# Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis

PEISEN YAO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JINGUO ZHOU, Ant Group, China
XIAO XIAO, Ant Group, China
QINGKAI SHI, The State Key Laboratory for Novel Software Technology, Nanjing University, China
RONGXIN WU, Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen University, China
CHARLES ZHANG, The Hong Kong University of Science and Technology, China

This paper presents a scalable path- and context-sensitive data dependence analysis. The key is to address the aliasing-path-explosion problem when enforcing a path-sensitive memory model. Specifically, our approach decomposes the computational efforts of disjunctive reasoning into 1) a context- and semi-path-sensitive analysis that concisely summarizes data dependence as the symbolic and storeless value-flow graphs, and 2) a demand-driven phase that resolves transitive data dependence over the graphs, piggybacking the computation of fully path-sensitive pointer information with the resolution of data dependence of interest. We have applied the approach to two clients, namely thin slicing and value-flow bug finding. Using a suite of 16 C/C++ programs ranging from 13 KLoC to 8 MLoC, we compare our techniques against a diverse group of state-of-the-art analyses, illustrating the significant precision and scalability advantages of our approach.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: data dependence analysis, path-sensitive analysis

**ACM Reference Format:**
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. https://doi.org/10.1145/3656400

## 1 INTRODUCTION

Data dependence analysis identifies the def-use information in a program, which is critical for various analysis clients such as change-impact analysis [2, 66], program slicing [52, 80], and memory disambiguation [1, 106]. However, the presence of pointers and references obscures this information. The analysis must cut through the tangle of aliasing to reason about data dependence.

Path sensitivity is a common axis for pursuing precision but is stunningly challenging for data dependence analysis. In particular, maintaining a path-sensitive memory model can suffer from the "aliasing-path-explosion" problem. For example, at a load statement $p = *x$, we need to track the path condition of this statement and the path conditions under which $x$ points to different memory objects. Each load or store statement may access hundreds of memory objects; each memory object

Authors' addresses: Peisen Yao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, pyao@cse.ust.hk; Jinguo Zhou, Ant Group, China, jinguo.zjg@antfin.com; Xiao Xiao, Ant Group, China, xx@antgroup.com; Qingkai Shi, The State Key Laboratory for Novel Software Technology, Nanjing University, China, qingkaishi@gmail.com; Rongxin Wu, Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen University, China, wurongxin@xmu.edu.cn; Charles Zhang, The Hong Kong University of Science and Technology, China, charlesz@cse.ust.hk.


ACM 2475-1421/2024/6-ART170
https://doi.org/10.1145/3656400

may be accessed at hundreds or thousands of statements; and the number of program paths under which the statements execute is exponential. Consequently, the number of disjunctive cases to track becomes extremely large, far too many to enable a scalable analysis.

### 1.1 Existing Efforts

Existing solutions to building path-sensitive memory models can be classified into two major categories. The *bootstrapped approach* maintains a path-sensitive memory model without a priori points-to analysis, such as symbolic execution [7] and shape analysis [70]. This approach uses various logic to generate formulas that encode the entire history of memory writes and reads, which allows for establishing correlations between variables automatically. However, the approach offloads the aliasing-path-explosion problem to the constraint solvers and encodes constraints following control-flow paths, regardless of their relevance to the data dependence of interests. Such "dense" analysis is known to have performance problems. For instance, Focal [43], a state-of-the-art backward symbolic executor, takes approximately 230 hours to answer on-demand queries for a program with nearly 33 KLoC.

Alternatively, the *layered approach* uses an independent, auxiliary points-to analysis to approximate the def-use information, which is then used to guide the subsequent path-sensitive analysis [6, 95, 99]. The idea of leveraging pre-computed points-to information has advanced flow- and context-sensitive pointer and typestate analyses, via sparsification [30, 83], pruning [21], and partitioning [40]. For path-sensitive analysis, the use of auxiliary pointer analysis has also shown promise in accelerating software model checking via partitioned model models [95], as well as guiding the demand-driven, symbolic exploration of program paths [6, 99].

Unfortunately, the existing studies on the layered approach suffer from two limitations that hinder scalability. First, they primarily rely on flow- and path-insensitive pointer analysis as the auxiliary pre-analysis and attempt to recover a path-sensitive memory model in the subsequent analysis. However, the imprecise pre-analysis can lead to spurious and redundant propagation of pointer information in the subsequent analysis [99]. Second, they utilize an explicit points-to abstraction in the auxiliary pre-analysis, requiring the subsequent analysis to perform cast-splitting over the points-to sets while handling indirect loads/stores. This cast-splitting process has been a major cause of disjunctive case explosion in previous path-sensitive analyses [6, 59, 99] due to the potentially large size of points-to sets.

### 1.2 This Work

This paper presents Falcon, a fused approach to path-sensitive data dependence analysis. Our key insight is two-fold. First, a data dependence relation induced by pointer expressions can be identified without knowing the concrete memory objects referenced by the pointers. Second, many program paths qualifying data dependence relations are redundant and can be symbolically identified and merged while preserving precision.

Based on this insight, we first introduce an all-program-points pointer analysis that builds guarded and storeless value-flow graphs of a program without computing exhaustive points-to sets. First, it offers the key benefits of path sensitivity, state pruning, merging, and simplification, by using lightweight semi-decision procedures over a propositional abstraction of the program. Second, to achieve context sensitivity without expensive summary cloning, it selectively clones access-path expressions that are rooted at a function parameter and incur side effects, thereby enabling local reasoning of value flows instead of global reasoning about the entire heap.

Then, our client analysis can use the pre-computed value-flow graphs as "conduits" to track transitive data dependence on demand. Importantly, the guarded graph edges concisely merge value flows induced by different memory objects and program paths, which offers two benefits. First,

the client can sparsely track the dependence by following the edges, piggybacking the computation of fully path-sensitive pointer information and the resolution of data dependence of interest. Second, the client does not need to perform explicit cast-splitting over the points-to sets when handling indirect loads/stores, alleviating a major source of case explosion in previous (on-demand) path-sensitive analyses [6, 59, 99].

In summary, we decompose the burden of aliasing-path-explosion into the client-independent pre-analysis that builds value-flow graphs and the client-specific data dependence tracking. Crucially, the guarded and storeless graphs enable a path-sensitive memory model upfront and allow us to separate the reasoning task about "how the values flow through different memory objects" from the task of answering queries about transitive dependence. Specifically, there are two novel and critical features in our algorithm itself:

- The analysis for building value-flow graphs is both on-the-fly sparse and path-sensitive, in that it computes the heap def-use chains incrementally along with the path-sensitive pointer information discovered. The SPAS algorithm [88] is the only previous pointer analysis with the same property, but they achieve incremental sparsity through the level-by-level analysis [101] and, thus, is exhaustive.
- When answering demand data dependence queries, our analysis can stop as soon as enough evidence is gathered, without trying to find all pointed-to by memory objects. The previous analyses [77, 78, 98, 106] can answer demand alias queries via different storeless representations [41]. However, none of the techniques can introduce path sensitivity.

Although the dependence information is significantly more limited in providing complex heap invariants than a full-blown shape analysis, it is sufficient for many interesting applications. One such application is thin slicing [52, 80, 92], which aids in program debugging and understanding using the statements that affect the value of a variable. Another example is value-flow bug finding [74, 87, 99], which hunts memory safety bugs by tracking the flow of pointer values. Additionally, we have utilized Falcon to optimize container manipulation programs [94], guide directed fuzzing [35], and infer status code specifications [91].

One thing that Falcon is not, at least in its current form, a sound verification framework for unbounded programs. We unroll each loop and function call in the control flow graph and the call graph twice and follow the assumption that distinct parameters are not aliases with each other [54, 97]. We leave further investigation of unbounded programs as future work.

To sum up, we make the following key contributions in this paper:

- We identify and discuss the major challenges in scaling path-sensitive data dependence analysis, and particularly, in enforcing a path-sensitive memory abstraction.
- We introduce a precise and efficient approach to constructing value-flow graphs. Based on the approach, we present a path-sensitive data dependence analysis that gracefully scales to multi-million-line code bases with the precision of full path sensitivity.
- We demonstrate the utility of our techniques with two clients, thin slicing-based program understanding and value-flow bug finding. We conduct experiments on 16 real-world C/C++ programs ranging from 13 KLoC to 8 MLoC, showcasing the significant precision and scalability benefits of our approach.

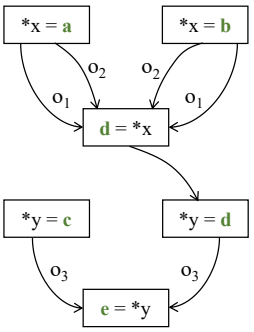
## 2 OVERVIEW

We use the example in Fig. 1 to motivate the path-sensitive data dependence analysis, highlight its challenges, and explain the essence of our approach.

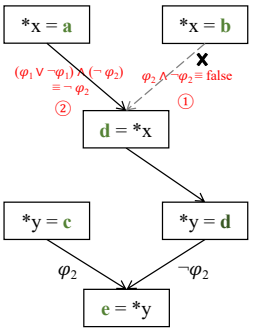
**Importance of Path-Sensitive Data Dependence**. Suppose we need to detect double-free bugs in the program shown in Fig. 1(a). In this program, there are two memory deallocation statements:

```
1  int *foo(int **y) {   //o3
2    int **x; *y = c;
3    if (φ1) { x = malloc();  } //o1
4    else { x = malloc();  }    //o2
5    *x = a;
6    if (φ2) {
7      *x = b;  free(a);
8    } else {
9      int *d = *x;  *y = d;
10   }
11   int *e = *y;
12   if (φ2) { free(e); }
13 }
```

(a) Code snippet




(b) Conventional value-flow graph




(c) Our value-flow graph

Fig. 1. Comparing the conventional value-flow graph and our value-flow graph for answering demand dependence queries. In (b), the label on an edge represents a memory object. In (c), the label on an edge is a path condition.

*free(a)* and *free(e)*. Observe that the value of $a$ can flow to $e$ only under the condition $\neg\phi_2$. Thus, the program is safe, because the deallocations execute under the condition $\phi_2$.

Assume we only approximate the data dependence information with a path-insensitive pointer analysis, which leads to the conclusion that $e$ is data-dependent on $a$. Now, suppose that the bug-finding phase only *partially* tracks path correlations of the deallocation statements and is unaware of the path condition of the data dependence relation. Observe that the path conditions for the two statements *free(a)* and *free(e)* are both $\varphi_2$, which do not conflict. Here, if taking $\varphi_2$ as the path condition for a double-free vulnerability, our analysis would raise a false alarm because the condition for $e$ to be data dependent on $a$ is $\neg\varphi_2$. To summarize, the imprecise data dependence information caused by the pointer analysis is passed on to the clients, hurting the precision.

**Problem of Aliasing-Path-Explosion**. However, obtaining path-sensitive data dependence information is far from trivial. The key challenge is maintaining a path-sensitive memory model, which requires reasoning about numerous disjunctive cases when tracking path-sensitive pointer information. For instance, at a load statement $p = *x$ or store statement $*x = q$, we need to track the path condition of the statement and the conditions under which $x$ points to different memory objects. The transfer function has to store and propagate an enormous amount of path-sensitive information because (1) the number of pointers can be huge, each of which may access hundreds of memory objects, (2) each memory object may be visited at hundreds or thousands of program statements, and (3) the number of program paths (through different calling contexts and control-flow paths) under which the statements execute is exponential. We refer to this problem as *aliasing-path-explosion*: analyzing path-sensitive data dependence information can require reasoning about an excessive number of paths [6].

**The State-of-the-Art**. A recent analysis [99] has leveraged the idea of *sparsity* to refine flow-insensitive results and make them path-sensitive on demand. It first constructs the flow-insensitive def-use chains with Andersen analysis. These def-use chains then enable the subsequent on-demand and path-sensitive analysis [99]. For instance, as shown in Fig. 1(b), the two edges between $*x = a$ and $d = *x$ state that the value of pointer $a$ can flow to the pointer $d$ via the memory objects $o_1$ or $o_2$, implying that $d$ may be data-dependent on $a$.

However, the flow-insensitive pre-analysis does not retain path information. Consequently, when answering demand queries, the primary analysis still suffers from aliasing-path-explosion. For example, if a client asks "what are the set of variables $e$ may be data-dependent on?", we perform an on-demand backward traversal from $e = *y$ to $*x = a, *x = b, *y = c$, respectively. In the

worst case, this graph traversal requires searching five paths and solving five path constraints. This number of paths exceeds the total number of paths in the program (which is four), meaning that the aliasing-path-explosion can be even worse than the well-known scalability problem caused by conditional branching in symbolic execution [6].

In essence, existing sparse analysis can use a pre-analysis to identify the relevant memory objects (as in Fig. 1(b)). However, the pre-analysis can only reduce the number of memory objects to track, but not the number of value flow paths going through those relevant memory objects, because it is unaware of the path conditions qualifying the value flows. When pursuing path sensitivity, the primary analysis has to introduce sensitivity to all potentially polluting statements, which can leave the overhead out of control. Consequently, the primary path-sensitive analysis cannot avoid aliasing-path-explosion.

**Our Approach**. To mitigate the problem, our key insight is two-fold. First, we can identify a data dependence relation induced by pointer expressions without knowledge of the specific memory objects referenced by the pointers. Second, many program paths qualifying a dependence relation are redundant and can be symbolically identified and merged while maintaining precision. For example, intuitively, the variable $d$ may be data-dependent $a$, regardless of whether $x$ points to $o_1$ or $o_2$. Besides, only the branching condition $\varphi_2$ affects the dependence relation, and the truth value of the branching condition $\varphi_1$ is irrelevant. However, previous work [6, 99] *must* separate the two edges and label them with different memory objects to preserve the capability of precision refinement based on the memory objects.

Based on the insight, we present a fused approach to path-sensitive data dependence analysis. At a high level, our approach works in two phases.

In the first phase, we introduce a whole-program-point but lazy pointer analysis to build guarded and storeless value-flow graphs (§ 4). The analysis operates over a *propositional abstraction* of the program, taking advantage of lightweight, Boolean-level semi-decision procedures to effectively prune false value-flow edges, merge duplicate edges, and simplify the guards. For example, as illustrated in Fig. 1(c), our analysis achieves two significant benefits : ① efficiently pruning many infeasible value flows, and ② effectively merging and simplifying path constraints when merging value-flow edges. Additionally, to build interprocedural value-flow graphs, it only clones the memory access path expressions rooted at a function parameter and incurring side effects, avoiding the computation of a whole-program image of the heap.

In the second phase, we can answer demand transitive data dependence queries over the graphs (§ 5). Specifically, the client analysis can piggyback the computation of fully path-sensitive pointer information with the resolution of transitive dependence. During this process, it can collect the constraints and solve them using a full-featured SMT solver. For example, consider Fig. 1(c), where the memory objects pointed-to by $x$ and $y$ are implicit. To determine the values that $e$ is data-dependent on, we perform a backward graph traversal, only requiring to traverse two paths: from $e$ to $c$ and from $e$ to $a$. To detect double-free bugs, we perform a forward traverse from $a$ to $e$, collecting the guard qualifying the edges (i.e., $\neg\varphi_2 \wedge \neg\varphi_2$). We also collect the path conditions of the two statements *free(a)* and *free(e)* (i.e., $\varphi_2 \wedge \varphi_2$). As can be seen, we can eliminate the false positive because $\neg\varphi_2 \wedge \varphi_2$ is unsatisfiable.

Compared with the state-of-the-art that constructs value-flow graphs via a flow-insensitive points-to analysis and labels each edge with a memory object [83, 99], our analysis has two major benefits. First, it catches the path correlations among memory operations, thus pruning more infeasible edges than path-insensitive analyses. Second, it concisely merges and simplifies the guards of valu-flow edges, which can be induced by different memory objects and control-flow paths.

$$
\begin{aligned}
&\textit{Program}\ \ P := F+\\
&\textit{Function}\ \ F := f(v_1, v_2, ...)\ \{\ S;\ \}\\
&\textit{Statement}\ \ S := v_1 = \&v_2 \mid v_1 = v_2 \mid v = \phi((\varphi_1, v_1), (\varphi_2, v_2), \ldots) \mid v_1 = *v_2 \mid *v_1 = v_2\\
&\qquad\qquad \mid r = \textit{call}\ f(v_1, v_2, ...) \mid \textit{return}\ v \mid \textit{if}\ (v)\ \{\ S_1;\ \}\ \textit{else}\ \{\ S_2;\ \} \mid S_1; S_2
\end{aligned}
$$

Fig. 2. The syntax of the language.

$$
\begin{aligned}
&\textit{Labels}\ \ \ell \in \mathcal{L} \quad \textit{Variables}\ v \in \mathcal{V} \quad \textit{Objects}\ o \in \mathcal{O} \quad \textit{Guard}\ \ \varphi(v_1, \ldots) \in \Psi\\
&\textit{Environment}\ \ \mathbb{E} := \mathcal{V} \to 2^{(\Psi, O)} \qquad \textit{Store}\ \ \mathbb{S} := \mathcal{O} \to 2^{(\Psi, \mathcal{L}, \mathcal{V})}
\end{aligned}
$$

Fig. 3. The abstract domains.

## 3 PROBLEM FORMULATION

**Bounded Programs**. We formalize our analysis with a simple language in Fig. 2. Programs are in the static single assignment form. We use $v_i$ to denote a program variable. In the $\phi$-assignment, $\varphi_i$ is the gated function for each $v_i$, which means $v = v_i$ if and only if $\varphi_i$ is satisfied. These gated functions can be computed in almost linear time [67]. Without loss of generality, we assume that each function has only one return statement.

Our analysis targets bounded programs to aid in understanding and detecting bugs in large, real-world software. We unroll each loop and function call in the control flow graph and the call graph twice, bounding the heap size that can be accessed. We also follow the assumption in the prior study [97] that distinct parameters are not aliases with each other (§ 4.2). Hence, in its current form, our approach is a not sound verification framework for unbounded programs.

**Abstract Domains**. The symbols and abstract domains are listed in Fig. 3. A label $\ell \in \mathcal{L}$ indicates the position of a statement in the control flow graph. A guard condition $\varphi(v_1, \ldots)$ is a first-order formula over $\mathcal{V}$. In the rest of the paper, we omit the variables in $\varphi$ for simplicity, which is aligned with [17, 18]. We factor the abstract domain to the points-to environment $\mathbb{E}$ and abstract store $\mathbb{S}$, where $\mathbb{E}(v) = \{(\varphi, o)\}$ means that the pointer $v$ points to the memory object $o$ under the condition $\varphi$, and $\mathbb{S}(o) = \{(\varphi, \ell, v)\}$ states that the memory object contains the value $v$, which is stored in the memory object at the program point $\ell$ on the condition $\varphi$. For simplicity, we define the operation $\Pi_\varphi$ to query $\mathbb{E}$ and $\mathbb{S}$ under a condition $\varphi$. Formally,

$$\Pi_\varphi(\mathbb{S}(o)) = \{(\pi \wedge \varphi, \ell, v) \mid (\pi, \ell, v) \in \mathbb{S}(o)\}$$
$$\Pi_\varphi(\mathbb{E}(v)) = \{(\pi \wedge \varphi, o) \mid (\pi, o) \in \mathbb{E}(v)\}$$

In addition, we also define a special set-union operator for $\mathbb{E}$ as below, i.e., $\uplus$, such that we merge points-to environment for the same memory object. That is, $\forall(\pi, o) \in \mathbb{E}(v)$ and $(\pi', o) \in \mathbb{E}'(v)$, we merge the abstract values such that $(\pi \vee \pi', o) \in \mathbb{E}(v) \uplus \mathbb{E}'(v)$. We abuse the union operators, i.e., $\mathbb{E}_1 \uplus \mathbb{E}_2$ and $\mathbb{S}_1 \cup \mathbb{S}_2$, to mean we merge the maps by applying $\uplus$ or $\cup$ to the values of each key.

**Value-Flow Graph**. Intuitively, a value $q$ flows to $p$ if $q$ is assigned to $p$ directly (via an assignment, such as $p = q$) or indirectly (via pointer dereferences, such as $*x = q; y = x; p = *y;$). In this work, we use value-flow graphs as below:

*Definition 3.1.* (Guarded and Storeless Value-Flow Graph) A guarded and storeless value-flow graph is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, C)$, where $\mathcal{N}$, $\mathcal{E}$, and $C$ are defined as following:

- $\mathcal{N}$ is a set of nodes, each denoted by $v@\ell$, meaning that the variable $v$ is defined or used at a program location $\ell$.

- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges, each of which represents a value-flow relation. $(v_1@\ell_1, v_2@\ell_2) \in \mathcal{E}$ means that the value $v_1@\ell_1$ flows to $v_2@\ell_2$.
- $C$ maps each edge in the graph to a condition $\varphi$, meaning the value-flow relation holds only when the condition is satisfied.

The graph is *guarded* because a value-flow edge is labeled with the condition $\varphi$ qualifying the edge. When establishing these edges, it is important to track the values stored at a store statement and determine the values that can be loaded at a load statement. We will explain this process in the following section. The graph is *storeless* because, unlike the def-use graph, memory SSA, or SVFG in [30, 84, 88, 99, 101], we do not label the (indirectly) accessed memory objects at a load or store, and the subsequent client analysis does not require case-splitting over the points-to sets.

**Problem Statement**. The central task in tracking path-sensitive data dependence is to enforce a path-sensitive memory model, which suffers from the aliasing-path-explosion problem (§ 2). A crucial question that drives this work is: *what should be the relationship between the pointer analysis (if any) and the path-sensitive memory model*? Without an auxiliary pointer analysis, the bootstrapped approach, such as symbolic execution, rests on purely symbolic encoding to capture the correlations of statements but struggles to scale. When using an (exhaustive, path-insensitive) auxiliary points-to analysis, as in the layered approach, there is a discrepancy between the memory abstraction established by the pointer analysis and the one enforced by the primary path-sensitive analysis. Worse still, the primary analysis may have to regain precision for a vast number of program statements and memory objects, resulting in unmanageable overhead.

To summarize, there is a tension in existing works between tracking path-sensitive pointer information too early (as in the bootstrapped approach)–which results in an overwhelming cost of symbolic reasoning –and tracking it too late (as in the layered approach)–which limits the benefits of path sensitivity because of spurious and redundant propagation of pointer information [6, 99]. To balance this tension, our fused approach builds a symbolic storeless representation for pointer expressions (§ 4), which concisely summarizes how values flow in and out of the memory, enables a path-sensitive memory model upfront, and boosts the subsequent on-demand, fully path-sensitive tracking of transitive data dependence (§ 5).

## 4 BUILDING GUARDED AND STORELESS VFG

In this section, we discuss the details of our approach in three parts: the intra-procedural analysis (§4.1), the inter-procedural analysis (§4.2), and the construction of the value-flow graph alongside the analysis (§4.3).

### 4.1 Intraprocedural Analysis

This section presents our intraprocedural analysis to compute $\mathbb{E}$ and $\mathbb{S}$ so that we can establish indirect value flows by querying what values can be loaded at a load statement. We first define the abstract transformers, which enable a conventional data-flow analysis. At the end of § 4.1.1, we summarize the challenges for optimization, which are addressed in § 4.1.2 and § 4.1.3.

*4.1.1 Abstract Transformers.* Fig. 4 lists the rules for analyzing the basic statements. Each rule is of the form $\mathbb{E}, \mathbb{S} \vdash \ell, \varphi : stmt : \mathbb{E}', \mathbb{S}'$, which states that given the current points-to environment $\mathbb{E}$, abstract store $\mathbb{S}$, and path condition $\varphi$, the statement $stmt$ at the program point $\ell$ produces new points-to environment $\mathbb{E}'$ and/or abstract store $\mathbb{S}'$. In these rules, we use $\mathbb{E}[p \mapsto \{\dots\}]$ and $\mathbb{S}[o \mapsto \{\dots\}]$ to mean that the maps are updated by binding the pointer $p$ and the memory object $o$ to new abstract values, respectively.

Rule ADDR creates memory objects at allocation sites. Rule COPY updates the points-to environment $\mathbb{E}$ of one variable $p$ via the other $q$. Note that since we assume the code is in the SSA form,

$$\frac{\mathbb{E}' = \mathbb{E}[p \mapsto (\varphi, alloc_a)]}{\mathbb{E}, \mathbb{S} \vdash \ell, \varphi : p = \&a : \mathbb{E}', \mathbb{S}}\ \textsc{addr} \qquad \frac{\mathbb{E}' = \mathbb{E}[p \mapsto \Pi_\varphi(\mathbb{E}(q))]}{\mathbb{E}, \mathbb{S} \vdash \ell, \varphi : p = q : \mathbb{E}', \mathbb{S}}\ \textsc{copy} \qquad \frac{\mathbb{E}' = \mathbb{E}[p \mapsto \uplus_{i=1}^{n} \Pi_{\varphi_i}(\mathbb{E}(p_i))]}{\mathbb{E}, \mathbb{S} \vdash \ell, \varphi : p = \phi((\varphi_1, p_1), ..., (\varphi_n, p_n)) : \mathbb{E}', \mathbb{S}}\ \textsc{phi}$$

$$\frac{\begin{array}{l}|\Pi_\varphi(\mathbb{E}(x))| > 1 \Rightarrow \mathbb{S}' = \mathbb{S}[o \mapsto \{(\pi, \ell, q) \mid (\pi, o) \in \Pi_\varphi(\mathbb{E}(x))\} \cup \mathbb{S}(o)] \\ |\Pi_\varphi(\mathbb{E}(x))| = 1 \Rightarrow \mathbb{S}' = \mathbb{S}[o \mapsto \{(\pi, \ell, q) \mid (\pi, o) \in \Pi_\varphi(\mathbb{E}(x))\}]\end{array}}{\mathbb{E}, \mathbb{S} \vdash \ell, \varphi : *x = q : \mathbb{E}, \mathbb{S}'}\ \textsc{store}$$

$$\frac{\mathbb{E}' = \mathbb{E}[p \mapsto \uplus_{(\pi,o) \in \Pi_\varphi(\mathbb{E}(y))} \bigcup_{(\varphi,\ell',v) \in \Pi_\pi(\mathbb{S}(o))} \Pi_\varphi(\mathbb{E}(v))]}{\mathbb{E}, \mathbb{S} \vdash \ell, \varphi : p = *y : \mathbb{E}', \mathbb{S}}\ \textsc{load} \qquad \frac{\mathbb{E}, \mathbb{S} \vdash S_1 : \mathbb{E}', \mathbb{S}' \qquad \mathbb{E}', \mathbb{S}' \vdash S_2 : \mathbb{E}'', \mathbb{S}''}{\mathbb{E}, \mathbb{S} \vdash S_1; S_2 : \mathbb{E}'', \mathbb{S}''}\ \textsc{sequencing}$$

$$\frac{\mathbb{E}, \mathbb{S} \vdash S_1 : \mathbb{E}', \mathbb{S}' \qquad \mathbb{E}, \mathbb{S} \vdash S_2 : \mathbb{E}'', \mathbb{S}''}{\mathbb{E}, \mathbb{S} \vdash if\ (v)\ \{\ S_1;\ \}\ else\ \{\ S_2;\ \} : \mathbb{E}' \uplus \mathbb{E}'', \mathbb{S}' \cup \mathbb{S}''}\ \textsc{branching}$$

Fig. 4. Basic rules for updating $\mathbb{E}$ and $\mathbb{S}$.

$$\textsc{vflow}(\ell_1, \varphi_1 : *x = q;\ \cdots;\ \ell_2, \varphi_2 : p = *y) \frac{\begin{array}{c}\forall (\pi_i, o_i) \in \Pi_{\varphi_2}(\mathbb{E}(y)) \\ \forall (\varphi_i, \ell_1, q) \in \Pi_{\pi_i}(\mathbb{S}(o_i))\end{array}}{(q@\ell_1, p@\ell_2) \in \mathcal{E}, C((q@\ell_1, p@\ell_2)) = \vee \varphi_i}$$

Fig. 5. Building indirect value-flow edges.

every top-level variable has a single definition. Thus, Rule ADDR and Rule COPY perform strong updates. Rule PHI merges the points-to environment at the joint point of multiple paths. Basically, Rule COPY and Rule PHI are self-explanatory. Thus, we will focus on the STORE and LOAD rules.

Rule STORE processes a store statement $*x = q$ under path condition $\varphi$, resulting in new configurations of the abstract store $\mathbb{S}$. We first query the memory objects $x$ may point-to, denoted as $\Pi_\varphi(\mathbb{E}(x))$. For all guarded memory objects $(\pi, o) \in \Pi_\varphi(\mathbb{E}(x))$, we update the abstract store $\mathbb{S}$ to record the values that $o$ may hold. Following conventional singleton-based algorithms, if $x$ points to at most one concrete memory object, we can perform an indirect strong update, which kills other values held by the memory object $o$ [19, 29, 30, 45, 101].

Given a load statement $p = *y$ under path condition $\varphi$ at program location $\ell$, we apply Rule LOAD as follows. Similar to the STORE rule, we query the memory objects that $y$ may point-to under the condition $\varphi$, denoted $\Pi_\varphi(\mathbb{E}(y))$. We then fetch the values from each memory object $(\pi, o) \in \Pi_\varphi(\mathbb{E}(y))$, denoted as $\Pi_\pi(\mathbb{S}(o))$. Finally, for every $(\varphi, \ell', v) \in \Pi_\pi(\mathbb{S}(o))$, we update $\mathbb{E}$ by adding the points-to set of $v$ under condition $\varphi$ as a subset of the points-to set of $p$.

Rule SEQUENCING and Rule BRANCHING deal with compound statements. The former says that we use the post-condition of a statement as the precondition of the immediate next statement. The latter merges the abstract store and environment from multiple paths.

**Merging Value-Flow Edges**. As mentioned in § 3, our analysis computes a guarded and storeless value-flow graph that summarizes value flows induced by the memory. We formalize the rule for building indirect value-flow edges in Fig. 5. The VFLOW rule states that when $q@\ell_1$ and $p@\ell_2$ are stored and loaded from the same memory object $o$, $p@\ell_2$ may alias with $q@\ell_1$.

In the rule, suppose that $\Pi_{\varphi_2}(\mathbb{E}(y)) = \{(\pi_1, o_1), (\pi_2, o_2)\}$, such that $(\varphi_1, \ell_1, q) \in \Pi_{\pi_1}(\mathbb{S}(o_1))$ and $(\varphi_2, \ell_1, q) \in \Pi_{\pi_2}(\mathbb{S}(o_2))$. As illustrated in Fig. 1(b), conventional approaches build flow-insensitive value-flow edges [84, 99]. Thus, they have to distinguish a value flow induced by different memory objects, so that they can preserve the capability of precision refinement based on the memory objects. However, these methods can still suffer from the aliasing-path-explosion problem, making the analysis hard to scale (§ 2).

To address this problem, the VFLOW rule merges the value-flow edges, which not only reduces the number of edges but also can normalize and simplify the conditions based on some simple rewriting rules. For example, when merging two value-flow edges under the conditions $\varphi \wedge \pi$ and $\neg\varphi \wedge \pi$ respectively, the condition can be simplified as $\pi$ after merging.

**Algorithm 1:** Write a value to memory objects and propagate the value

**Input:** A store statement $\ell, \varphi, *x = v$
**Output:** Update the abstract store $\mathbb{S}$

```
for (π, o) ∈ Π_φ(𝔼(x)) do
    𝕊_ℓ(o) ← 𝕊_ℓ(o) ∪ {(π, ℓ, v)};
    forall ℓ′ is a dominance frontier of ℓ do
        𝕊_ℓ′(o) ← 𝕊_ℓ′(o) ∪ {(π ∧ φ, ℓ, v)} ;
```

**Algorithm 2:** Read values from memory objects by walking up the dominator tree

**Input:** A load statement $\ell, \varphi : u = *x$
**Output:** Values that can be loaded from $*x$

```
R ← ∅;
for (β, o) ∈ Π_φ(𝔼(x)) do
    R ← R ∪ ReadFromObject(β, o, ℓ);
return R;
Function ReadFromObject(β, o, ℓ):
    σ ← true, R_o ← ∅;
    while ℓ ≠ null do
        for (π, ℓ′, v) ∈ 𝕊_ℓ(o) do
            φ ← π ∧ σ ∧ β;
            if φ is satisfiable then
                R_o ← R_o ∪ {(φ, ℓ′, v)};
            if (π, ℓ′, v) is a strong update at ℓ then
                return R_o;
            σ ← ¬π ∧ σ ;
        ℓ ← the immediate dominator of ℓ;
    return R_o;
```

**Challenges.** However, a highly precise (e.g., flow- and path-sensitive) analysis that uses the above rules to compute value-flow edges is notoriously expensive, due to the following challenges:

(1) *Conservative propagation.* Propagating data-flow facts along control flows is expensive [30]. To mitigate this problem, data-flow facts can be propagated along with def-use chains. However, the def-use information of memory objects is unavailable without a pointer analysis. To resolve the paradox, most existing efforts [30, 83, 84, 99, 100] perform a lightweight but imprecise pointer analysis to over-approximate the def-use chains. Due to the imprecision, many false def-use relations are introduced, hurting performance.
(2) *Constraints explosion.* Our analysis needs to account for a large number of guard updates for each statement, quickly causing the explosion of constraints. If we aim to design a demand-driven and path-sensitive data dependence analysis, it is both intractable and unnecessary to pay the full price of path-sensitive reasoning upfront.

To address these challenges, we use an *on-the-fly sparse* analysis that computes def-use relations incrementally during the analysis, instead of relying on precomputed imprecise def-use chains (§ 4.1.2), and design a *semi-path-sensitive* analysis that simplifies and partially solves the constraints to merge and prune value-flow edges (§ 4.1.3).

*4.1.2 On-the-Fly Sparse Analysis.* To address the challenge of conservative propagation, we utilize the idea of sparsity to skip unnecessary control flows when propagating data-flow facts. To this end, instead of leveraging the imprecise def-use relations computed by a pre-analysis, we construct the def-use relations incrementally during the analysis, along with the precise pointer information discovered. To formally present the idea, we maintain the abstract store $\mathbb{S}$ as a set of $\mathbb{S}_\ell$, which describes the abstract store at the program point $\ell$.

**The STORE Rule**. We follow the idea in SSA form where a variable defined at a program point $\ell$ can only be used at a program point dominated by $\ell$ or in the dominance frontier where the definition is merged with other definitions [13]. Suppose at program point $\ell$, a store statement writes a guarded value $(\varphi, \ell, v)$ to the memory object $o$. As shown in Alg. 1, it takes two steps to update the abstract store. First, we write $(\varphi, \ell, v)$ into the local store $\mathbb{S}_\ell(o)$ (Line 3). Second, we propagate the abstract value to the dominance frontiers of $\ell$. We update the guard for the propagated abstract value, which is the conjunction of $\varphi$ and the path condition of $\ell$ (Lines 4-5). Note that it is unnecessary to propagate the abstract value to program points dominated by $\ell$, because at the load time, we can walk up the dominance tree to find the corresponding definitions (see the next paragraph).

*Example 4.1.* Consider the program in Fig. 6. After the program point $\ell_1$ and the program point $\ell_2$, the pointer $x$ and the pointer $y$ point to the memory objects $alloc_m$, $alloc_n$, respectively. Suppose that we are analyzing the store statement $*x = d$ at the program point $\ell_4$. The abstract value $(\varphi, \ell_4, d)$ is stored into the memory object $alloc_m$ and then propagated to the abstract store of $\ell_4$'s dominance frontier, i.e., the program point $\ell_6$. Therefore, we have $\mathbb{S}_{\ell_4}(alloc_m) = \{(\varphi, \ell_4, d)\}$ and $\mathbb{S}_{\ell_6}(alloc_m) = \{(\varphi, \ell_4, d)\}$. Similarly, after analyzing the store statement $*y = d$ at the program point $\ell_5$, we have $\mathbb{S}_{\ell_5}(alloc_n) = \{(\neg\varphi, \ell_5, d)\}$, which is then propagated to the dominance frontier of the program point $\ell_5$, i.e., the program point $\ell_6$. Hence, we have $\mathbb{S}_{\ell_6}(alloc_n) = \{(\neg\varphi, \ell_5, d)\}$.

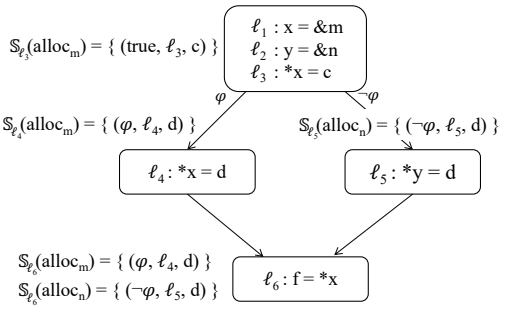


Fig. 6. An example of sparse analysis.

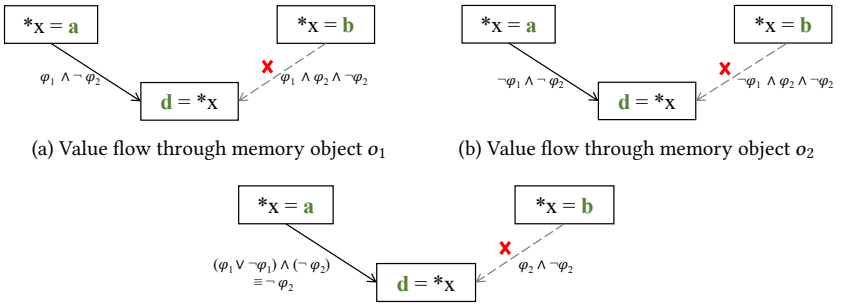
**The LOAD Rule**. As shown in Alg. 2, for a load statement $u = *x$ at the program point $\ell$, we track the values that can be read from the memory objects pointed-to by $x$. For each memory object $o$, it suffices to walk up the dominance tree (Lines 6-15) to gather abstract values until a strong update is found. The basic idea behind the approach is that the definition of a variable must dominate its uses. This is a linear search in the dominance tree.

*Example 4.2.* Consider the program in Fig. 6. When analyzing the load statement $f = *x$ at the program point $\ell_6$, we need to read values from the memory objects pointed-to by $x$. To this end, we gather abstract values stored into $alloc_m$ that is pointed-to by $x$. To do this, the sub-procedure `ReadFromObject` walks up the dominator tree from the program point $\ell_6$. From the dominator $\ell_6$, we can read the value $(\varphi, \ell_4, d)$, which is written into $alloc_m$ at $\mathsf{b}_2$ and propagated to $\mathsf{b}_4$. From the dominator $\ell_3$, we can read the value $(\varphi, \ell_3, c)$, which is written into $alloc_m$ at $\mathsf{b}_1$ that dominates $\mathsf{b}_4$.

*4.1.3 Semi-Path-Sensitive Analysis.* Path sensitivity comes in many flavors, depending on the information encoded as constraints. Previous research on path-sensitive pointer/heap analysis has either (1) adopted relatively coarse abstractions in which Boolean variables abstract the control flow, ignoring the actual predicate of the branching condition and the computations in each basic block [88], or (2) used expensive abstractions with first-order formulas to encode the entire history of memory writes and reads, resulting in significant overhead [18, 28, 54].[1]

To build the guarded and storeless value-flow graph, we explore a sweet spot in the space that is *semi-path-sensitive* with the following characteristics. First, we work on a propositional abstraction

[1]Livshits and Lam [54] invoke a computer algebra system. Hackett and Aiken [28] implement a procedure similar to bit-blasting that translates arithmetic constraints to SAT constraints. Dillig et al. [18] use the Mistral SMT solver.




(a) Value flow through memory object $o_1$

(b) Value flow through memory object $o_2$

(c) Merging value flows through $o_1$ and $o_2$

Fig. 7. Pruning and merging value-flow graph edges for the program in Fig. 1(a).

of the program, where program statements are abstracted as *Boolean skeletons*, as in CDCL($T$)-based SMT solving. For instance, we abstract the branching conditions $x > 2$ and $x \leq 2$ as well as the assignment t $y = 100/2$, to fresh Boolean literals $p$, $\neg p$, and $q$, respectively. This encoding allows for some degrees of branch correlation tracking while also preserving the capability to recover the full path condition in the subsequent transitive dependence tracking phase. Second, instead of applying a full-featured SAT solver, we adopt several linear time semi-decision procedures such as unit-propagation [102] for identifying "easy" unsatisfiable constraints, as well as performing lightweight logical simplifications such as tautology elimination.

In our experiment, we discovered that approximately 70% of the path conditions generated by the analysis are satisfiable. For the remaining ones, 80% of them are easy constraints and can be solved with the semi-decision procedures. We found that many infeasible path-sensitive facts can be filtered because programmers tend to maintain *implicit and simple correlation of conditional points-to relations*, which helps ensure some logical properties (e.g., cross-platform compatibility) and improves human readability. This correlation is made explicit by our analysis.

*Example 4.3.* Consider the program in Fig. 1(a). Observe that the variable $x$ may point to $o_1$ or $o_2$. A path-insensitive algorithm will conclude that $d$ may alias with $\{a, b\}$, where $b$ is a false positive. We now explain intuitively how our algorithm works and prunes the false positive. Let us consider the two cases where $x$ points to $o_1$ or $o_2$, respectively. First, if $x$ points to $o_1$, as in Fig. 7(a), our analysis will conclude that (1) $(\varphi_1 \wedge \neg\varphi_2, a)$ and (2) $(\varphi_1 \wedge \varphi_2 \wedge \neg\varphi_2, b)$ may flow to $d@d = *x$. The semi-path-sensitive analysis can decide that the guard of the second item is unsatisfiable. Hence, the value $b$ is pruned. Second, if $x$ points to $o_2$, as in Fig. 7(b), the analysis can also prune the value $b$. Finally, after merging the two graphs induced by $o_1$ and $o_2$, we obtain the graph in Fig. 7(c).

**Summary**. The on-the-fly sparse analysis and the semi-path-sensitive analysis conspire to address the challenges of conservative propagation and constraints explosion (§ 4.1.1). When propagating data-flow facts, sparse analysis skips unnecessary control flows, improving analysis efficiency. The semi-path-sensitive analysis removes false points-to-facts and merges and simplifies redundant ones, which not only improves precision but also benefits efficiency because smaller points-to sets lead to less work [46, 76]. Note that abstracting relations as Boolean variables is conservative. Our semi-decision procedure can soundly prune unsatisfiable ones but is incomplete, i.e., unsatisfiable ones may be classified as satisfiable but not vice versa.

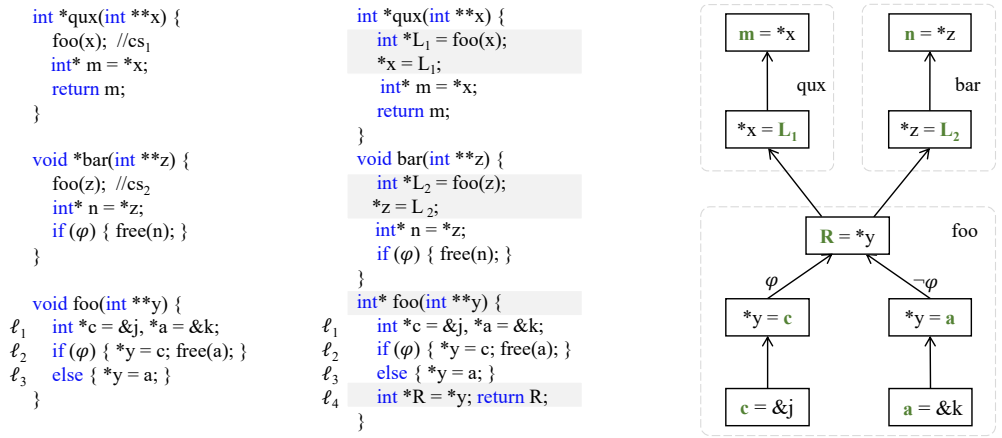
## 4.2 Interprocedural Analysis

For building interprocedural value-flow graphs, we perform a bottom-up and summary-based analysis, which breaks down the entire abstraction into smaller components to enable the on-demand resolution of data dependency.

**Algorithm 3:** Summarize an interface variable

**Input:** An interface variable (parameter or return) $x$
**Output:** Updated $\mathbb{E}, \mathbb{S}$ with auxiliary variables

$WL \leftarrow \{x\}$;
**while** *$WL$ is not empty* **do**
  $x \leftarrow$ pop an element from $WL$;
  $R, o_x \leftarrow$ create an auxiliary variable and a memory object;
  **for** $(\pi, o) \in \mathbb{E}(x)$ **do**
    **for** $(\varphi, \ell', v) \in \Pi_\pi(\mathbb{S}(o))$ **do**
      $\mathbb{E}(R) \leftarrow \mathbb{E}(R) \cup \Pi_\varphi(\mathbb{E}(v))$ ;
  $\mathbb{E}(x) \leftarrow \{(\texttt{true}, o_x)\}, \mathbb{S}(o_x) \leftarrow \{(\texttt{true}, \_, R)\}$;
  **if** *$R$ is a pointer* **then**
    $WL \leftarrow WL \cup \{R\}$




(a) Code snippet (b) Transformed code (c) The value-flow graph

Fig. 8. An example of using a concise summary and performing the on-demand search.

*4.2.1 Prerequisite and Assumption.* We first discuss a few design choices for our approach, which are important to make path-sensitive analysis practical.

**Call Graph**. A pointer analysis often faces the "chicken-and-egg" problem: performing the analysis requires a call graph, which in turn requires reasoning about function pointers [25]. To obtain a sound call graph, we use a Steensgaard-style, flow- and context-insensitive analysis [103]. Previous studies [30, 63] have shown that a precise call graph for C-like programs can be constructed using only flow-insensitive analysis. In our pointer analysis for building value-flow graphs, we do not refine the call graph on the fly. Applying on-the-fly call graph construction in a precise analysis, which is both flow- and semi-path-sensitive, can be computationally expensive. This design follows [54] but differs from the common practices of the Java pointer analysis community, and can be a source of imprecision in our approach.

**Entry Aliasing.** The handling of possible alias relations among function parameters in the absence of caller information is an important design choice in bottom-up summary-based pointer analysis [18]. The relevant context inference (RCI) [11] approach eagerly constructs one transfer function by assuming the presence of all possible entry aliasing, whereas the partial transfer function (PTF) [96] lazily constructs multiple transfer functions based on different aliasing at call sites. We notice that these approaches can be prohibitively expensive in a flow-, context- and semi-path-sensitive analysis. Hence, to trade soundness for scalability, we assume that there is no entry aliasing, following [97].

*4.2.2 Summary Generation and Application.* We present several important optimizations for improving the scalability of the analysis.

**Concise Size-Effects Summaries**. To achieve context sensitivity, conventional summary-based analyses conservatively identify the side effects of a function, which are then cloned at every call site of the summarized function in the upper-level callers [18, 97]. However, the size of the side-effect summary can quickly explode, becoming a significant obstacle to scalability. To illustrate, consider the program in Fig. 8(a). In conventional approaches, the summary of *foo* is the points-to information, $\mathbb{E}$ and $\mathbb{S}$, of the interface variable $y$ at the exit point of *foo*:

$$\mathbb{E}(y) = \{(\texttt{true}, o)\};\ \ \mathbb{S}(o) = \{(\varphi, \ell_2, c), (\neg\varphi, \ell_3, a)\};$$

The two variables $a$ and $c$ are cloned twice by cloning the summary to the two call sites in *qux* and *bar*. When the summaries of *qux* and *bar* are cloned to their upper-level callers, $a$ and $c$ will continue to be cloned. As a result, the size of the summary will increase exponentially.

To mitigate the problem, our basic idea is to introduce symbolic auxiliary variables, each of which stands for a class of variables to clone. Then, we can only clone a single auxiliary variable during interprocedural analysis, reducing the burden of cloning. For the above example, we introduce an extra value $R$ for the function *foo* to represent all values (e.g., $a$ and $c$) stored in the memory object pointed to by $y$. As a result, the function summary gets smaller as the following, and we only need to clone a single variable $R$ to the callers during the interprocedural analysis.

$$\mathbb{E}(y) = \{(\texttt{true}, o)\};\ \ \mathbb{S}(o) = \{(\texttt{true}, \ell_4, R)\};$$
$$R \mapsto \{(\varphi, \ell_2, c), (\neg\varphi, \ell_3, a)\}$$

Intuitively, this summarization process entails adding an extra return value to the function *foo*, as depicted in Fig. 8(b). Formally, the process is illustrated in Alg. 3, where the points-to results are merged into a single auxiliary variable, to alleviate the burden imposed by the cloning processes. Each auxiliary variable represents a modified non-local memory object accessed via an access path rooted at an interface variable. In summary, the process facilitates local reasoning concerning the value flows, as opposed to global reasoning encompassing the entire heap.

**Summary Application.** When a caller may invoke a summarized function, we apply the function summary at the call site, as shown with the rule in Fig. 9. In the rule, we use $\mathbb{E}_f$ and $\mathbb{S}_f$ to denote the points-to environment and abstract store of the callee $f$ respectively and use $\mathbb{E}$ and $\mathbb{S}$ for those of the caller. The rule consists of two major steps.

- Step (1) instantiates the summary of the callee $f$ by replacing symbols in callee with those in the caller, which results in the (potentially) updated $\mathbb{E}_f$ and $\mathbb{S}_f$ using context-specific mappings. The mapping of formal parameters/returns is straightforward (Note that $r$ uses strong updates due to SSA), so we focus on the auxiliary variables. Assume we have $AP(F) = *para$, where $para$ and $F$ are the formal parameter and auxiliary variable respectively. We can instantiate $F$ by querying the values that can be loaded from $*u$ ($u$ is the actual parameter), using the points-to environment and abstract store of the caller.
- Step (2) applies the instantiated summary of the callee, which simulates the memory behaviors of the function call by merging the environment and stores of callee into the caller, i.e., performing the summary cloning. Particular, if a memory object is allocated at the callee $f$ (or its transitive callees) and can escape to the caller, we clone the escaped object to distinguish allocations made at different call sites [44, 53].

*Memorizing Instantiation Results.* When processing a function with many (transitive) callees (e.g., functions near the root of the call graph), the above procedure can be time-consuming. To accelerate the process, we use a memorization strategy as follows. In real-world C/C++ programs,

$$\frac{(1)\ \mathbb{E}'_f = \mathbb{E}_f[u/para; A/F; r/ret] \qquad \mathbb{S}'_f = \mathbb{S}_f[u/para; A/F; r/ret] \quad (2)\ \mathbb{E}' = \mathbb{E} \uplus \Pi_\varphi(\mathbb{E}'_f) \qquad \mathbb{S}' = \mathbb{S} \cup \Pi_\varphi(\mathbb{S}'_f)}{\mathbb{E}, \mathbb{S} \vdash \ell, \varphi : r = call\ f(u) : \mathbb{E}', \mathbb{S}'}\ \textsc{call}$$

Fig. 9. Apply the summary of a callee $f(para) \rightarrow ret$, where $para$ and $ret$ are formal parameter and return, respectively. We assume that $u$ and $r$ are actual parameter and return, respectively; $A$ and $F$ are auxiliary actual/formal parameters, respectively.

it is common that different callee of a function share the same structure pointer parameter. As a result, certain access paths need to be instantiated multiple times. Meanwhile, we observe that many structure fields are seldom changed once initialized [93], although not declared as "static". Consequently, the instantiation results tend to be the same across different call sites within a function, except for the path conditions. Hence, to reduce redundant querying and updating of the abstract store $\mathbb{S}$ and points-to environment $\mathbb{E}$, we maintain a cache of summary instantiation results within each function.

### 4.3 Constructing the Value-Flow Graphs

Alongside the analysis, we can construct the value-flow graph (§ 3) for each function. The graph has two types of edges representing value-flow relations: (1) the *direct edge* connects a store to a load (following the rule in Fig. 5), which merges indirect value flows through the relevant memory objects and program paths; and (2) the *summary edge* connects $s$ to $t$ if $s$ can transitively flow to $t$, which summarizes transitive dependencies in a certain program scope. Currently, our analysis eagerly connects the summary edges between a formal argument (formal-in) at the function entry and the return value (formal-out) at the function exit. As shown in Fig. 8(c), in the subsequent client analysis, the local value-flow graphs are stitched together by matching formal and actual parameters as well as the return value and its receivers.

Remark 1. *The edges in our value-flow graphs connecting stores and loads bear similarities to several previous works. For example, the "match edges" in [79] are based solely on filed types, without checking whether they access a common object. The match edges are used in CFL reachability-based points-to analysis. Another example is TAJ [92], which connects stores to loads via Andersen analysis. These edges are utilized by the subsequent context-sensitive hybrid thin slicing. In our approach, the edges are computed via a flow- and semi-path-sensitive pointer analysis, which forms the foundation of our path-sensitive memory model.*

## 5 ANSWERING DEMAND DATA DEPENDENCE QUERIES

By forward or backward graph traversals, the guarded and storeless value-flow graphs can be adopted in various applications.

**Thin Slicing for Program Understanding**. The first typical application is *thin slicing* [52, 80], which can be implemented via a backward traversal on the value-flow graph. Thin slicing is introduced by Sridharan et al. [80] to facilitate program debugging and understanding. A thin slice for a program variable, a.k.a., the slicing seed, includes only the *producer statements* that directly affect the values of the variable. In contrast to conventional slicing, control dependence and the data dependence of the variable's base pointer are excluded. Hence, thin slices are typically much smaller than conventional program slices.

*Example 5.1.* Consider the program in Fig. 8. To build the thin slice for the slicing seed $m$ at function *qux*, we traverse the value-flow graphs from $m$ in a reversed direction, collecting all program

statements that need to be included in the slice. For instance, the statements $*y = a$ and $*y = b$ will be visited and included in the result as they are the producer statements of $m$.

**Value-flow Bug Finding**. The second typical application is *to find value-flow bugs* [12, 87]. The analysis of value flows underpins the inspection of a broad range of software bugs, such as the violations of memory safety (e.g., null dereference, double free, etc.), the violations of resource usage (e.g., memory leak, socket leak, etc.), and security problems (e.g., the use of tainted data). It is vital to precisely resolve value flows caused by pointer aliasing, which is the key problem we address in the paper. Value-flow bug finding can be implemented via a forward traversal on the graphs, during which the alias constraints and property-specific constraints can be gathered together and handed to an SMT solver.

*Example 5.2.* Suppose we need to detect double-free bugs for the program in Fig. 8(a). We traverse its value-flow graphs (Fig. 8(c)) starting from $a$ and obtain one path from $a$ to $n$. We then stitch together the path condition under which $n$ is data-dependent on $a$ (i.e., $\neg\varphi$), and the path conditions of the two statements *free(a)* and *free(n)* (i.e., $\varphi \wedge \varphi$). Observe that we do not compute the interprocedural data dependence between $(a, m)$ or $(c, n)$.

**Summary of Our Approach**. When constructing the value-flow graphs, our analysis operates over a propositional abstraction of the program, which prunes many false value-flow edges, merges duplicate ones, and simplifies the data dependence guards, enabling a path-sensitive memory model upfront. When resolving transitive data dependence over the graphs, the client can piggyback the computation of fully path-sensitive pointer information with the resolution of client-specific dependences, during which it (1) can sparsely track value flows by following the precise edges on demand; and (2) does not need to perform explicit cast-splitting over the points-to sets when handling indirect loads/stores, alleviating a major source of case explosion in previous work [6, 59, 99]. Consequently, the client concentrates computational effort on the path- and context-sensitive pointer information only when it matters to the data dependence of interest.

Remark 2. *Our analysis for building the VFGs is flow-sensitive, context-sensitive, and semi-path-sensitive, requiring a whole program but "non-aggressive" Boolean reasoning to resolve memory dependencies. Thanks to the guarded graphs, the client analyses can compute the path- and context-sensitive conditions on demand. However, there are sources of imprecision, such as the handling of function pointers § 4.2.1, which can result in a loss of precision.*

## 6 EVALUATION

To demonstrate the utility of Falcon, we examine its scalability in constructing the value-flow graphs (§ 6.2) and apply it to two practical clients, namely semi-path-sensitive thing slicing (§ 6.3), and fully path-sensitive bug hunting (§ 6.4).

### 6.1 Experimental Setup

**Implementation**. We implemented Falcon on top of LLVM and Z3 SMT solver. While the language in § 3 has restricted language constructs, Falcon supports most features of C/C++, such as unions, arrays, and classes. Our algorithm's bottom-up and compositional nature lends itself well to parallelism, i.e., functions without dependence can be analyzed in parallel. We mainly report sequential analysis results in this section for a fair comparison.

**Baselines**. In this section, we compare Falcon against three groups of existing analyzers.

- First, we compare it with the following analyses for constructing the value-flow graphs: (1) SVF [85], the Andersen analysis implemented in SVF,[2] (2) SFS [30], an inclusion-based, flow-sensitive, context-insensitive pointer analysis,[3] (3) DSA [44], a unification-based, flow-insensitive, context-sensitive pointer analysis;[4] (4) SUPA-FS [84, 86], a demand-driven, flow-sensitive, context-insensitive pointer analysis, and (5) SUPA-FSCS [84, 86], a demand-driven, flow- and context-sensitive pointer analysis. Note that both SUPA-FS and SUPA-FSCS rely on SVF to build the VFGs, based on which they answer demand points-to queries.
- Second, for the thin slicing client, we compare with SUPA-FSCS [84, 86], the state-of-the-art demand-driven flow- and context-sensitive pointer analysis for C/C++.
- Finally, for the value-flow bug finding client, we compare with (1) CRED [99], which is a state-of-the-art path-sensitive pointer analysis using the layered approach,[5] and (2) Clang Static Analyzer (CSA), which is a state-of-the-art, industry-strength symbolic executor.

We cannot compare with the pointer analyses in [27, 48, 49, 88, 89, 101, 105] because they are not publicly available. For bug finding, we tried our best to compare with Saturn [28] and Compass [16, 18], but they are not runnable on the experimental environment that we can set up.

**Subjects**. Table 1 shows the benchmarks. Six of them are taken from SPEC CINT2000 and ten are from open-source projects. These programs cover various applications such as text editors and database engines, with sizes ranging from 13 KLoC to 8 MLoC. It is important to note that, as Falcon unrolls loops on the control flow graph and the call graph, we feed the same transformed code to other tools.

**Environment**. All experiments are conducted on a 64-bit machine with 40 Intel Xeon E5-2698 CPUs@2.20 GHz and 256 GB of RAM. The reported data represents the medians of three runs.

### 6.2 Value-flow Graph Construction

First, we examine the scalability of Falcon for constructing value-flow graphs. The cutoff time per tool per program is 12 hours.

**Comparing with SVF, SFS, and DSA**. Table 1 and Fig. 10 show the results of the four analyses. In terms of runtime overhead, they perform similarly in small-sized programs. However, on programs with more than 500 KLoC, SVF and SFS get derailed and become orders-of-magnitude more expensive and fail to analyze `mysql`, `rethinkdb`, and `firefox` within 12 hours. DSA is comparable to Falcon on `vim` and `php`, but much slower on other large programs (`git`, `wrk`, `libicu`, and `mysql`). Also, DSA cannot finish the analysis of `rethinkdb` and `firefox`. To sum up, Falcon is on average 17×, 25×, and 4.4× faster than SVF, SFS, and DSA, respectively. In terms of memory consumption, on average, Falcon takes 1.4×, 1.9×, and 4.2× less memory than SVF, SFS, and DSA, respectively.

We attribute the graceful scalability of Falcon to two factors. First, combining on-the-fly sparsity and semi-path-sensitivity translates into significant gains in precision and performance (§ 4.1). Second, instead of cloning the full points-to information to achieve context sensitivity, we utilize a concise summary to avoid computing a whole-program image of the heap (§ 4.2). We acknowledge that a drawback of the design is that we cannot use the computed VFG to answer points-to queries in $O(1)$ time, as our analysis is not an exhaustive points-to analysis.

**Comparing with SUPA-FS and SUPA-FSCS**. The two state-of-the-art, demand-driven pointer analyses (§ 6.1) rely on SVF to build flow-insensitive VFGs for answering demand queries. We

[2]The authors of https://github.com/SVF-tools/SVF have implemented several optimizations for improving the performance of Andersen analysis, such as HCD and wave propagation. We use its default configuration for the evaluation.
[3]We use the implementation maintained by the SVF team, which is also available at https://github.com/SVF-tools/SVF.
[4]We use the implementation maintained by the Seahorn team named sea-dsa, i.e., https://github.com/seahorn/sea-dsa
[5]The tool is not open-source. We implement the algorithm on top of SVF.

Table 1. Benchmark size (**KLoC**) and runtime (in **minutes**) of building value-flow graphs.

| Program | Size | SVF | SFS | DSA | Falcon | Falcon(PI) | Falcon(SAT) |
|---|---|---|---|---|---|---|---|
| crafty | 13 | <0.1 | <0.1 | <0.1 | <0.1 | 0.2 | 1.9 |
| eon | 22 | 0.3 | 0.7 | 0.2 | 0.3 | 0.4 | 2.1 |
| gap | 36 | 0.4 | 1.7 | 0.9 | 0.5 | 0.7 | 13 |
| vortex | 49 | 0.1 | 0.2 | <0.1 | 0.3 | 0.3 | 26 |
| perlbmk | 73 | 0.7 | 2.7 | 2.1 | 2.1 | 3.2 | 125 |
| gcc | 135 | 0.7 | 7.4 | 8.6 | 7.8 | 8.1 | 145 |
| git | 185 | 121 | 243 | 21 | 10 | 16 | 393 |
| vim | 333 | 186 | 221 | 17 | 14 | 23 | 484 |
| wrk | 340 | 85 | 115 | 131 | 6 | 15 | 537 |
| libicu | 537 | 454 | 570 | 37 | 5 | 6 | OOT |
| php | 863 | 519 | 614 | 12 | 12 | 41 | OOT |
| ffmpeg | 967 | 43 | 122 | 113 | 13 | 36 | OOT |
| ppsspp | 1648 | 34 | 94 | 92 | 8 | 25 | OOT |
| mysql | 2030 | OOT | OOT | 113 | 46 | 64 | OOT |
| rethinkdb | 3776 | OOT | OOT | OOT | 90 | 113 | OOT |
| firefox | 7998 | OOT | OOT | OOT | 167 | 273 | OOT |

OOT means the analysis runs out of the time budget (12 hours).

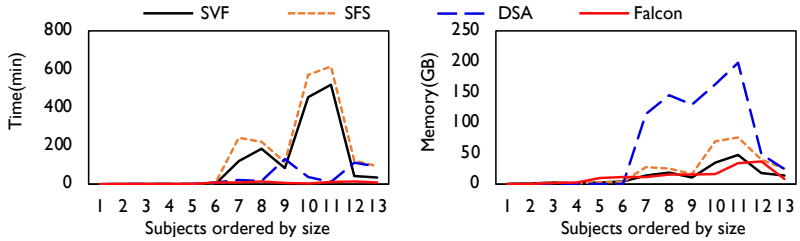



Fig. 10. Comparing the time and memory cost of SVF, SFS, DSA, and Falcon for VFG construction. We present the results of the 13 programs analyzed by all tools within the time budget.

attempted to run them to refine the flow-insensitive VFGs, i.e., using the "VFG refinement client" to generate demand queries. However, the number of queries is huge as we need to refine the results for all functions. And for the demand-driven search, computing reachability between two nodes may resort to a graph traversal among all nodes in the worst. In practice, we found that SUPA-FS and SUPA-FSCS only finished analyzing crafty and econ, running out of the time budget for the remaining programs.

**The Effects of Semi-Decision Procedures**. To understand the effects of constraint solving, we set up two additional configurations of Falcon for constructing value-flow graphs. Specifically, Falcon-PI is path-insensitive, while Falcon-SAT uses a full-featured SAT solver. The last three columns of Table 1 compare the three configurations. Falcon is usually more–and occasionally much more–efficient than Falcon-PI, due to the increased precision. However, Falcon-SAT is not a good choice in practice: its precision is offset by unbearable runtime overhead. In particular, Falcon-SAT runs out of the time budget for all programs of more than 500 KLoC.

The results indicate that solving constraints when building value-flow graphs pays off, which naturally raises the question: *could we do better by tuning the semi-decision procedure more aggressively?* However, we find that being "too aggressive" can lead to performance overhead that overwhelms the benefits. For instance, we attempted the $O(n^3)$ Gaussian elimination algorithm for solving linear constraints, leaving the analysis hard to scale to millions of lines of code. Adapting the decision procedures defines a sophisticated design space that deserves further optimizations.

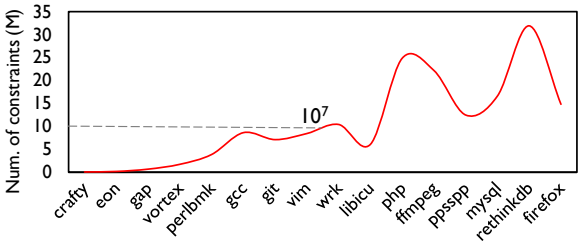



Fig. 11. Number of constraints Falcon deals with when building value-flow graphs.

To provide insight into the nature of constraints explosion, in Fig. 11, we report the number of constraints Falcon deals with. Even when unrolling all loops in the control-flow graphs and callgraph, we can see that it is not unusual to have over $10^7$ constraints.

### 6.3 Thin Slicing for Program Understanding

We aim to measure the precision of Falcon's semi-path-sensitive value-flow graphs (§ 4.1.3) in this experiment. So, for this client, we do not invoke an SMT solver to achieve full path sensitivity, To generate realistic queries, we use the bug reports issued by a third-party typestate analysis. The slicers start from the problematic variables and program locations, and the results can assist developers in understanding these reports. The following experimental results exclude the time required for building the value-flow graphs.

**Speed**. Our results show that Falcon scales gracefully for the thin-slicing client, with each demand query taking less than 240 milliseconds. In summary, it achieves up to 302× speedups than SUPA-FSCS and 54× on average. This performance improvement is attributed to the more compact value-flow graphs generated by Falcon compared to SUPA-FSCS. On one hand, SUPA-FSCS constructs the graphs using flow-insensitive analysis, while Falcon employs flow-sensitive analysis. On the other hand, SUPA-FSCS has to explicate a memory object on an edge so that it can answer demand queries, where many edges are redundant.

**Precision**. Besides, Falcon is more precise for answering the queries. Fig. 12 compares the precision of Falcon against SVF, SFS, DSA, and SUPA-FSCS on the 13 programs got analyzed by all tools. The results were manually verified by two authors of the paper. The data for each program is normalized based on the results of SVF, where a higher bar corresponds to a more precise analysis. We make the following observations:

- The average size of slices produced by Falcon is 5.5×, 1.9×, 2.6×, and 1.3× smaller than that of SVF, SFS, DSA, and SUPA-FSCS, respectively.
- Comparing SFS and SVF, we see that flow sensitivity can substantially improve the precision of Andersen's analysis in some programs, such as `php` and `ffmpeg`.
- DSA is comparable to SVF in some cases, and much more precise than SVF in many programs (e.g., `vim`, `libicu`, `ffmpeg`). The combination of context sensitivity and unification may bring better precision than the flow- and context-insensitive Andersen's analysis.

In summary, Falcon offers a visibly improved precision. An important reason is that Falcon's flow- and semi-path-sensitive analysis (§ 4.1) can prune away more spurious value flows, compared with the demand-driven flow- and context-sensitive analysis in SUPA-FSCS.

**Recall**. Falcon's unsound assumption that function parameters are alias-free does not affect the soundness for >90% of the queries, validated by manually checking the results. Similar to our observation, two previous studies [23, 89] also show that the function parameters of real-world C/C++ programs tend to have few aliasing relations.

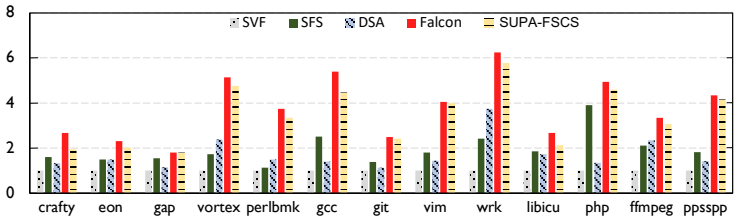



Fig. 12. Improvement in average slice size compared with the baseline SVF.

Table 2. Results of use-after-free bug detection.

| | CRED | | CSA | | Falcon | |
|---|---|---|---|---|---|---|
| **Program** | Time(m) | #FP/#Rep | Time(m) | #FP/#Rep | Time(m) | #FP/#Rep |
| crafty | 0.3 | 0/0 | 715 | 0/0 | 0.1 | 0/0 |
| eon | 1.4 | 0/0 | 397 | 0/0 | 0.9 | 0/0 |
| gap | 3.7 | 0/0 | OOT | 0/0 | 1.9 | 0/0 |
| vortex | 1.2 | 1/1 | 642 | 0/0 | 1.3 | 0/0 |
| perlbmk | 97 | 0/0 | 285 | 0/0 | 81 | 0/0 |
| gcc | 31 | 0/0 | 398 | 0/0 | 26 | 0/0 |
| git | 218 | 0/3 | OOT | 1/1 | 41 | 1/4 |
| vim | 1852 | 0/0 | OOT | - | 137 | 0/0 |
| wrk | 247 | 0/0 | FAIL | - | 9 | 0/0 |
| libicu | 932 | 4/7 | OOT | 0/0 | 16 | 1/3 |
| php | OOT | - | OOT | 0/0 | 50 | 2/9 |
| ffmpeg | 383 | 3/11 | OOT | 0/0 | 52 | 2/10 |
| ppsspp | 176 | 2/3 | OOT | - | 75 | 1/3 |
| mysql | OOT | - | OOT | - | 214 | 1/4 |
| rethinkdb | OOT | - | OOT | 0/2 | 179 | 0/0 |
| firefox | OOT | - | OOT | - | 483 | 0/2 |
| **%FP** | 40.0% (10/25) | | 33.3% (1/3) | | 27.8% (11/36) | |

"OOT" means out of the time budget. "FAIL" means the tool crashed abnormally.

### 6.4 Path-Sensitive Value-Flow Bug Finding

In this study, we investigate the efficiency and effectiveness of Falcon for use-after-free detection, by comparing it against CRED [99] and Clang Static Analyzer (CSA). We use the CSA configuration that employs Z3 [15] for path-sensitivity, which is align with CRED and Falcon. We impose a 15-second time limit for each SMT query sent to Z3. Each analyzer is run in single-thread mode with a cutoff time of 24 hours per program.

Table 2 presents the time overhead of the tools, the number of reported warnings, and the number and rate of false positives. As can be seen, Falcon surpasses the performance of CRED and CSA for most large-scale programs, achieving up to 10.3× and 1620.8× speedups on average (measured using the projects finished by the tools). Although not shown in the table, we remark that, if allowing the concurrent analysis of 10 threads, Falcon can finish the checking of each program within two hours. The false-positive rates of Falcon, CRED, and CSA are 40.0%, 33.3%, and 27.8%, respectively. We notice that CSA reports much fewer warnings than CRED and Falcon, due partly to the frequent timeouts and its limited capability in analyzing paths across compilation units. Falcon aligns with the common industrial requirement of 30% false positives [5, 62].

Overall, our findings conclude that the ideas behind Falcon have considerably practical value. The tool shows promise in providing an industrial-strength capability of hunting use-after-free bugs, considering scaling efforts, precision, and recall.

## 7 RELATED WORK

**Path-Sensitive Analysis**. Table 3 gives key properties of several existing path-sensitive algorithms. Here we summarize some of the approaches with a focus on pointer reasoning.

Table 3. A comparison of key properties of several existing typical analyses. The "Inter-PS" column represents interprocedural path-sensitive. The "PS-Heap" and "Sparse" columns respectively indicate whether the analysis uses path-sensitive heap abstraction and is sparse. Finally, the "Shown to Scale" column indicates whether the algorithm has been shown to scale to large programs with multi-million lines of code.

| Algorithm | Inter-PS | PS-Heap | Sparse | Shown to Scale |
|---|---|---|---|---|
| Das et al. [14] | X | | | X |
| Ball and Rajamani [4] | X | | | |
| Livshits and Lam [54] | X | X | | |
| Hackett and Aiken [28] | | X | | X |
| Babic and Hu [3] | X | | | X |
| Chandra et al. [9] | X | X | | |
| Dillig et al. [18] | X | X | | |
| Sui et al. [88] | | X | X | X |
| Blackshear et al. [6] | X | X | | |
| Li et al. [47] | X | X | | |
| Yan et al. [99] | X | X | X | |
| Kim et al. [43] | X | X | | |
| Smaragdakis et al. [75] | X | X | | |
| Current paper | X | X | X | X |

Livshits and Lam [54] introduce a flow-, path-, and context-sensitive pointer analysis, which only scales to programs up to 13KLoC. The pointer analyses in [28, 88] are only intraprocedurally path-sensitive. Dillig et al. [16, 18] present a path- and context-sensitive heap analysis that scales to program with 128KLoC. Blackshear et al. [6] introduce a symbolic-explicit representation that incorporates the pre-computed flow-insensitive points-to facts to guide the backward symbolic execution. Similar to the index variables in [16, 18] and symbolic variables in [6], we use guards qualifying value-flow graph edges to mitigate the issue of case splitting over points-to sets. However, their approaches are either not demand-driven or non-sparse. Smaragdakis et al. [75] present a symbolic analysis for Ethereum smart contracts. They use a (lightweight) symbolic solver to collaborate with the flow computation of the analysis, similar to our first phase of value-flow graph construction. In comparison, we use space analysis to accelerate the first phase and an SMT solver in the second phase, which (in general) allows for more sophisticated constraint reasoning.

Our work follows a long line of research on path-sensitive dataflow analysis. ESP [14] encodes a typestate property into a finite state automaton, which is used as a criterion for merging program paths. ESP is similar to *trace partition* [60] and *elaborations* [72] that control the trade-off between performing joining operations or logical disjunctions at control flow merge points. By contrast, Falcon uses logical disjunction to precisely merge value-flow guards.

Shape analysis [70] proves data-structure invariants and has had a major impact on the verification community. Precise shape analyses [47, 70] that are capable of path-sensitive heap reasoning do not readily scale to large programs [22]. There have been scalable solutions such as compositional shape analysis based on bi-abduction [8, 26], yet they do not guarantee precision.

**Data Dependence Analysis**. There is a huge amount of literature on context-sensitive data dependence analysis via CFL reachability [10, 34, 69, 77, 79], or other language reachability problems [90, 104]. P/taint [24] unifies points-to and taint analysis by extending the Datalog rules of the underlying pointer analysis and then computing the information all together. Our second phase bears similarities with P/taint in that it can combine aliasing information and client-specific constraints. However, we use a pre-analysis to enable a path-sensitive memory abstraction. In the compiler community, there have been several solutions to path-sensitive array dependence analysis [61], using SMT solving [64], quantifier elimination [65], among others [68]. Typically, they focus on array-manipulating loops and do not handle complicated pointer operations.

**Sparse Pointer Analysis**. The idea of sparse analysis stems from the static single assignment (SSA) form that encodes def-use chains explicitly. By leveraging the partial SSA form in LLVM, Hardekopf and Lin [29] propose an inclusion-based and *semi-sparse* flow-sensitive pointer analysis. It is semi-sparse since it only utilizes the def-use chains of the top-level pointers.

To achieve full sparsity, the def-use information of address-taken variables is needed. There are two classes of full-sparse analysis. First, the *staged sparse approach* [30, 83, 84, 99] exploits a light-weight and exhaustive pointer analysis to approximate the def-use relations, such as Andersen analysis. Because of the imprecision, spurious value flows will be introduced, harming the performance of the subsequent analysis. Second, the *on-the-fly sparse approach* [48, 88, 101] constructs the def-use chains alongside the pointer analysis. Specifically, SPAS [88] is the only previous analysis that is both path-sensitive and on-the-fly sparse. However, it achieves incremental sparsity by extending the level-by-level analysis [101], which is exhaustive. Falcon belongs to the on-they-fly sparse approach and the crux is to avoid an exhaustive but imprecise pre-analysis. Our use of domination relation is similar to Madsen and Møller [58]'s sparse dataflow analysis, which does not address path sensitivity. Besides, while they compute dataflow facts together with pointer information, we use a client-independent analysis for VFG construction and resolve client-specific dependence in the subsequent phase.

**Demand-Driven Pointer Analysis**. Demand-driven program analyses only analyze parts of the program that are relevant for answering a given query. To date, most existing demand-driven pointer analyses for C/C++ [33, 71, 106] and Java [20, 57, 73, 79, 81, 82, 98] are flow-insensitive. Their underlying data structures, such as the pointer expression graph [106], entirely or partially lose the control flow information and, thus, are not easy to extend for path sensitivity. Recently, there has been a resurgence of interest in demand-driven flow- or path-sensitive pointer analysis [77, 78, 84, 99]. Some of these approaches are not sparse [77, 78]. Some of them are sparse but suffer from the aliasing-path-explosion problem [84, 99].

Over the last decade, there has been a large body of work on *hybrid pointer analyses* [42] that resemble demand-driven approaches. By contrast, they are not query-driven, but schedule analysis strategies such as selective context sensitivity for different pointers. Introspective analysis [76] tunes context sensitivity per function based on a pre-analysis that computes heuristics such as "total points-to information". Since then, several follow-up works have been conducted [31, 32, 36–39, 50, 51, 55, 56], which have significantly advanced context-sensitive pointer analysis for Java. Our work focuses on C/C++ programs. We employ a flow-insensitive analysis for function pointers and a precise path- and context-sensitive analysis for other pointers of interest.

## 8 CONCLUSION

We presented Falcon, our approach to path-sensitive sparse data dependence analysis. Its graceful scalability and high precision rest on our solution to the aliasing-path-explosion problem. Our work provides strong evidence that employing path-sensitive data-dependence analysis is a reasonable choice for millions of lines of real-world code.

## ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments. We also appreciate Dr. Chengpeng Wang for insightful discussions. This work is supported by the National Key R&D Program of China (2023YFB3106000), the National Natural Science Foundation of China (62302434, 62272400), ITS/440/18FP grant from the Hong Kong Innovation and Technology Commission, and research grants from Huawei, Microsoft, and TCL. This work was mostly completed at The Hong Kong University of Science and Technology. Qingkai Shi is the corresponding author.

## REFERENCES


[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. New York, NY, USA.

[2] Robert S Arnold. 1996. Software Change Impact Analysis. (1996).

[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 (ICSE '08)*. New York, NY, USA.

[4] Thomas Ball and Sriram K Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. New York, NY, USA.

[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).

[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 (PLDI '13)*. New York, NY, USA.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. Berkeley, CA, USA.

[8] Cristiano Calcagno, Dino Distefano, Peter W O'hearn, and Hongseok Yang. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011).

[9] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. New York, NY, USA.

[10] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck Reachability for Data-dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 30 (Dec. 2017).

[11] Ramkrishna Chatterjee, Barbara G Ryder, and William A Landi. 1999. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. New York, NY, USA.

[12] 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 (PLDI '07)*. New York, NY, USA.

[13] 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 Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991).

[14] 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 (PLDI '02)*. New York, NY, USA.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Berlin, Heidelberg.

[16] 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 (ESOP'10)*. Berlin, Heidelberg.

[17] Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise reasoning for programs using containers. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM.

[18] 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 (PLDI '11)*. New York, NY, USA.

[19] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. New York, NY, USA.

[20] Yu Feng, Xinyu Wang, Isil Dillig, and Calvin Lin. 2015. EXPLORER: query-and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. New York, NY, USA.

[21] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. New York, NY, USA.

[22] 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).

[23] Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. 2016. Flow- and Context-Sensitive Points-To Analysis Using Generalized Points-To Graphs. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.).

[24] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017).

[25] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (Nov. 2001).

[26] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. 2009. Bottom-Up Shape Analysis. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.).

[27] Samuel Z Guyer and Calvin Lin. 2005. Error checking with client-driven pointer analysis. *Sci. Comput. Program.* 58, 1-2 (Oct. 2005).

[28] Brian Hackett and Alex Aiken. 2006. How is aliasing used in systems software?. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. New York, NY, USA.

[29] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.).

[30] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*.

[31] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. New York, NY, USA.

[32] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2023. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis. *IEEE Trans. Software Eng.* 49, 2 (2023).

[33] Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. New York, NY, USA.

[34] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.* 12, 1 (Jan. 1990).

[35] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 36–50.

[36] Minseok Jeon, Sehun Jeong, Sung Deok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019).

[37] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018).

[38] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).

[39] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (Oct. 2017).

[40] Vineet Kahlon. 2008. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. New York, NY, USA.

[41] Vini Kanvar and Uday P Khedker. 2016. Heap abstractions for static analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016).

[42] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. New York, NY, USA.

[43] Yunho Kim, Shin Hong, and Moonzoo Kim. 2019. Target-driven Compositional Concolic Testing with Function Summary Refinement for Effective Bug Detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. New York, NY, USA.

[44] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. New York, NY, USA.

[45] 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 (POPL '11)*. New York, NY, USA.

[46] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008).

[47] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. 2017. Semantic-directed clumping of disjunctive abstract states. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. New York, NY, USA.

[48] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. New York, NY, USA.

[49] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and scalable context-sensitive pointer analysis via value flow graph. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. New York, NY, USA.

[50] Yue Li, Tian Tan, Anders Moller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018).

[51] Yue Li, Tian Tan, Anders Moller, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. New York, NY, USA.

[52] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Dagstuhl, Germany.

[53] Donglin Liang and Mary Jean Harrold. 2001. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of the 8th International Symposium on Static Analysis (SAS '01)*. London, UK, UK.

[54] 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 (ESEC/FSE-11)*. New York, NY, USA.

[55] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.).

[56] Jingbo Lu and Jingling Xue. 2019. Precision-Preserving yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019).

[57] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An incremental points-to analysis with CFL-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction (CC'13)*. Berlin, Heidelberg.

[58] Magnus Madsen and Anders Møller. 2014. Sparse dataflow analysis with pointers and reachability. In *International Static Analysis Symposium (SAS '14)*. Springer.

[59] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. New York, NY, USA.

[60] Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Conference on Programming Languages and Systems (ESOP'05)*. Berlin, Heidelberg.

[61] Dror E Maydan, John L Hennessy, and Monica S Lam. 1991. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. New York, NY, USA.

[62] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and Incremental Software Bug Detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. New York, NY, USA.

[63] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2004. Precise call graphs for C programs with function pointers. *Automated Software Engg.* 11, 1 (Jan. 2004).

[64] Mahdi Soltan Mohammadi, Kazem Cheshmi, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2018. Extending Index-Array Properties for Data Dependence Analysis *(LCPC '14)*.

[65] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse Computation Data Dependence Simplification for Efficient Compiler-generated Inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. New York, NY, USA.

[66] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. 2004. An Empirical Comparison of Dynamic Impact Analysis Algorithms. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. Washington, DC, USA.

[67] Karl J Ottenstein, Robert A Ballance, and Arthur B MacCabe. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. New York, NY, USA.

[68] Alex Pothen and Sivan Toledo. 2004. Elimination Structures in Scientific Computing.

[69] Thomas Reps. 1998. Program analysis via graph reachability. *Information and software technology* 40, 11 (1998).

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

[71] Diptikalyan Saha and CR Ramakrishnan. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05)*. New York, NY, USA.

[72] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. 2006. Static analysis in disjunctive numerical domains. In *Proceedings of the 13th International Conference on Static Analysis (SAS'06)*. Berlin, Heidelberg.

[73] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. New York, NY, USA.

[74] 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 (PLDI 2018)*. New York, NY, USA.

[75] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts. *Proc. ACM Program. Lang.* 5, OOPSLA (2021).

[76] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. New York, NY, USA.

[77] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019).

[78] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.).

[79] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. New York, NY, USA.

[80] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. New York, NY, USA.

[81] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. New York, NY, USA.

[82] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems (BRACIS '14)*. Washington, DC, USA.

[83] 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 (CGO '16)*. New York, NY, USA.

[84] 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 (FSE 2016)*. New York, NY, USA.

[85] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. New York, NY, USA.

[86] Y. Sui and J. Xue. 2018. Value-Flow-Based Demand-Driven Pointer Analysis for C and C++. *IEEE Transactions on Software Engineering* (2018).

[87] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, Mats Per Erik Heimdahl and Zhendong Su (Eds.).

[88] 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 (APLAS'11)*. Berlin, Heidelberg.

[89] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. 2014. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw. Pract. Exper.* 44, 12 (Dec. 2014).
[90] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. New York, NY, USA.
[91] Wensheng Tang, Yikun Hu, Gang Fan, Peisen Yao, Rongxin Wu, Guangyuan Bai, Pengcheng Wang, and Charles Zhang. 2021. Transcode: Detecting Status Code Mapping Errors in Large-Scale Systems. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 829–841.
[92] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. New York, NY, USA.
[93] Christopher Unkel and Monica S Lam. 2008. Automatic inference of stationary fields: a generalization of java's final fields. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. New York, NY, USA.
[94] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. 2022. Complexity-guided container replacement synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–31.
[95] Wei Wang, Clark W. Barrett, and Thomas Wies. 2017. Partitioned Memory Models for Program Analysis. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.).
[96] Robert P Wilson and Monica S Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. New York, NY, USA.
[97] 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 (POPL '05)*. New York, NY, USA.
[98] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. New York, NY, USA.
[99] 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 (ICSE '18)*. New York, NY, USA.
[100] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-Based Selective Flow-Sensitive Pointer Analysis. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8723)*, Markus Müller-Olm and Helmut Seidl (Eds.).
[101] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. New York, NY, USA.
[102] Hantao Zhang and Mark E Stickely. 1996. An Efficient Algorithm for Unit Propagation. *Proc. of AI-MATH* 96 (1996).
[103] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. New York, NY, USA.
[104] Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. New York, NY, USA.
[105] Jisheng Zhao, Michael G Burke, and Vivek Sarkar. 2018. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. New York, NY, USA.
[106] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. New York, NY, USA.

Received 2023-11-16; accepted 2024-03-31