44] or only be added at the later stage of bug detection [24]) and (2) What is the strategy to determine the separation (e.g., the descriptor of "paths" could be full-fledged first order logic constraints [42] or conditions collected from a propositional abstraction of the program [44]).

As a good balance between precision and efficiency, we follow the design of Falcon [44]: points-to facts are tracked path-sensitively and modularly, where the path conditions are constructed from a propositional abstraction of the program.

Strong updates in pointer / alias analysis. Strong update [35] refers to overwriting the existing data flow facts (e.g., the points-to target) in a flow-sensitive pointer analysis. While crucial for achieving high precision, enabling strong updates can also significantly hurt the performance [13, 19, 25, 37] (essentially making the flow function non-monotone [2]).

In the context of path-sensitive pointer analysis, the concept of strong update is generalized to accounting for the specific condition under which the old fact is invalidated, where all existing works rely on the formulation of blocking conditions [12, 13, 44]. To the best of our knowledge, Tuna is the first to tackle the efficiency problem of performing strong updates in the path-sensitive analysis for pointer-induced memory dependencies.

Must alias analysis. Must alias analysis has been used to enable strong updates to refine the results of a flow- and context-insensitive may alias analysis, which contributes to improving the precision of null pointer dereference detection [27], typestate verification [16], and termination analysis [30]. These works rely on the conservative may-alias result to determine when strong update is possible, e.g., must-aliasing is discovered when the points-to set contains a single memory object. Meanwhile, must-aliasing information could be discovered in shape analysis [8] such as in recency abstraction [4] and the use of three-valued logic [32]. These approaches distinguish between summary and concrete heap locations by maintaining logical predicates that could lead to exponential complexity.

In contrast, our must-alias inference is precise (path-sensitive, meaning more must-alias pairs are discovered) yet efficient (syntactical equivalence checking based on hash computation), enabling Tuna to perform strong updates in a path-sensitive data dependence analysis more efficiently.

For analyzing higher-order languages, methods like abstract counting [29] and singleness inference [17, 20] have been employed to compute must-aliases, which facilitate advanced compiler optimizations such as closure conversion [29]. These techniques are flow-sensitive, require the singleness property for abstract objects to deduce must-aliases (noting that our must alias analysis allows a variable to reference different memory objects along distinct paths), and are specifically tailored to address unique challenges in higher-order languages, such as implicit control flow [20] and recursion [17]. Consequently, these methods complement our work.

Under-approximate static bug finders. For practical bug finding in large and realistic codebases, static analyzers often sacrifices soundness by exploring an under-approximation of the state space [1, 7, 21, 23, 33]. Tuna follows the same design and our evaluation shows that the performance improvement of Tuna is even more significant when the limitation parameters are more relaxed, showing its potential for enabling the static analyzer to explore a larger state space.

7 Conclusion

We present Tuna, a technique that accelerates path-sensitive data dependence analysis. By identifying must-kill relations between memory stores, Tuna enables efficient strong updates without costly path-sensitive reasoning. Evaluation shows Tuna significantly outperforms the state of the art in runtime, memory usage, and state coverage. The artifact of Tuna is available at https://figshare.com/s/0830800e211d5544f9d9?file=52911869.

Acknowledgments

We thank the reviewers for their valuable comments. This work is funded by research grants from Ant Group and ByteDance.

References

[1] 2022. The Clang Static Analyzer. https://clang-analyzer.llvm.org/. Online; accessed 28-July-2022.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 211–220. doi:10.1145/1368088.1368118

[4] Gogul Balakrishnan and Thomas Reps. 2006. Recency-Abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis* (Seoul, Korea) *(SAS'06)*. Springer-Verlag, Berlin, Heidelberg, 221–239. doi:10.1007/11823230_15

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. doi:10.1145/1646353.1646374

[6] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 275–286. doi:10.1145/2491956.2462186

[7] 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

[8] Bor-Yuh Evan Chang, Cezara Drăgoi, Roman Manevich, Noam Rinetzky, and Xavier Rival. 2020. Shape Analysis. *Foundations and Trends® in Programming Languages* 6, 1–2 (2020), 1–158. doi:10.1561/2500000037

[9] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-Flow Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 480–491. doi:10.1145/1250734.1250789

[10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[11] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 57–68. doi:10.1145/512529.512538

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Fluid Updates: Beyond Strong vs. Weak Updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems* (Paphos, Cyprus) *(ESOP'10)*. Springer-Verlag, Berlin, Heidelberg, 246–266.

[13] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 567–577.

[14] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 72–82. doi:10.1109/ICSE.2019.00025

[15] 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

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

[17] Kimball Germane and Jay McCarthy. 2021. Newly-single and loving it: improving higher-order must-alias analysis with heap fragments. *Proc. ACM Program. Lang.* 5, ICFP, Article 96 (Aug. 2021), 28 pages. doi:10.1145/3473601

[18] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (jan 2019), 29 pages. doi:10.1145/3290370

[19] 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.

[20] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. 1998. Single and loving it: must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 329–341. doi:10.1145/268946.268973

[21] Yoonseok Ko and Hakjoo Oh. 2023. Learning to Boost Disjunctive Static Bug-Finders. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1097–1109. doi:10.1109/ICSE48619.2023.00099

[22] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[23] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (apr 2022), 27 pages. doi:10.1145/3527325

[24] Wei Le and Mary Lou Soffa. 2008. Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 272–282. doi:10.1145/1453101.1453137

[25] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 3–16. doi:10.1145/1926385.1926389

[26] V Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 317–326.

[27] Xiaodong Ma, Ji Wang, and Wei Dong. 2008. Computing Must and May Alias to Detect Null Pointer Dereference. In *Leveraging Applications of Formal Methods, Verification and Validation*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 252–261.

[28] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: explaining program failures via postmortem static analysis. *SIGSOFT Softw. Eng. Notes* 29, 6 (Oct. 2004), 63–72. doi:10.1145/1041685.1029907

[29] Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. *SIGPLAN Not.* 41, 9 (Sept. 2006), 13–25. doi:10.1145/1160074.1159807

[30] undefinedurica Nikolić and Fausto Spoto. 2012. Definite expression aliasing analysis for java bytecode. In *Proceedings of the 9th International Conference on Theoretical Aspects of Computing* (Bangalore, India) *(ICTAC'12)*. Springer-Verlag, Berlin, Heidelberg, 74–89. doi:10.1007/978-3-642-32943-2_6

[31] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. doi:10.1145/199448.199462

[32] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (May 2002), 217–298.

[33] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. 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* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706. doi:10.1145/3192366.3192418

[34] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 930–943. doi:10.1145/3453483.3454086

[35] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (apr 2015), 1–69. doi:10.1561/2500000014

[36] Gregor Snelting, Torsten Robschink, and Jens Krinke. 2006. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.* 15, 4 (oct 2006), 410–457. doi:10.1145/1178625.1178628

[37] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 460–473. doi:10.1145/2950290.2950296

[38] 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 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[39] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 254–264. doi:10.1145/2338965.2336784

[40] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems* (Kenting, Taiwan) *(APLAS'11)*. Springer-Verlag, Berlin, Heidelberg, 155–171.

[41] Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned Memory Models for Program Analysis. In *Verification, Model Checking, and Abstract Interpretation*, Ahmed Bouajjani and David Monniaux (Eds.). Springer International Publishing, Cham, 539–558.

[42] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 351–363. doi:10.1145/1040305.1040334

[43] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering* (, Gothenburg, Sweden,) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 327–337. doi:10.1145/3180155.3180178

[44] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2024. Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis. *Proc. ACM Program. Lang.* 8, PLDI, Article 170 (June 2024), 26 pages. doi:10.1145/3656400