Anchor: Fast and Precise Value-Flow Analysis for Containers via Memory Orientation

CHENGPENG WANG, The Hong Kong University of Science and Technology, China
WENYANG WANG, Ant Group, China
PEISEN YAO, The Hong Kong University of Science and Technology, China
QINGKAI SHI, Ant Group, China
JINGUO ZHOU, Ant Group, China
XIAO XIAO, Ant Group, China
CHARLES ZHANG, The Hong Kong University of Science and Technology, China

Containers are ubiquitous data structures that support a variety of manipulations on the elements, inducing the indirect value flows in the program. Tracking value flows through containers is stunningly difficult because it depends on container memory layouts, which are expensive to be discovered.

This work presents a fast and precise value-flow analysis framework called Anchor for the programs using containers. We introduce the notion of anchored containers and propose the memory orientation analysis to construct a precise value-flow graph. Specifically, we establish a combined domain to identify anchored containers and apply strong updates to container memory layouts. Anchor finally conducts a demand-driven reachability analysis in the value-flow graph for a client. Experiments show that it removes 17.1% spurious statements from thin slices and discovers 20 null pointer exceptions with 9.1% as its false-positive ratio, while the smashing-based analysis reports 66.7% false positives. Anchor scales to millions of lines of code and checks the program with around 5.12 MLoC within 5 hours.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Software safety**.

Additional Key Words and Phrases: Abstract interpretation, value-flow analysis, data structure analysis

1 INTRODUCTION

A container, e.g., *list*, *set*, or *map*, is an abstract data type that supports manipulating a collection of objects by its interfaces. General-purpose programming languages provide many implementations, such as the C++ STL containers [1], the Java Collections Framework (JCF) [2], and the specific classes in the Java EE framework (Java EE) [3]. Their pervasive usages require a static analyzer to reason how an object flows into and out of containers for various tasks, such as program understanding [4–6], bug detection [7–11], and debugging [12, 13].

Goal and Challenge. Our goal is to establish a fast and precise reasoning of container memory layouts for value-flow analysis. Unfortunately, the problem is always one of the "Achilles' heels" of static analysis [14]. Note

Authors' addresses: Chengpeng Wang, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China, cwangch@cse.ust.hk; Wenyang Wang, Ant Group, China, penguin.wwy@antgroup.com; Peisen Yao, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China, pyao@cse.ust.hk; Qingkai Shi, Ant Group, China, qingkai.sqk@antgroup.com; Jinguo Zhou, Ant Group, China, jinguo.zjg@antgroup.com; Xiao Xiao, Ant Group, China, xx@antgroup.com; Charles Zhang, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China, charlesz@cse.ust.hk.

1049-331X/2022/10-ART $15.00
https://doi.org/10.1145/3565800

```
Queue<Pr> ps = new LinkedList<Pr>();
Pr p = Space.getProject(dir);
ps.offer(p);
ps.offer(new Proj(newDir, arg));
executeProject(ps.peek());
```

(a) Example code in Hibernate-ORM

```
HttpSession<String,Object> s = new HttpSession<>();
s.put(Support.FIND_BLOCK, Boolean.FALSE);
s.put(Support.FIND_WHAT, searchWord);
String word = (String) s.get(Support.FIND_WHAT);
addToHistory(new EditorFindSupport(block, word));
```

(b) Example code in NetBeans

```
Stack<State> s = new Stack<>();
s.addElement(otherStateElem);
s.push(new State(cursor));
State state = includeStack.pop();
```

(c) Example code in Struts

```
Dictionary<String, String> config = new Hashtable<>();
config.put(Factory.CLASS, Driver.getName());
config.put(Factory.NAME, "iotdb");
String name = config.get(Factory.NAME);
```

(d) Example code in IoTDB

Fig. 1. Examples of a programming idiom in Hibernate-ORM, NetBeans, Struts and IoTDB

that a container modification changes both which objects are stored, i.e., the ownership, and which indexes are associated with the objects, i.e., the index-value correlation. The precise reasoning of containers requires the strong updates upon container memory layouts, involving the program facts in multiple domains. First, we require a precise pointer analysis [15–17] to identify the manipulated objects, including both the containers and the elements. Second, applying strong updates to a list-like container relies on numeric analyses [18–20] to determine the relational positions of the manipulations. Third, the indexes of map-like containers are general comparable objects, and its strong updates often depend on complex relational properties of strings [21–24] and user-defined data structures [25, 26]. More importantly, the prerequisites are closely intertwined, demanding a solution to address them simultaneously. The overall quality of the results would collapse if any one of these analyses became imprecise.

Existing Effort. Reasoning container memory layouts is theoretically an undecidable problem [27]. Existing approaches mainly adopt two different strategies to achieve the over-approximation. One line of the techniques smashes a container and only reasons about the ownership without analyzing the indexes [28–32]. Although the analyses scale to large programs, the spurious value flows plague the analysis results such that 75.2% of the client analysis are false positives [33]. The other line of the techniques encodes the program values by logical formulae and applies strong updates by enforcing the container axioms [34, 35]. Despite the high precision, the exhaustive symbolic reasoning introduces a significant number of case analyses, causing the disjunctive explosion problem [36] and degrading the scalability significantly. For example, Compass only scales to 128 KLoC even when the solving procedure is optimized [35, 37].

Insight. Although analyzing generic containers is pie in the sky, there exists a particular class of containers, of which the memory layouts can be precisely tracked by deterministic indexes. As shown in Figure 1, for example, the modifications of the container objects always occur at the end or use constant keys, which can be discovered by tracking all possible modifications upon container objects. The stored objects can be identified by the deterministic indexes, which enables strong updates upon container memory layouts. Specifically, we formulate the idiom by the notion of *anchored containers*. A container is an anchored container at a program location if all the preceding modifications have deterministic indexes. Anchored containers widely exist in real-world programs, e.g., 75.6% Java EE containers in the top 10 cases searched on GitHub conform to the programming idiom[1]. They establish the "anchors" for memory objects, which can be used to identify precise value flows through containers.

[1] The empirical data is listed online: https://containeranalyzer.github.io/empirical.pdf.

```
 1 void foo(String s) {
 2   HttpSession hs;//o1
 3   Map m = new HashMap<String,String>();//o2
 4   hs.setAttribute("id", "a");
 5   hs.setAttribute("age", null);
 6   m.put("id", "b");
 7   String i = hs.getAttribute("id");
 8   if (c) {m.put(s, i);}
 9   else {m.put(s, null);}
10   Stack<String> ids = new Stack<>();//o3
11   String j = m.get("id");
12   ids.add(i);
13   if (c) {ids.add(j);}
14   bar(hs, ids);
15 }
16 void bar(HttpSession hs, Stack ids) {
17   String p = hs.getAttribute("age");
18   String q = ids.peek();
19   String r = hs.getAttribute("id");
20   if (c)
21     out(p.length()+q.length()+r.length());
22 }
```

Fig. 2. A motivating program[2]

Solution. We introduce the *memory orientation analysis* to identify anchored containers and compute their precise index-value correlations, which further support a fast and precise value-flow analysis. Specifically, we establish a combined abstract domain embodied with path constraints to track multi-domain properties precisely, such as points-to facts and deterministic indexes. At a high level, our approach works in two stages:

- The memory orientation analysis identifies anchored containers and applies the strong updates to their memory layouts. A non-anchored container is smashed without analyzing its index-value correlation. Based on container memory layouts, the memory orientation analysis enables the construction of a precise value-flow graph. For example, the container objects o_1 and o_3 in Figure 2 are anchored containers. It index-value correlation implies that p is *null* and r is not *null* at line 21, and the analysis constructs the precise value-flow graph with the solid edges in Figure 3.
- We conduct a demand-driven reachability analysis to solve an instance of the value-flow problem. It collects the value-flow facts of interest when traversing the value-flow graph. The constraints are collected and solved to determine the reachability if necessary. For example, the null pointer exception (NPE) detector traverses the value-flow graph in Figure 3 from *null* values to dereferenced pointers, and reports an NPE with no false positive.

Note that it is non-trivial to identify and apply strong updates to anchored containers in the first stage, involving the accumulative effects of the modifications along control flow paths. For example, the container object o_2 are modified at lines 6, 8, and 9, and the last two modifications have non-deterministic indexes, so o_2 is not an anchored container after line 8. Particularly, we establish a subdomain to maintain the accumulative effects of modifications upon each container object, which explicitly indicates whether it is an anchored container. When transforming each subdomain simultaneously, we instantiate a semantic reduction operator [38] to track the interleaving among multiple subdomains, and apply strong updates to anchored containers.

Highlight. The memory orientation analysis benefits value-flow analysis with three characteristics as follows:

- The memory orientation analysis applies strong updates to anchored containers in the combined domain rather than enforcing container axioms by logics exhaustively, making the analysis more precise than [29, 32] and less vulnerable to disjunctive explosion [36].
- Although the memory orientation analysis only computes the precise index-value correlations of anchored containers, it amplifies the precision benefit and obtains the more precise ownership information of other containers, no matter whether they are anchored containers or not.

[2] The program is simplified from Hibernate-ORM, IoTDB, and Struts.

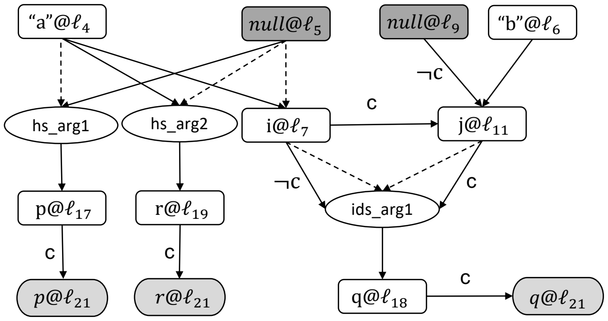


Fig. 3. The value-flow graph (VFG)[3]of Figure 2. A node represents a value at a program location, and an edge from $a@\ell_1$ to $b@\ell_2$ indicates that the value a flows to the value b between the program locations ℓ_1 and ℓ_2. The nodes *hs_arg*1, *ids_arg*1, and *hs_arg*2 represent the auxiliary parameters [39–41], which indicate the elements accessed at lines 17, 18, and 19, respectively.

- The memory orientation analysis divorces analyzing container semantics from value-flow analysis and delays reasoning about the feasibility of value-flow paths until client analyses, effectively alleviating the burden of constraint solving.

We implement and evaluate Anchor by choosing thin slicing [4] and value-flow bug detection [42] as the clients. It is shown that Anchor enables a satisfactory improvement in thin slicing, reporting 17.1% fewer statements than the smashing-based slicer for the real-world programs. Moreover, it discovers all the taint flows through containers in the OWASP benchmark projects [43] with no false positive, and detects 20 NPEs in the real-world projects with 9.1% (2/22) as its false-positive ratio. In contrast, the smashing-based detector reports 31.0% spurious taint flows and 66.7% false positives of NPEs, respectively. Remarkably, Anchor features graceful scalability and finishes analyzing the program with 5.12 MLoC in 5 hours. Upon the submission, there had been 12 true positives of NPEs confirmed by the developers [44]. We also prove the soundness of Anchor theoretically. Anchor has been integrated into the static analysis platform Pinpoint in the Ant Group, an international FinTech company with over 1 billion global users. In summary, we make the following main contributions:

- We introduce the notion of *anchored containers* and establish a combined abstract domain to identify them automatically.
- We propose the *memory orientation analysis* to apply strong updates to anchored containers, which promotes further value-flow analysis.
- We implement and evaluate Anchor by thin slicing and value-flow bug detection, showing its high precision and linear scalability in real-world scenarios.
- Anchor has been deployed in Ant Group, reporting hundreds of bugs in the CI process. We have published the list of the bugs detected by Anchor online [44], along with the detailed empirical data of anchored containers.

The rest of the article is organized as follows. Section 2 shows the motivating example and the outline of our approach. Section 3 presents the preliminary background, and Section 4 formulates the problem we focus on. Section 5 defines the abstract memory, and Section 6 presents the details of the memory orientation analysis. Section 7 discusses the demand-driven reachability analysis and presents two typical clients of value-flow analysis, including thin slicing and value-flow bug detection. Sections 8 and 9 demonstrate the details of the implementation and the evaluation, respectively, followed by the discussion of several lines of related studies in Section 10. Section 11 provides the conclusion of the work.

[3] For simplicity, we omit the constraint ϕ in Figure 3 if $\phi = T$.

2 OVERVIEW

The section presents the container categorization, illustrates a motivating example, and finally outlines our overall idea.

2.1 Category of Containers

We adopt the classification in [35] and categorize the containers into two types, namely *position-dependent containers* and *value-dependent containers*. In a position-dependent container, e.g., ArrayList and Stack in JCF, each element has a position, representing the location where it is stored. A value-dependent container, e.g., the HashMap in JCF and HttpSession in Java EE, stores its elements based on their values. Particularly, the Set is also a value-dependent container. Generally, a container is manipulated by an interface call at an *index*. An index is a non-negative integer in a position-dependent container, which denotes the position where the interface manipulates the container, or a key in a value-dependent container, which is a comparable object, such as a string and other user-defined types.

2.2 Motivating Example

This section presents a motivating example program and discusses the limitations of existing efforts. Finally, we demonstrate our aim at the end of the section.

Program Description. Figure 2 shows a container-manipulating program, which contains two functions, i.e., foo and bar. Specifically, there are three container objects allocated in the function foo, namely the HttpSession object o_1, the HashMap object o_2, and the Stack object o_3. After inserting several elements into the container objects, the function foo invokes the function bar and passes o_1 and o_3 as the actual parameters. In the function bar, three elements are retrieved from the two container objects, and finally dereferenced at line 21. To simplify the example, we introduce a boolean variable c as the branch conditions, where c does not always evaluate to *true* or *false*.

Following existing efforts on value-flow analysis [41, 45], we leverage the value-flow graph (VFG) to demonstrate how each value propagates in the program, which is shown in Figure 3. Particularly, an edge $(v_1@\ell_i,\ v_2@\ell_j)$ labeled a constraint ϕ in the VFG indicates that the value can flow between v_1 at ℓ_i and v_2 at ℓ_j when ϕ holds, where ℓ_i and ℓ_j denote the positions in the control flow graph. Based on the VFG, we can further perform a variety of value-flow clients. In the NPE detection, a feasible path from *null* value to a dereferenced pointer indicates a possible NPE in the program. For example, the feasible path from $null@\ell_5$ to $p@\ell_{21}$ indicates that the dereference of p causes an NPE at line 21. Unfortunately, it is non-trivial to establish the VFG for a container-manipulating program, as it is required to analyze container memory layouts, which involves multiple sophisticated analyses, such as points-to analysis, numeric analysis, etc.

Limitations of Existing Work. Existing static approaches mainly analyze container memory layouts in two ways, which are not precise or scalable enough for real-world programs. One line of the recent efforts smashes each container into a set of objects and does not reason the indexes [28–32]. Although they often obtain gentle scalability, spurious value flows can plague the results of the clients. For example, a smashing-based analysis can introduce five dash edges in the VFG shown in Figure 3, which indicate the spurious value flows. Specifically, the dash edge from $null@\ell_5$ to $i@\ell_7$ shows the spurious fact that i can be *null* after line 7, as the smashing-based analysis does not analyze the index-value correlation of o_1. Similarly, it discovers that $q@\ell_{21}$ and $r@\ell_{21}$ can be *null* and finally yields two false positives.

The other line leverages symbolic analysis to compute the program values by logical formulae and applies strong updates to memory layouts by enforcing container axioms [34, 35]. Container memory layouts, including index-value correlations, are abstracted by logical formulae explicitly and exhaustively, and case analysis makes the number of the disjunctions exponentially large. For instance, s in Figure 2 does not have a deterministic value,

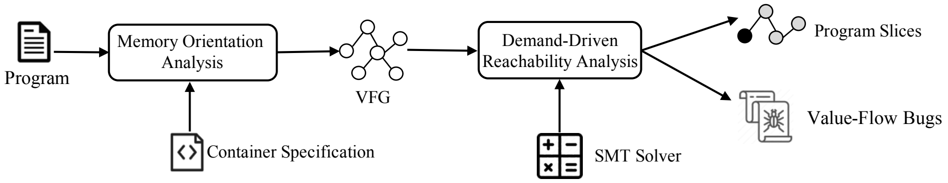


Fig. 4. Schematic overview of our approach

which introduces two disjunctions for o_2 after line 8, corresponding to the cases that s is equal or not equal to "*id*", respectively. Thus, the analyses have to handle the verbose constraints, preventing its scalability significantly.

Our Aim. In this work, we aim to establish fast and precise reasoning of container memory layouts for value-flow analysis. More specifically, we expect the analysis to obtain precise value flows through containers so that it could remove more spurious value flows than smashing-based approaches. Besides, the analysis is required to exhibit high efficiency and gentle scalability when analyzing large-scaled container-manipulating programs. It would have a great impact on many static clients, including thin slicing and value-flow bug detection, and thus, promote the understanding and improve the reliability of container-manipulating programs in the real world.

2.3 Our Approach

Our work balances the tension between precision and scalability by utilizing a programming idiom. In many cases, the modifications of a position-dependent container occur at its beginning or end, while the keys of a value-dependent container are constant. Given a container memory layout, the modified layout is unique if the index of the modification is deterministic. Moreover, if all the modifications upon a container have deterministic indexes before the program location ℓ, its memory layout can be precisely determined at ℓ by tracking the modifications before ℓ along control flow paths. We define the class of containers as *anchored containers*, which enable strong updates and further promote a precise value-flow analysis.

Utilizing anchored containers as our sweet spot, we propose the *memory orientation analysis* to analyze container memory layouts precisely. Figure 4 shows the schematic overview of our approach. In the high level, our approach consists of two phases, namely the memory orientation analysis and the demand-driven reachability analysis. In the first phase, the memory orientation analysis tracks multi-domain properties simultaneously, such as points-to facts and deterministic indexes, which support identifying anchored containers for strong updates. Based on container memory layouts, it constructs a precise value-flow graph, e.g. constructing the graph with the solid edges in Figure 3 for the program in Figure 2. In the second phase, we conduct a demand-driven reachability analysis by the graph traversal to solve an instance of the value-flow problem. Note that we do not invoke SMT solvers in the memory orientation analysis but store the constraints compactly in the graph, which are only collected and checked on demand in the traversal to avoid unnecessary overhead.

Benefit. The memory orientation analysis unleashes the strength of anchored containers with twofold benefits:

- Precise memory layouts of anchored containers support discovering more precise value flows and promotes a chain of further analyses in the clients. For example, the anchored container o_1 in Figure 2 finally enables the analysis to avoid the false positives in the NPE detection.
- The precision benefit can be further amplified, which promotes the reasoning the memory layouts of other containers. For example, the non-anchored container o_2 exclude the *null* value defined at line 5, as the anchored container o_1 makes the *null* not reach i at line 7.

$$
\begin{aligned}
\textit{Program}\ P &:= F+\\
\textit{Function}\ F &:= f(v_1, v_2, \ldots)\{S;\}\\
\textit{Statement}\ S &:= v = \textbf{new}\ \ \tau \mid v = u \mid v = a\\
&\quad \mid c.\textbf{insert}(u, v) \mid c.\textbf{remove}(u) \mid v = c.\textbf{access}(u)\\
&\quad \mid S_1; S_2 \mid \textbf{if}\ (v)\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \mid r = \textbf{call}\ f(v_1, v_2, \ldots)\\
&\quad \mid \textbf{foreach}\ \ (u, v)\ \ \textbf{in}\ \ c\ \ \textbf{do}\ \ S\ \ \textbf{od} \mid \textbf{return}\ v
\end{aligned}
$$

Fig. 5. The syntax of the language

The motivating example illustrates the workflow of ANCHOR. From the example, we find it is crucial but non-trivial to identify and utilize anchored containers. In Section 4, we formulate our problem and formally define anchored containers, following the outline of other sections.

3 PRELIMINARIES

The section presents several preliminary concepts, including program syntax, concrete memory, concrete semantics, and value-flow graph.

3.1 Program Syntax

We borrow the language syntax in [35] and formalize our analysis with a simple Java-like language in Figure 5. A program is in the static single assignment form [46]. Statements include allocation statements, assignments, container interface calls, sequencing, branches, function calls, container traversals, and return statements. Particularly, the right-hand side of an assignment can be a variable or a literal.

We analyze two types of containers, namely position-dependent containers and value-dependent containers. Container interface calls have three cases, namely inserting, accessing, and removing elements. Each container interface can manipulate a container at a specific index. Particularly, we add an artificial index τ_e to represent the end of a position-dependent container so that the language supports adding an element at the end more flexibly. Besides, a loop can traverse a container, and all the elements are accessed once exactly. We assume that a loop in the program is memoryless [47], i.e., the order of the iterations does not affect the semantics of the loop. The language also supports the nesting of containers because v in the insertion can point to another container.

Remark. Our work mainly focuses on the semantic analysis of container manipulations. We do not discuss how to handle the fields of memory objects, although we achieve the field sensitivity based on existing techniques [41, 48]. The language syntax in Figure 5 also omits several program constructs, such as reflective method calls, which are not the major concerns of our work. With the benefit of the formulation of the above syntax, our analysis can ensure the soundness in analyzing all the program constructs shown in Figure 5, which is proven in Section 6.7.2.

3.2 Concrete Memory and Concrete Semantics

Given a program P, a program location ℓ is the position in the control flow graph. We regard program memory as a collection of values $\nu \in \mathcal{V}$ bound with addresses $\alpha \in \mathcal{D} \subseteq \mathcal{V}$ and indexes $\delta \in \Delta \subseteq \mathcal{V}$. Specifically, a memory location is denoted by a pair $(\alpha, \delta) \in \mathcal{D} \times \Delta$. An index δ can be an address or a literal in the program. We introduce two sets $\mathcal{D}_p$ and $\mathcal{D}_v$ to denote the set of the addresses where the position-dependent and value-dependent containers are stored, respectively.

Definition 3.1 (Concrete Memory State). A concrete memory state M at the program location ℓ is (E, L), where

$$\text{Ins}\ \frac{\begin{array}{c} E(c) = \alpha_c \quad E(u) = v_u \quad E(v) = v_v \quad L'(\alpha_c,\ v_u) = v_v \\ L' = L[(\alpha_c,\ j) \rightarrow (j,\ sec(L)(\alpha_c,\ j-1)) \mid j > v_u + 1] \ \text{ if } \ \alpha_c \in \mathcal{D}_p \\ L' = L[(\alpha_c,\ v_u) \rightarrow (v_u,\ v_v)] \text{ if } \alpha_c \in \mathcal{D}_v \end{array}}{(E,\ L) \vdash c.\mathbf{insert}(u, v) : (E,\ L')}$$

$$\text{Rem}\ \frac{\begin{array}{c} E(c) = \alpha_c \quad E(u) = v_u \\ L' = L[(\alpha_c,\ j) \rightarrow (j,\ sec(L)(\alpha_c,\ j+1)) \mid j \geq v_u] \ \text{ if } \ \alpha_c \in \mathcal{D}_p \\ L' = L[(\alpha_c,\ v_u) \rightarrow (v_u,\ \bot)] \ \text{ if } \ \alpha_c \in \mathcal{D}_v \end{array}}{(E,\ L) \vdash c.\mathbf{remove}(u) : (E,\ L')}$$

$$\text{Acc}\ \frac{\begin{array}{c} E(c) = \alpha_c \quad E(u) = v_u \\ L(\alpha_c, v_u) = (v_u, v') \ \ E' = E[v \rightarrow v'] \end{array}}{(E,\ L) \vdash v = c.\mathbf{access}(u) : (E',\ L)} \qquad \text{Proc-II}\ \frac{\begin{array}{c} P = P' \cup \{(\delta',\ v')\} \\ E_1 = E[u \rightarrow \delta',\ v \rightarrow v'] \\ (E_1,\ L) \vdash S : (E_2,\ L_2) \\ (E_2,\ L_2) \vdash proc(u, v, P', S) : (E',\ L') \end{array}}{(E,\ L) \vdash proc(u, v, P, S) : (E',\ L')}$$

$$\text{Proc-I}\ \frac{P = \varnothing}{(E,\ L) \vdash proc(u, v, P, S) : (E,\ L)}$$

$$\text{Loop}\ \frac{\begin{array}{c} E(c) = \alpha_c \quad P = \{(\delta_i,\ v_i) \mid (\delta_i,\ v_i) \in L(\alpha_c,\ \delta_i),\ v_i \neq \bot\} \\ (E,\ L) \vdash proc(u, v, P, S) : (E',\ L') \end{array}}{(E,\ L) \vdash \mathbf{foreach}\ \ (u, v)\ \ \mathbf{in}\ \ c\ \ \mathbf{do}\ \ S\ \ \mathbf{od} : (E',\ L')}$$

Fig. 6. Concrete semantics of container-manipulating programs

- The *environment* E is a function mapping a program variable $u \in \mathcal{X}$ to a value $v \in \mathcal{V}$, indicating the value of the variable. Particularly, $E(u)$ is the address where the object pointed by u is stored if $E(u) \in \mathcal{D}$.
- The *layout* L maps a memory location $(\alpha, \delta) \in \mathcal{D} \times \Delta$ to the pair of an index and a value $(\delta, v) \in \Delta \times \mathcal{V}$. Particularly, $\bot \in \mathcal{V}$ is introduced to show that there does not exist any value stored at the index.

Based on this concrete memory, we can define the operational semantics of container interfaces straightforwardly, which is similar to the definitions in [35]. To simplify the notation, we introduce the mapping $sec(L)$ to get the value stored at a specific index in a container, i.e., $L(\alpha, \delta) = (\delta,\ sec(L)(\alpha, \delta))$. Figure 6 shows the definitions of concrete semantics.

- The rules Ins and Rem define the concrete operational semantics of the insertion and removal, respectively. Due to the difference between position and value-dependent containers, the two rules conduct the case analysis when modifying the layout in the concrete memory.
- For the access interface call, the rule Acc obtains the value v' stored at the index $E(u)$ and enforce the $E'(v)$ equal to the value v'.
- A loop traversing a container takes each index-value pair in the container to execute the statement S in each iteration. Particularly, the helper rule Proc is defined inductively to exercise the traversal.

Example 3.2. In Figure 2, the interface setAttribute adds two key-value pairs in the container object pointed by *hs* at lines 4 and 5. After line 5, we have

$$E(hs) = \alpha,\ L(\alpha, \text{“}id\text{”}) = (\text{“}id\text{”}, \text{“}a\text{”}),\ L(\alpha, \text{“}age\text{”}) = (\text{“}age\text{”}, null)$$

3.3 Value-Flow Graph

A value q flows to p if q is assigned to p directly (via an assignment, such as $p = q$) or indirectly (via container interface calls, such as $c.\mathbf{insert}(0, p); q = c.\mathbf{access}(0)$). We can construct a graph to abstract how the value reaches a program location from another by an edge labeled with a constraint. Formally, we define the value-flow graph as follows.

Definition 3.3. A value-flow graph (VFG) is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \Theta)$, where $\mathcal{N}$, $\mathcal{E}$ and Θ are defined as follows:

- $\mathcal{N}$ is a set of nodes, each of which is denoted by $v@\ell$, indicating v is defined or used at a program location ℓ.
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges. $(v_1@\ell_1, v_2@\ell_2) \in \mathcal{E}$ means that the value $v_1@\ell_1$ flows to $v_2@\ell_2$.
- Θ maps each edge to a constraint ϕ, meaning that the value-flow relation holds only when ϕ is satisfied.

Example 3.4. In Figure 2, we can find that "*a*" is associated with the index "*id*" before line 7, so we add the edge from "*a*"$@\ell_4$ to $i@\ell_7$ to the VFG in Figure 3, indicating the value flow induced by container interface calls at lines 4 and 7.

Following the previous studies [41, 42, 49], we formulate value-flow analysis as a reachability problem over a VFG. We can track the value-flow facts for various clients, such as thin slicing [4], value-flow bug detection [42], etc. For example, we collect all feasible paths from *null* to the dereferenced values in the NPE detection.

4 CONTAINER-AWARE VALUE-FLOW PROBLEM

To obtain precise value-flow paths, we need to identify the indirect flows induced by container interface calls. Concretely, we need to determine the objects accessed by each access interface call, and the reachability relation should perceive the induced value flows. We call this *container-aware value-flow problem*, which is a fundamental concern in static analysis. Any specific instance of value flow analysis has to resolve the problem when analyzing programs using containers.

Based on the semantics of container interface calls, container memory layouts, i.e., the ownership and index-value correlations, determine the value flow through containers. First, an object is never accessed if it is not stored in the container. Second, an object can be accessed by the access interface call at the index δ if associated with δ. Now we formalize the two properties and provide the formal definition of the container memory layout.

Definition 4.1. (Container Memory Layout) Assume that a container object o is stored at the address $\alpha \in \mathcal{D}$ in the concrete memory $M = (E, L)$. The memory layout of the container object o consists of the following two properties:

- Ownership: It indicates whether there exists $\delta \in \Delta$ for a value v such that $L(\alpha,\ \delta) = (\delta,\ v)$. For a value-dependent container object o, it also indicates whether there exists $v' \in \mathcal{V}$ for a value v such that $L(\alpha,\ v) = (v,\ v')$.
- Index-value correlation: For any pair of an index and a value $(\delta,\ v)$, the index-value correlations indicates whether v is paired with δ at the address α, i.e., $L(\alpha,\ \delta) = (\delta,\ v)$.

It is necessary to analyze container memory layouts precisely and efficiently to support analyzing real-world programs using containers. However, precise reasoning of container memory layouts, as other non-trivial semantic properties, is an undecidable problem for general programs [27]. Fortunately, as explained in Section 2.3, there is a typical class of container objects of which modifications occur at deterministic indexes. Their memory layouts are deterministic after the modifications, which enables precise reasoning without case analysis. Formally, we define the notion of *anchored container* as follows.

Definition 4.2. (Anchored Container) A container object o is an anchored container at the program location ℓ if an arbitrary modification interface call *st* before ℓ has the constant index δ. Particularly, τ_e is a constant index.

Intuitively, we can identify anchored containers and analyze their memory layouts precisely. Meanwhile, the precision enhancement introduced by anchored containers also benefits analyzing memory layouts of other containers, even if the containers are not anchored containers.

Example 4.3. In Figure 2, setAttribute modifies o_1 upon constant keys at lines 4 and 5, so o_1 is an anchored container after line 5. Therefore, i must be equal to "a" at line 7. Meanwhile, o_2 is not an anchored container, as s is not constant at lines 8 and 9. It is worth noting that we still obtain more precise ownership of o_2 that the *null* at line 5 is excluded, showing that the precision enhancement can even propagate to non-anchored containers.

Roadmap. To solve the container-aware value-flow problem, we propose the memory orientation analysis, which is our main technical contribution, to identify and apply strong updates to anchored containers. The combined effects of container semantics are analyzed without sophisticated reasoning of indexes, and finally encoded in the VFG. However, as explained in Section 1, it is non-trivial to enable the identification, involving analyzing the facts in multiple domains simultaneously. Specifically, we present a novel memory abstraction to maintain the multi-domain program facts (Section 5), based on which the memory orientation analysis reasons container semantics by applying abstract transformers (Section 6.1~Section 6.5) and constructs a precise VFG (Section 6.6). For a specific client, we conduct a demand-driven reachability analysis by traversing the VFG (Section 7), which benefits from the precise enhancement provided by the memory orientation analysis.

5 ABSTRACT MEMORY

The section presents the abstract memory used in this work (Section 5.1~Section 5.3), and highlights the technical challenges of memory orientation analysis that rests on the abstraction (Section 5.4).

5.1 Abstract Memory State

We abstract the memory based on allocation sites [50] and form a finite set of abstract objects $\mathbf{O} := \mathbf{O}_p \cup \mathbf{O}_v \cup \mathbf{O}_s$, where $\mathbf{O}_p$, $\mathbf{O}_v$, and $\mathbf{O}_s$ are the sets of position-dependent container objects, value-dependent container objects, and non-container-typed objects, respectively. Besides, we construct a finite set of literals $\mathbf{O}_c \subseteq \mathbf{O}_s$, where $\hat{\tau}_e \in \mathbf{O}_c$ represents the end position of a position-dependent container. $\mathbf{X}$ is a set of program variables, and Φ is a set of path constraints. Formally, we define abstract memory state as follows.

Definition 5.1. (Abstract Memory State) An abstract memory state $\mathbf{M}$ at the program location ℓ is a 4-tuple $(\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$. Here, $\mathbf{E}, \mathbf{L}, \mathbf{C}$, and $\mathbf{U}$ are defined as follows.

- *Abstract environment* $\mathbf{E}$ maps a program variable v to a set of abstract memory object (o) and constraint (ϕ) pairs, indicating v points to o when ϕ holds.
- *Abstract layout* $\mathbf{L} := (\mathbf{B}, \mathbf{R})$ contains two subdomains:
  - $\mathbf{B} := \mathbf{O}_p \cup \mathbf{O}_v \rightarrow \mathcal{B}$ abstracts the ownership of each container object, where
    $$\mathcal{B} := \mathcal{P}(\mathbf{O} \times \Phi) \times \mathcal{P}(\mathbf{O} \times \Phi)$$
    Basically, we set the first entry of $\mathbf{B}(o_p)$ to $\{(o, T) \mid o \in \mathbf{O}\}$ for $o_p \in \mathbf{O}_p$, indicating that we only concern the stored objects without considering their positions in a position-dependent containers.
  - $\mathbf{R} := \mathbf{O}_p \cup \mathbf{O}_v \rightarrow \mathcal{R}_p \cup \mathcal{R}_v$ abstracts the index-value correlations of the container objects, where
    $$\mathcal{R}_p := \mathcal{P}(\cup_{k=0}^{N} \mathbf{O}^k \times \Phi), \quad \mathcal{R}_v := \mathcal{P}(\mathbf{O}_c \times \mathbf{O} \times \Phi)$$
    N is the number of insertions upon position-dependent containers, bounding the sizes of the container objects.
- *Constant domain* $\mathbf{C}$ is a function of program variables. $\mathbf{C}(v) \in \mathbf{O}_c$ when v must point to a literal, and $\mathbf{C}(v) = \bot$ when v is not initialized. Otherwise, $\mathbf{C}(v) = \top$.
- *Uniqueness domain* $\mathbf{U}$ maps $o \in \mathbf{O}_p \cup \mathbf{O}_v$ to 1 if o is always modified upon deterministic indexes. Otherwise, $\mathbf{U}(o) = 0$.

Essentially, an abstract environment $\mathbf{E}$ over-approximates the points-to facts, i.e., v may point to a memory object o in a concrete execution if ϕ is satisfied, where $(o, \phi) \in \mathbf{E}(v)$. Similarly, the ownership and the index-value correlation of each container are over-approximated by the separate subdomains of $\mathbf{L}$. Meanwhile, a constant domain $\mathbf{C}$ under-approximates whether a variable points to a literal. Lastly, $\mathbf{U}$ stores the accumulative effects of the modifications upon each container object, and thus, serves as the criteria of identifying anchored containers.

Definition 5.2. (Criteria of Anchored Container) An abstract container object o is an anchored container at the program location ℓ if $\mathbf{U}(o) = 1$ where $\mathbf{M} = (\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$ is the abstract memory at ℓ.

Example 5.3. Consider $o_1 \in \mathbf{O}_v$ pointed by *hs* in Figure 2. ind_1 and ind_2 are the indexes of the insertions at lines 4 and 5, respectively. Obviously, we have $\mathbf{C}(ind_1) = $ "*id*" and $\mathbf{C}(ind_2) = $ "*age*". Therefore, o_1 is always modified upon deterministic indexes after line 5. Concretely, we have

$$\mathbf{B}(o_1) = (\{(\text{“}id\text{”}, T), (\text{“}age\text{”}, T)\}, \{(\text{“}a\text{”}, T), (null, T)\})$$

$$\mathbf{R}(o_1) = \{((\text{“}id\text{”}, \text{“}a\text{”}), T), ((\text{“}age\text{”}, null), T)\}, \quad \mathbf{U}(o_1) = 1$$

Particularly, $\mathbf{U}(o_1) = 1$ indicates that o_1 is an anchored container after line 5.

5.2 Join Operator and Partial Order

Given the definitions of $\mathbf{O}$, Φ, and $\mathbf{M}$, we introduce the join operator and the partial order of the combined domain.

Definition 5.4. (Join Operator of $\mathbf{M}$) $\sqcup_{\mathbf{M}}$ is the join operator of $\mathbf{M}$, i.e., $\mathbf{M} = \mathbf{M}_1 \sqcup_{\mathbf{M}} \mathbf{M}_2$, where $\mathbf{M}_1 = (\mathbf{E}_1, \mathbf{L}_1, \mathbf{C}_1, \mathbf{U}_1)$ and $\mathbf{M}_2 = (\mathbf{E}_2, \mathbf{L}_2, \mathbf{C}_2, \mathbf{U}_2)$. $\mathbf{M} = (\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$ is defined as follows.

$$\mathbf{E}(v) := \mathbf{E}_1(v) \sqcup_{\mathbf{E}} \mathbf{E}_2(v)$$

$$\mathbf{B}(o) := \mathbf{B}_1(o) \sqcup_{\mathbf{B}} \mathbf{B}_2(o) \quad \mathbf{R}(o) := \mathbf{R}_1(o) \sqcup_{\mathbf{R}} \mathbf{R}_2(o)$$

$$\mathbf{C}(v) := \mathbf{C}_1(v) \sqcup_{\mathbf{C}} \mathbf{C}_2(v) \quad \mathbf{U}(o) := \mathbf{U}_1(o) \sqcup_{\mathbf{U}} \mathbf{U}_2(o)$$

Particularly, we define $\sqcup_{\mathbf{E}}$, $\sqcup_{\mathbf{C}}$, and $\sqcup_{\mathbf{U}}$ as follows:

$$\mathbf{E}_1(v) \sqcup_{\mathbf{E}} \mathbf{E}_2(v) := \{(o,\ \phi_1 \vee \phi_2) \mid (o, \phi_1) \in \mathbf{E}_1(v),\ (o, \phi_2) \in \mathbf{E}_2(v)\}$$

$$\mathbf{B}_1(o) \sqcup_{\mathbf{B}} \mathbf{B}_2(o) := \{((o_k, \phi_k^1 \vee \phi_k^2), (o_v, \phi_v^1 \vee \phi_v^2)) \mid ((o_k, \phi_k^i), (o_v, \phi_v^i)) \in \mathbf{B}_i(o),\ i \in \{1,\ 2\}\}$$

$$\begin{aligned}\mathbf{R}_1(o) \sqcup_{\mathbf{R}} \mathbf{R}_2(o) :=& \{(t,\ \phi_1 \vee \phi_2) \mid (t, \phi_1) \in \mathbf{R}_1(o),\ (t, \phi_2) \in \mathbf{R}_2(o),\ t \in \cup_{k=0}^{N} \mathbf{O}^k\} \\ & \cup \{(t,\ \phi_1 \vee \phi_2) \mid (t, \phi_1) \in \mathbf{R}_1(o),\ (t, \phi_2) \in \mathbf{R}_2(o),\ t \in \mathbf{O}_c \times \mathbf{O}\}\end{aligned}$$

$$\mathbf{C}_1(v) \sqcup_{\mathbf{C}} \mathbf{C}_2(v) := \mathbf{ite}(\mathbf{C}_1(v) = \mathbf{C}_2(v), \mathbf{C}_1(v), \top), \quad \mathbf{U}_1(o) \sqcup_{\mathbf{U}} \mathbf{U}_2(o) := \mathbf{ite}(\mathbf{U}_1(o) = \mathbf{U}_2(o), \mathbf{U}_1(o), 0)$$

Specifically, **ite** is the shorthand of *if-then-else* expression. For clarity, we assume that $(o, F) \in \mathbf{E}(v)$ if there does not exist ϕ such that $(o, \phi) \in \mathbf{E}(v)$. For other subdomains, we also use the similar assumptions to make the definitions compact.

Definition 5.5. (Partial Order of $\mathbf{M}$) $\mathbf{M}_1 \sqsubseteq_{\mathbf{M}} \mathbf{M}_2$ if and only if there exists $\mathbf{M}'$ such that $\mathbf{M}_1 \sqcup_{\mathbf{M}} \mathbf{M}' = \mathbf{M}_2$. $\sqsubseteq_{\mathbf{M}}$ also naturally exports the definitions of $\sqsubseteq_{\mathbf{E}}$, $\sqsubseteq_{\mathbf{L}}$, $\sqsubseteq_{\mathbf{C}}$, and $\sqsubseteq_{\mathbf{U}}$.

Intuitively, a join operator is a set union with a disjunction of path constraints. The complexity of a join operator of each subdomain is $O(|D_1| + |D_2|)$, because we can maintain D_1 and D_2 by sorted lists, and a join operator takes linear time. Further, we naturally extend the join operators $\sqcup_{\mathbf{R}}$ and $\sqcup_{\mathbf{E}}$ to the batch join operators $\widetilde{\sqcup}_{\mathbf{R}}$ and $\widetilde{\sqcup}_{\mathbf{E}}$ in order to join multiple elements, which are applied in the rules in Section 6.3 and Section 6.5.

5.3 Layout Operator for Strong Update

According to the definition of $\mathbf{L}$, its second subdomain $\mathbf{R}$ abstracts the index-value correlations of container objects, i.e., how objects are associated with the indexes. To support strong updates, we define the layout operators to describe the semantics of inserting, accessing, and removing the objects, which are applied to update the abstract memory in Section 6.

Definition 5.6. (Layout Operator) Given $o_l \in \mathbf{O}_p$, $o_m \in \mathbf{O}_v$, $o_i, o_k \in \mathbf{O}_c$ and $o_v \in \mathbf{O}$, a layout operator has one of the following forms.

- $\mu(\mathbf{R}, o_l, o_i, o_v)$ and $\mu(\mathbf{R}, o_m, o_k, o_v)$ insert an single object and a pair of objects to o_l and o_m at a deterministic position o_i and key o_k, respectively, producing the new container memory layouts after the insertions.
- $\pi(\mathbf{R}, o_l, o_i)$ and $\pi(\mathbf{R}, o_m, o_k)$ collect the object at a deterministic position o_i and key o_k, respectively, returning a set of abstract objects paired with constraints, which indicate accessed elements and conditions.
- $\omega(\mathbf{R}, o_l, o_i)$ and $\omega(\mathbf{R}, o_m, o_k)$ remove the single object and the pair of objects at a deterministic position o_i and key o_k from o_l and o_m, respectively, producing the new container memory layouts after the removals.

Any operation on an anchored container is essentially a layout operator, which is a function returning a new state in the subdomain $\mathbf{R}$, corresponding to the index-value correlations after the operation. Due to the finite size of $\mathbf{O}_c$, the complexity of a layout operator is bounded by $|\mathbf{R}(o)| \cdot |\mathbf{O}_c|$. Although $|\mathbf{R}(o)|$ depends on the numbers of the insertions and removals in different branches, computing layout operators is still more light-weighted than solving complex constraints qualifying positions and keys with a sheer number of disjunctions.

Example 5.7. In Figure 2, we have $\mathbf{R}(o_1) = \varnothing$ before line 4. The insertion at line 4 induces $\mu(\mathbf{R}, o_1, \text{“}id\text{”}, \text{“}a\text{”})$, which updates $\mathbf{R}(o_1)$ to $\{((\text{“}id\text{”}, \text{“}a\text{”}), T)\}$.

5.4 Summary

According to the abstract states, we can determine the indirect value flows induced by the container interface calls. Specifically, the points-to facts in the abstract environment $\mathbf{E}$ provide sufficient information of adding value-flow edges in the value-flow graph, based on which the client of value-flow analysis can be performed.

Example 5.8. Before line 7 in the program shown in Figure 2, we have

$$\mathbf{R}(o_1) = \{((\text{“}id\text{”}, \text{“}a\text{”}), T), ((\text{“}age\text{”}, null), T)\}$$

After line 7, we can obtain $\mathbf{E}(i) = \{(\text{“}a\text{”}, T)\}$. Finally, we add an edge from “a”@ℓ_4 to $i@\ell_7$ to the VFG in Figure 3.

Technical Challenges. Based on the abstract memory, we have to compute the abstract state for each program location. Specifically, we should resolve the following issues:

- We should identify the manipulated memory objects precisely for container interface calls, posing the challenge in updating points-to facts in $\mathbf{E}$.
- A container object may be modified and accessed at dozens of locations in the program and form different memory layouts, making it challenging to maintain $\mathbf{L}$ precisely and efficiently.
- The facts in the subdomains have sophisticated interactions, which requires a non-trivial semantic reduction operator [38]. For example, the update of $\mathbf{L}$ should be aware of $\mathbf{U}$ to determine whether strong updates should be applied.

6 MEMORY ORIENTATION ANALYSIS

The section presents the memory orientation analysis that addresses the technical challenges discussed in Section 5.4. We first present the abstract transformers of operations not related to containers and show how to maintain precise points-to facts (Section 6.1). We then define the partial abstract transformers of container

$$\text{T-Alloc}\ \frac{\mathbf{E}' = \mathbf{E}[v \to \{(o,T)\}] \quad \mathbf{C}' = \mathbf{C}[v \to \top]}{\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U} \vdash v = \textbf{new}\ \ \tau : \mathbf{E}',\mathbf{L},\mathbf{C}',\mathbf{U}}$$

$$\text{T-Ass-var}\ \frac{\mathbf{E}' = \mathbf{E}[v \to \mathbf{E}(u)] \quad \mathbf{C}' = \mathbf{C}[v \to \mathbf{C}(v) \sqcup_{\mathbf{C}} \mathbf{C}(u)]}{\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U} \vdash v = u : \mathbf{E}',\mathbf{L},\mathbf{C}',\mathbf{U}}$$

$$\text{T-Ass-lit}\ \frac{\mathbf{E}(a) = \{(o_a,\phi)\} \quad \mathbf{E}' = \mathbf{E}[v \to \{(o_a,\phi)\}] \quad \mathbf{C}' = \mathbf{C}[v \to \mathbf{C}(v) \sqcup_{\mathbf{C}} o_a]}{\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U} \vdash v = a : \mathbf{E}',\mathbf{L},\mathbf{C}',\mathbf{U}}$$

Fig. 7. Abstract transformer of allocation and assignment

$$\text{T-Sequencing}\ \frac{\mathbf{M} \vdash S_1 : \mathbf{M}_0 \quad \mathbf{M}_0 \vdash S_2 : \mathbf{M}'}{\mathbf{M} \vdash S_1; S_2 : \mathbf{M}'}$$

$$\text{T-Branch}\ \frac{\begin{array}{c}\phi = \bigvee_{(o,\phi)\in\mathbf{E}(v)} (o \neq \mathit{null} \land \phi) \\ \mathbf{M} \land \phi \vdash S_1 : \mathbf{M}_1 \quad \mathbf{M} \land (\neg\phi) \vdash S_2 : \mathbf{M}_2 \quad \mathbf{M}' = \mathbf{M}_1 \sqcup_{\mathbf{M}} \mathbf{M}_2\end{array}}{\mathbf{M} \vdash \textbf{if}\ (v)\ \textbf{then}\ S_1\ \textbf{else}\ S_2 : \mathbf{M}'}$$

Fig. 8. Abstract transformer of sequencing and branch

interface calls (Section 6.2), and propose a semantic reduction operator, i.e., *witness operator*, to obtain a more precise abstract state by applying strong updates to anchored containers (Section 6.3). Based on the partial abstract transformers and witness operators, we depcit the abstract semantics of a container interface call (Section 6.4). We also define the abstract semantics of container traversals briefly (Section 6.5). Finally, we illustrate how the memory orientation analysis integrates the reasoning of container semantics into the VFG construction (Section 6.6). The benefits of the memory orientation analysis are further summarized along with the formulation and the proof of the soundness (Section 6.7).

In what follows, we describe the abstract transformers as deductive rules of the form:

$$\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U} \vdash\ st : \mathbf{E}',\mathbf{L}',\mathbf{C}',\mathbf{U}'$$

where $(\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U})$ and $(\mathbf{E}',\mathbf{L}',\mathbf{C}',\mathbf{U}')$ are the abstract memory states before and after the statement st, respectively.

6.1 Abstract Semantics of Non-Container Operation

Non-container operations include allocation statements, assignments, sequencing, and branches. We define their abstract transformers as follows.

Allocation Statement and Assignment. Figure 7 presents the abstract transformers for allocation statements and assignments. The rules are straightforward. For instance, when a memory object is allocated, we create an abstract object $o \in \mathbf{O}$. To maintain the points-to relation between v and o, we apply the strong update to the abstract environment $\mathbf{E}$ by setting $\mathbf{E}(v)$ to $\{(o,T)\}$, indicating v must point to o after the statement. Meanwhile, v can not be a variable pointing to a literal. Thus $\mathbf{C}(v)$ is set to $\top$.

The assignment $v = u$ applies the strong update to the points-to set of v by setting $\mathbf{E}(v)$ to $\mathbf{E}(u)$, and the abstract values of v and u in the constant domain are merged. Similarly, we define the abstract transformer for a literal assignment.

$$\text{T-Ins}\ \frac{\begin{array}{c}(o_c,\phi_c)\in\mathbf{E}(c)\quad(o_u,\phi_u)\in\mathbf{E}(u)\quad(o_v,\phi_v)\in\mathbf{E}(v)\\ B=(\{(o_u,\phi_c\wedge\phi_u)\},\{(o_v,\phi_c\wedge\phi_v)\})\\ \mathbf{B}'=\mathbf{B}[o_c\rightarrow\mathbf{B}(o_c)\sqcup_{\mathbf{B}}B]\quad\mathbf{R}'=\mathbf{R}[o_c\rightarrow\triangledown_{\mathbf{R}}(\mathbf{R},o_c)]\\ \mathbf{U}'=\mathbf{U}[o_c\rightarrow\textbf{ite}(\mathrm{C}(o_u)\in\mathbf{O}_c,\ \mathbf{U}(o_c),\ 0)]\end{array}}{\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U}\vdash_t c.\textbf{insert}(u,v):\mathbf{E},\mathbf{L}',\mathbf{C},\mathbf{U}'}$$

$$\text{T-Rem}\ \frac{\begin{array}{c}(o_c,\phi_c)\in\mathbf{E}(c)\quad(o_u,\phi_u)\in\mathbf{E}(u)\\ \mathbf{R}'=\mathbf{R}[o_c\rightarrow\triangledown_{\mathbf{R}}(\mathbf{R},o_c)]\quad\mathbf{U}'=\mathbf{U}[o_c\rightarrow\textbf{ite}(\mathrm{C}(o_u)\in\mathbf{O}_c,\ \mathbf{U}(o_c),\ 0)]\end{array}}{\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U}\vdash_t c.\textbf{remove}(u):\mathbf{E},\mathbf{L}',\mathbf{C},\mathbf{U}'}$$

$$\text{T-Acc}\ \frac{\begin{array}{c}(o_c,\phi_c)\in\mathbf{E}(c)\ \ \mathbf{B}(o_c)=(B_u,B_v)\\ \mathbf{C}'=\mathbf{C}[v\rightarrow\top]\quad\mathbf{E}'=\mathbf{E}[v\rightarrow\{(o_e,\phi_c\wedge\phi_e)\mid(o_e,\phi_e)\in B_v\}]\end{array}}{\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U}\vdash_t v=c.\textbf{access}(u):\mathbf{E}',\mathbf{L},\mathbf{C}',\mathbf{U}}$$

Fig. 9. Partial abstract transformers of container interface call

Sequencing and Branch. In Figure 8, the rule T-Sequencing defines the abstract transformer of a sequence of statements, which is the composition of the abstract transformer of each statement. Similarly, the rule T-Branch defines the abstract transformer of a branch statement. It first conjoins the branch conditions ϕ and $\neg\phi$ with every constraint in the abstract states respectively, and then transforms the abstract states along two branches. Finally, it joins the abstract states at the end of two branches and obtains the abstract state after the branch statement. Particularly, we use the notation $\mathbf{M}\wedge\phi$ as a shorthand of conjoining ϕ with the constraints in $\mathbf{M}$.

6.2 Partial Abstract Transformer of Container Interface Call

We proceed to depict the abstract semantics for container interface calls. The overall idea is to locate the abstract container objects manipulated by the interface call based on the points-to facts in $\mathbf{E}$ and then recompute $\mathbf{L}$. Because we can not determine whether the manipulated container object is an anchored container or not from $\mathbf{U}$ before the statement, we postpone the strong updates on the memory layouts of anchored containers by an additional operator (Section 6.3), and define a partial abstract transformer first without analyzing index-value correlations of the manipulated containers. For clarity, we first define the unary operator $\triangledown_{\mathbf{R}}$ to construct the upper bound of the abstract layout of a container object.

Definition 6.1. (Upper-Bound Operator) Given $o\in\mathbf{O}_p\cup\mathbf{O}_v$ and $\mathbf{R}$, we define the upper-bound operator $\triangledown_{\mathbf{R}}$:

$$\triangledown_{\mathbf{R}}(\mathbf{R},o):=\{(t,\ \textstyle\bigvee_{(t',\phi)\in\mathbf{R}(o)}\phi)\mid t\in\mathcal{T}\}$$

where $\mathcal{T}:=\cup_{k=0}^{N}\mathbf{O}^k$ when $o\in\mathbf{O}_p$ and $\mathcal{T}:=\mathbf{O}_c\times\mathbf{O}$ when $o\in\mathbf{O}_v$. Note that $\triangledown_{\mathbf{R}}(\mathbf{R},o)$ is the conceptual upper bound of the abstract layout of o. We use this notation for defining the rules while do not compute it explicitly.

We describe the partial abstract transformers of container interface calls in the following deductive form:

$$\mathbf{E},\mathbf{L},\mathbf{C},\mathbf{U}\vdash_t\ st:\mathbf{E}',\mathbf{L}',\mathbf{C}',\mathbf{U}'$$

Figure 9 presents the partial abstract transformers for container interface calls. First, the rule T-Ins defines the partial abstract transformer of an insertion. For each abstract objects pointed by c, u, and v, it transforms the abstract memory state in three steps.

- For an abstract container object o_c pointed by c, the rule updates its abstract value in $\mathbf{B}$ by inserting o_u and o_v with the constraint $\phi_c\wedge\phi_u$.
- To update $\mathbf{R}$, T-Ins simply sets the abstract value of o_c to its conceptual upper bound to obtain a sound approximation by the upper-bound operator in Definition 6.1.

$$\text{W-Ins}\ \frac{\begin{array}{c}\mathbf{E}(c) = \{(o_c, \phi_c)\} \quad (o_v, \phi_v) \in \mathbf{E}(v) \quad \mathbf{C}(u) = o_u \\ A(o_v, \phi_v) = \widetilde{\sqcup}_{\mathbf{R}}\{(t, \phi \wedge \phi_v) \mid (t, \phi) \in \mu(\mathbf{R}, o_c, o_u, o_v)\} \\ \mathbf{R}'' = \mathbf{R}'[o_c \longrightarrow \widetilde{\sqcup}_{\mathbf{R}}\{A(o_v, \phi_v) \mid (o_v, \phi_v) \in \mathbf{E}(v)\}] \\ \mathbf{U}'(o_c) = 1 \quad \mathbf{L}'' = (\mathbf{B}', \mathbf{R}'') \quad \mathbf{M}'' = (\mathbf{E}', \mathbf{L}'', \mathbf{C}', \mathbf{U}')\end{array}}{\mathbf{M}, \mathbf{M}' \vdash_w c.\mathbf{insert}(u, v) : \mathbf{M}''}$$

$$\text{W-Rem}\ \frac{\begin{array}{c}\mathbf{E}(c) = \{(o_c, \phi_c)\} \quad \mathbf{C}(u) = o_u \quad \mathbf{R}'' = \mathbf{R}'[o_c \longrightarrow \omega(\mathbf{R}, o_c, o_u)] \\ \mathbf{U}'(o_c) = 1 \quad \mathbf{L}'' = (\mathbf{B}', \mathbf{R}'') \quad \mathbf{M}'' = (\mathbf{E}', \mathbf{L}'', \mathbf{C}', \mathbf{U}')\end{array}}{\mathbf{M}, \mathbf{M}' \vdash_w c.\mathbf{remove}(u) : \mathbf{M}''}$$

$$\text{W-Acc}\ \frac{\begin{array}{c}\mathbf{E}(c) = \{(o_c, \phi_c)\} \quad \mathbf{C}(u) = o_u \quad E = \{(o_e, \phi_c \wedge \phi_e) \mid (o_e, \phi_e) \in \pi(\mathbf{R}, o_c, o_u)\} \\ \mathbf{U}'(o_c) = 1 \quad \mathbf{E}'' = \mathbf{E}[v \longrightarrow E] \quad \mathbf{M}'' = (\mathbf{E}'', \mathbf{L}', \mathbf{C}', \mathbf{U}')\end{array}}{\mathbf{M}, \mathbf{M}' \vdash_w v = c.\mathbf{access}(u) : \mathbf{M}''}$$

Fig. 10. Witness operator of container interface call

- The abstract value of o_c in the uniqueness domain is also updated and set to 0 if the abstract value of o_u in the constant domain is not literal. Otherwise, it is preserved and equal to the original abstract value.

Similarly, we define the rule T-Rem for a removal. The difference is that T-Rem does not change **B** with the assumption that no element is removed to guarantee the soundness.

Lastly, T-Acc defines the partial abstract transformer of an access interface call. $\mathbf{C}(v)$ is set to be $\top$, as v is not assigned by a literal. Meanwhile, T-Acc iterates each abstract container object o_c pointed by c and the abstract objects o_e in its value set, and o_e is exactly the accessed object. Finally, T-Acc applies the strong update to $\mathbf{E}(v)$ by enforcing v point to o_e under $\phi_c \wedge \phi_e$, where ϕ_c and ϕ_e are the paired constraints of o_c and o_e, respectively.

Example 6.2. Before line 5 in Figure 2, we have $\mathbf{U}(o_1) = 1, \mathbf{R}(o_1) = \{((\text{“}id\text{”}, \text{“}a\text{”}), T)\}, \mathbf{B}(o_1) = (\{(\text{“}id\text{”}, T)\}, \{(\text{“}a\text{”}, T)\})$, and $\mathbf{C}(ind_2) = \text{“}age\text{”} \in \mathbf{O}_c$, where ind_2 is the insertion index, so we have $\mathbf{U}'(o_1) = 1$ after line 5. Applying T-Ins, we have

$$\mathbf{B}'(o_1) = (\{(\text{“}id\text{”}, T), (\text{“}age\text{”}, T)\}, \{(\text{“}a\text{”}, T), (null, T)\})$$
$$\mathbf{R}'(o_1) = \triangledown_{\mathbf{R}}(\mathbf{R}, o_1) = \{(t, T) \mid t \in \mathbf{O}_c \times \mathbf{O}\}$$

As defined in the rule T-Ins, the values of the manipulated container object in **B** are merged by $\sqcup_{\mathbf{B}}$. According to Definition 5.4, the disjunctions in constraints enable the strong update on **B**. However, while the rules support precise reasoning about **B**, we can not obtain how objects are stored in each container because **C** and **U** are opaque to **R**.

6.3 Witness Operator

To utilize anchored containers and obtain more precise memory layouts, we instantiate the witness operators [51] for semantic reduction [38]. Based on Definition 5.2 in Section 5.4, $\mathbf{U}'$ determines all the anchored containers at the statement st. Therefore, the key idea underlying the witness operators is to sharpen **R** as long as we find that an abstract container object is anchored (according to the subdomain **U**). Specifically, the witness operator takes as input the abstract memory states $\mathbf{M} := (\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U})$ before the statement and $\mathbf{M}' := (\mathbf{E}', \mathbf{L}', \mathbf{C}', \mathbf{U}')$ after applying the partial abstract transformer, and finally returns a smaller abstract memory state $\mathbf{M}'' := (\mathbf{E}'', \mathbf{L}'', \mathbf{C}'', \mathbf{U}'')$ compared with $\mathbf{M}'$. We describe the witness operator as deductive rules of the following form:

$$(\mathbf{E}, \mathbf{L}, \mathbf{C}, \mathbf{U}), (\mathbf{E}', \mathbf{L}', \mathbf{C}', \mathbf{U}') \vdash_w st : (\mathbf{E}'', \mathbf{L}'', \mathbf{C}'', \mathbf{U}'')$$

Figure 10 presents the witness operators for container interface calls. With the facts in $\mathbf{C}$ and $\mathbf{U}$, it is possible to transform $\mathbf{R}$ to smaller states than $\mathbf{R}'$ in an insertion and removal, and obtain a smaller state $\mathbf{E}''$ than $\mathbf{E}'$ in an access interface call.

- For an insertion, the rule W-Ins first examines whether u points to a literal and the abstract container object o_c pointed by c is deterministic. If o_c is an anchored container, the layout operator μ updates its memory layout maintained by $\mathbf{R}$. Because the number of the abstract memory objects pointed by v can be larger than 1, all the updated abstract layouts $\mathbf{R}$ of o_c are joined by $\widetilde{\sqcup}_{\mathbf{R}}$. Similarly, the rule W-Rem applies the strong update to $\mathbf{R}$ and removes the stored objects from the memory layout of an anchored container by ω.
- The rules of the witness operators for access interface calls are straightforward. If u points to a literal and c points to an anchored container, W-Acc utilizes the layout operator π to collect the memory objects o_e paired with the index o_u in $\mathbf{R}$ and then applies the strong update to $\mathbf{E}(v)$ by enforcing v point to o_e.

Example 6.3. After applying the rule T-Ins for the insertion at line 5 in Figure 2, we have $\mathbf{U}'(o_1) = 1$, indicating that o_1 is an anchored container. Thus, we have

$$\mathbf{R}''(o_1) = \widetilde{\sqcup}_{\mathbf{R}}\{(t, \phi \wedge T) \mid (t, \phi) \in \mu(\mathbf{R}, o_1, \text{“}age\text{”}, null)\} = \{((\text{“}id\text{”}, \text{“}a\text{”}), T), ((\text{“}age\text{”}, null), T)\}$$

Generally, a semantic reduction operator in the combined domain can be expensive, as it requires pair-wise or clique-wise operators [38, 52]. We notice that join operators are applied multiple times in the rule W-Ins, iterating the k-tuples or pairs of abstract objects in $\mu(\mathbf{R}, o_c, o_u, o_v)$ and computing the constraint of each k-tuple or pair. A naive design has quadratic time complexity in the total number of k-tuples or pairs. We optimize witness operators by maintaining a hash value for each k-tuple and pair so that the join operator can be applied in linear time complexity. Specifically, if (t_1, ϕ_1) and (t_2, ϕ_2) satisfy the condition that t_1 and t_2 have the same hash value, we are aware that t_1 and t_2 are equal and then create the disjunction $\phi_1 \vee \phi_2$ paired with t_1 (i.e., t_2).

6.4 Abstract Semantics of Container Interface Call

For each container interface call, we can easily compute its abstract semantics by applying witness operators after the partial abstract transformers. Specifically, we have the abstract transformer $\mathbf{M} \vdash st : \mathbf{M}'$ for a container interface call st if and only if $\mathbf{M} \vdash_t st : \mathbf{M}_0$ and $\mathbf{M}, \mathbf{M}_0 \vdash_w st : \mathbf{M}'$.

In contrast to the partial abstract transformers in Figure 9, the witness operators demand the post-states of the subdomains after applying the partial abstract transformers, such as $\mathbf{U}'$, to identify anchored containers for strong updates. However, the partial abstract transformers in Figure 9 induces the transition of each subdomain independently, permitting the parallel updates of the subdomains. We separate the witness operators from the partial abstract transformers to achieve better efficiency in the transition.

It is worth noting that witness operators only affect the abstract memory states when the container interface calls manipulate an anchored container at a deterministic index. For example, we can safely skip other computations of witness operators if u does not point to a literal. Although we compose witness operators with partial abstract transformers eagerly, we can end unnecessary witness operators by scheduling the computations of the premises of the rules in Figure 10.

With the benefit of witness operators, we can obtain more precise container memory layouts by achieving the semantic reduction. We state the correctness of the witness operators in Figure 10 as the following theorem.

Theorem 6.1. *Given a container interface call st and an abstract memory state $\mathbf{M}$ before st, we have*

$$\mathbf{M} \vdash_t st : \mathbf{M}' \ \wedge\ \mathbf{M}, \mathbf{M}' \vdash_w st : \mathbf{M}'' \ \Rightarrow\ \mathbf{M}'' \sqsubseteq_{\mathbf{M}} \mathbf{M}'$$

Proof. We sketch the proof for the case in which st is an insertion. Without the loss of generality, we only prove the correctness of W-Ins for position-dependent containers. For value-dependent containers, the proofs

$$\text{Fix}\,\frac{\mathbf{M} \vdash S : \mathbf{M}_1 \quad \mathbf{M} = \mathbf{M}_1}{\mathbf{M} \vdash fix(S) : \mathbf{M}} \qquad \text{Fix}\,\frac{\mathbf{M} \vdash S : \mathbf{M}_1 \quad \mathbf{M} \neq \mathbf{M}_1 \quad (\mathbf{M} \triangledown \mathbf{M}_1) \vdash fix(S) : \mathbf{M}'}{\mathbf{M} \vdash fix(S) : \mathbf{M}'}$$

$$\text{T-Loop}\,\frac{\begin{array}{c}(o_c, \phi_c) \in \mathbf{E}(c) \quad (B_u, B_v) = \mathbf{B}(o_c) \\ E_u = \widetilde{\sqcup}_{\mathbf{E}}\{(o_u, \phi \wedge \phi_c) \mid (o_u, \phi) \in B_u\} \quad E_v = \widetilde{\sqcup}_{\mathbf{E}}\{(o_v, \phi \wedge \phi_c) \mid (o_v, \phi) \in B_v\} \\ \mathbf{E}_1 = \mathbf{E}[u \longrightarrow E_u, v \longrightarrow E_v] \quad \mathbf{C}_1 = \mathbf{C}[u \longrightarrow \top, v \longrightarrow \top] \quad (\mathbf{E}_1, \mathbf{L}, \mathbf{C}_1, \mathbf{U}) \vdash fix(S) : \mathbf{M}'\end{array}}{\mathbf{M} \vdash \textbf{foreach}\ \ (u, v)\ \ \textbf{in}\ \ c\ \ \textbf{do}\ \ S\ \ \textbf{od} : \mathbf{M}'}$$

Fig. 11. Abstract transformer of container traversal

can be provided in almost the same way. Also, we can construct the similar proofs for the other two kinds of container interface calls.

According to the definitions of W-Ins in Figure 10, we only need to prove that $\mathbf{L}'' \sqsubseteq_{\mathbf{L}} \mathbf{L}'$. Based on the definition of W-Ins, we have $\mathbf{L}'' = (\mathbf{B}', \mathbf{R}'')$. Therefore, we only have to prove that for an arbitrary $o_p \in \mathbf{O}_p$ and $((o_1^1, \dots, o_1^k), \phi_1) \in \mathbf{R}''(o_p)$, there exists $((o_2^1, \dots, o_2^k), \phi_2) \in \mathbf{R}'(o_p)$ such that $o_1^i = o_2^i (1 \leq i \leq k)$ and ϕ_1 implies ϕ_2. Based on the definition of T-Ins and W-Ins, $\mathbf{R}''$ and $\mathbf{R}'$ only differ at o_c pointed by c, where $\mathbf{E}(c) = \{(o_c, \phi_c)\}$. We have

$$A(o_v, \phi_v) = \widetilde{\sqcup}_{\mathbf{R}}\{(t, \phi \wedge \phi_v) \mid (t, \phi) \in \mu(\mathbf{R}, o_c, o_u, o_v)\}$$
$$\mathbf{R}'' = \mathbf{R}'[o_c \longrightarrow \widetilde{\sqcup}_{\mathbf{R}}\{A(o_v, \phi_v) \mid (o_v, \phi_v) \in \mathbf{E}(v)\}]$$

Consider an arbitrary $((o_1, o_2, \dots, o_m), \phi'') \in \mathbf{R}''(o_c)$, we can derive the following fact from the definition of μ:

$$\exists((o_1, o_2, \dots, o_{o_i - 1}, o_{o_i + 1}, \dots, o_m), \phi) \in \mathbf{R}(o_c), \quad \phi \wedge \phi_v = \phi''$$

Based on the definition of $\triangledown_{\mathbf{R}}$, we have $\mathbf{R}'(o_c) = \triangledown_{\mathbf{R}}(\mathbf{R}, o_c) = \{(t, \bigvee_{(t', \phi) \in \mathbf{R}(o_c)} \phi) \mid t \in \cup_{k=0}^{N} \mathbf{O}^k\}$. Let $t = (o_1, o_2, \dots, o_m)$, and $\phi' = \bigvee_{(t', \phi) \in \mathbf{R}(o_c)} \phi$. We have ϕ implies ϕ' according to the property of the logical disjunction. Meanwhile, we have ϕ'' implies ϕ based on $\phi \wedge \phi_v = \phi''$, so we get ϕ'' implies ϕ'. Thus, we have proved that for each $((o_1, o_2, \dots, o_m), \phi'') \in \mathbf{R}''(o_c)$, there exist $((o_1, o_2, \dots, o_m), \phi') \in \mathbf{R}'(o_c)$ such that ϕ'' implies ϕ'. □

Remark. The precision enhancement provided by witness operators also propagates to non-anchored containers because of the interactions among the subdomains **E**, **B**, and **R**. With the benefit of precise index-value correlations in **R**, the witness operators in Figure 10 also generate more precise points-to facts in **E**, based on which the transformers in Figure 9 produce more precise ownership in **B** for general containers.

Another interesting benefit of witness operators is that the objects with fields can be precisely analyzed in a field sensitive manner. Essentially, such an object is a particular kind of an anchored container. The constant indexes of the manipulations are exactly the field names of the object. In real-world programs, container objects are often used as the fields of a user-defined object. A field-insensitive analysis can hardly apply strong updates upon container memory layouts, as it does not identify which container objects are pointed by the field precisely. Therefore, the witness operators can compute the precise points-to facts for each field, which further enables strong updates for the container objects pointed by the fields.

6.5 Semantics of Container Traversal

For a traversal, the rules in Figure 11 iterate the loop body to obtain the fixed point. For example, the rule T-Loop first enumerates the objects stored in a container and enforces u and v point to these objects. Besides, u and v can not point to certain literals, so $\mathbf{C}(u)$ and $\mathbf{C}(v)$ are set to $\top$. The rule Fix finally computes the fixed point for the loop body.

Table 1. Rules of computing value-flow edges

Statement	**Condition**	**Edge**
$v = a;$	$\exists o_a,\ (o_a, \phi_a) \in \mathbf{E}(a) \land (o_a, \phi_v) \in \mathbf{E}(v)$	$a \rightarrow v$: $\phi_a \land \phi_v$
$v = u;$	$\exists o,\ (o, \phi_v) \in \mathbf{E}(v) \land (o, \phi_u) \in \mathbf{E}(u)$	$u \rightarrow v$: $\phi_v \land \phi_u$
$v = c.access(u)$	$\exists o \exists w,\ (o, \phi_v) \in \mathbf{E}(v) \land (o, \phi_w) \in \mathbf{E}(w)$	$w \rightarrow v$: $\phi_v \land \phi_w$

To assure the termination, we define and apply the widening operator $\triangledown$ [53] if the abstract state $\mathbf{M}$ before an iteration is not equal to the abstract state $\mathbf{M}_1$ after the iteration. The path constraint after widening is set to be *true* if it is changed in the iteration. The finite sizes of $\mathbf{X}$ and $\mathbf{O}$ guarantee that $\mathbf{M}_1 = \mathbf{M}$ must hold after applying the rule Fix finite times, and the rule T-Loop must be terminating.

6.6 Value-Flow Graph Construction

Based on the points-to facts in $\mathbf{E}$, we can identify the accessed container object, add the value-flow edges labeled with constraints, and finally, stitch value flows from different functions by function summaries to construct a global VFG. Compared with previous value flow analyses [41, 48, 54], the memory orientation analysis extends the abstract memory model to analyze the semantics of container interface calls, finally discovering the value flows through containers precisely.

Following the existing approaches [41, 48], the memory orientation analysis computes the abstract memory and constructs the VFG in two phases as follows.

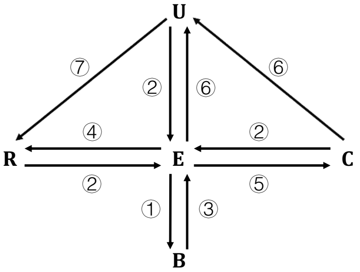
Intraprocedural Analysis. We construct the VFG for a function straightforwardly based on the rules in Table 1. For an assignment, we check the points-to sets of the variables on the left and right-hand sides and add the edge if they are not disjoint. For an access interface call, a value-flow edge is produced between w and v when their points-to sets are not disjoint after the statement. The guarded constraints of the edges are exactly the conjunctions of the constraints of the points-to facts in $\mathbf{E}$. Particularly, we do not invoke SMT solvers on the constraints but store them in the graph as an edge label. To simplify the constraints, we follow the previous work [41] and utilize lightweight semi-decision procedures to filter out apparent contradictions.

Interprocedural Analysis. In the presence of function calls, we introduce the abstract objects for the formal parameters at the function entry and the formal return values at the function exit. Following existing techniques [39–41], we also add auxiliary parameters and return values to support depicting side effects. Then, we compute the abstract memory in the intraprocedural analysis and build the function summary according to the abstract memory [35], which are the edges between the parameters to the returns in the VFG of a single function, abstracting the effect of calling the function [55]. Finally, we inline the function summary of the callee at each call site located in the caller function in a bottom-up manner, yielding a global VFG of the whole program.

Example 6.4. For the function bar in Figure 2, the last element of o_3 is accessed at line 18, so we add an auxiliary parameter node ids_arg1 to the VFG in Figure 3. Based on the abstract state before line 14, we can determine that $i@\ell_7$ and $j@\ell_{11}$ can be the last element, inducing two edges from $i@\ell_7$ and $j@\ell_{11}$ to ids_arg1, respectively. Similarly, we add the other two auxiliary parameter nodes, i.e., hs_arg1 and hs_arg2, and connect them with $null@\ell_5$ and "a"$@\ell_4$, respectively, to form the interprocedural value flow.

6.7 Summary

6.7.1 Benefit of the Combined Domain. Our memory orientation rests on the combined domain to apply strong updates to the memory layouts of anchored containers, allowing a more precise solution than the one obtained by solving each subdomain separately. Essentially, $\mathbf{C}$ supports the constant propagation [56] and affects the state



Rule Set ID	Rules
①	{T-Ins}
②	{W-Acc}
③	{T-Acc}
④	{T-Ins, T-Rem, W-Ins, W-Rem}
⑤	{T-Ass-lit}
⑥	{T-Ins, T-Rem}
⑦	{W-Ins, W-Rem}

Fig. 12. The interactions between the subdomains and the corresponding rules

transition in the uniqueness domain **U**. Even if an index is computed at runtime, we can still effectively identify whether it is constant or not. Furthermore, **E** and **U** enable us to introduce the witness operators for semantic reduction [38] and obtain more precise abstract states.

Figure 12 shows the interactions between the subdomains, which are indicated by the edges labeled with the sets of the rules. In total, there are ten edges in Figure 12 showing ten ways of interactions of the subdomains. We discuss two typical interactions as follows.

- The edge from **U** to **R** indicates the precision benefit introduced by the rules W-Ins and W-Rem. They apply the strong updates upon the memory layouts of the anchored containers and compute their precise index-value correlations in **R**.
- The edge from **E** to **B** indicates that we can obtain more precise ownership even if the container is not an anchored container, as the rule T-Ins propagates the precision benefit from **E** to **B**.

Overall, the combined domain and its abstract transformers serve as a critical role in the memory orientation analysis, eventually promoting the precision of value flow analysis.

6.7.2 Soundness of the Memory Orientation Analysis. Lastly, we discuss the soundness of the memory orientation analysis. An anchored container is essentially a data structure with a set of fields. According to the partial abstract transformers in Figure 9, the uniqueness domain **U** indicates whether a container object must be an anchored container or not. This implies that we perform strong updates conservatively, as we under-approximate the set of anchored containers. As long as we identify anchored containers, we finally reason their index-value correlations in the way of existing field-sensitive analyses [30, 57], which ensures the soundness theoretically.

Formally, we state the soundness of abstract semantics by the following theorem.

Theorem 6.2. *Given a program P, there exists an abstraction function α such that for any concrete memory state M and abstract memory state $\mathbf{M}$ if $\alpha(M) \sqsubseteq_{\mathbf{M}} \mathbf{M}$, then*

$$M \vdash P : M' \;\wedge\; \mathbf{M} \vdash P : \mathbf{M}' \;\Rightarrow\; \alpha(M') \sqsubseteq_{\mathbf{M}} \mathbf{M}'$$

Proof. We sketch the proof of the theorem as follows. The key of the proof is to construct the abstraction function α. According to Definition 3.1 and Definition 5.1, we can define the function $\sigma : \mathcal{V} \to \mathbf{O}$ as follows to abstract the values in the concrete memory by the abstract objects in the abstract memory.

- If v is an address in the concrete memory, σ maps it to the abstract memory object based on the allocation-site abstraction, Specifically, $\sigma(v)$ is the abstract memory object allocated by the statement applying the address v.
- If v is a literal, σ maps v to the corresponding literal object $o \in \mathbf{O}_c$.

Given a concrete memory M at the program location ℓ, we can define $(\widehat{\mathbf{E}}, \widehat{\mathbf{L}}, \widehat{\mathbf{C}}, \widehat{\mathbf{U}}) := \alpha(M)$ by utilizing σ. First, we construct the abstract environment $\widehat{\mathbf{E}}$ based on E. According to the concrete memory M, we can easily computes the path constraint ϕ by creating the conjunction of the equality of the variables in the branch conditions before ℓ. Thus, we define $\widehat{\mathbf{E}}$ as follows for any $\nu \in \mathcal{V}$:

$$\widehat{\mathbf{E}}(\sigma(\nu)) = \{(\sigma(E(v)), \phi)\}$$

Second, we construct the abstract layout $\widehat{\mathbf{L}} := (\widehat{\mathbf{B}},\ \widehat{\mathbf{R}})$. Here, we consider two cases in which $\sigma(\nu)$ is a position-dependent and value-dependent abstract container object, respectively.

- $\sigma(\nu) \in \mathbf{O}_p$: We can obtain the memory layout of the position-dependent container object storing at the address ν based on the concrete memory M, i.e., $L(\nu, i) = (i, \nu_i)$, where $1 \le i \le k$. Therefore, we have

$$\widehat{\mathbf{B}}(\sigma(\nu)) = \{(\{(o, T) \mid o \in \mathbf{O}\},\ \{(\sigma(\nu_i), \phi) \mid 1 \le i \le k\})\}$$
$$\widehat{\mathbf{L}}(\sigma(\nu)) = \{((\sigma(\nu_1), \cdots, \sigma(\nu_k)),\ \phi)\}$$

- $\sigma(\nu) \in \mathbf{O}_v$: We can obtain the memory layout the value-dependent container object storing at the address ν based on the concrete memory M, i.e., $L(\nu, \kappa_i) = (\kappa_i, \nu_i)$, where $1 \le i \le k$. Similarly, we have

$$\widehat{\mathbf{B}}(\sigma(\nu)) = (\{(\sigma(\kappa_i), \phi) \mid 1 \le i \le k\},\ \{(\sigma(\nu_i), \phi) \mid 1 \le i \le k\})$$
$$\widehat{\mathbf{L}}(\sigma(\nu)) = \{((\sigma(\kappa_i),\ \sigma(\nu_i)),\ \phi) \mid 1 \le i \le k\}$$

Third, we construct the state in constant domain $\widehat{\mathbf{C}}$. For any variable v in the program, we enforce $\widehat{\mathbf{C}}(v) = \sigma(\theta)$ if v is always equal to a certain literal θ before ℓ. If v has not been initialized, we set $\widehat{\mathbf{C}}(v)$ to $\bot$. Otherwise, we let $\widehat{\mathbf{C}}(v)$ be $\top$.

Fourth, we can compute $\widehat{\mathbf{U}}(\sigma(\nu))$ in the similar way of defining $\widehat{\mathbf{C}}$. Specifically, $\widehat{\mathbf{U}}(\sigma(\nu)) = 1$ if the container object storing at the address ν satisfy the condition of an anchored container before the program location ℓ. Otherwise, we set $\widehat{\mathbf{U}}(\sigma(\nu))$ to 0.

Therefore, we can obtain the abstraction function α, which maps M to $\widehat{\mathbf{M}} = (\widehat{\mathbf{E}},\ (\widehat{\mathbf{B}},\ \widehat{\mathbf{R}}),\ \widehat{\mathbf{C}},\ \widehat{\mathbf{U}})$. Given the concrete semantics in Figure 6, we can examine the relation $\alpha(M')$ and $\mathbf{M}'$ based on Definitions 9 and 10 straightforwardly. Intuitively, the composition of the partial abstract transformers and the witness operators updates the abstract memory conservatively, preserving the relation that $\alpha(M') \sqsubseteq_{\mathbf{M}} \mathbf{M}'$. Meanwhile, the monotonicity of the join operator $\sqcup_{\mathbf{M}}$ guarantees the soundness of analyzing the program in the presence of the branches and loops. Finally, the sound function summary implies the soundness of analyzing the program with function calls. □

According to the rules of computing value-flow edges in Table 1, we can further obtain the following corollary based on Theorem 6.2, which states the soundness of the overall value-flow analysis. We omit the sketch of its proof in the paper, as it can be obtained from the soundness of computing the abstract memory immediately.

Corollary 6.1. *Given a program P, if a value u at the program location ℓ_1 flows to a value v at the program location ℓ_2 in a concrete execution of P, then there must exist a value-flow edge from $u@\ell_1$ to $v@\ell_2$ in the VFG constructed in the memory orientation analysis.*

7 DEMAND-DRIVEN REACHABILITY ANALYSIS

Once we obtain the VFG by the memory orientation analysis in Section 6, we can reduce the container-aware value-flow problem to a reachability problem [41, 58]. For a specific client, we conduct a demand-driven reachability analysis by traversing the graph and collecting the value-flow facts of interest. The section presents two fundamental clients, namely thin slicing and value-flow bug detection, to demonstrate that our approach benefits program understanding and improves memory safety, respectively. We utilize the motivating example in Figure 2 and its VFG in Figure 3 to illustrate the details of the clients throughout the section.

7.1 Thin Slicing

Program slicing identifies a subset of the program relevant to a program variable and a statement, called the *seed*. It has wide applications in program understanding [4–6] and debugging [12, 13]. Different from traditional slicing, the *thin slice* for a seed includes only the statements that affect the values of the variable directly, called the *producer statements*, and exclude the dependencies of the base pointer and control dependencies [4]. Thus, thin slices are smaller than conventional slices and support more precise program understanding.

To identify the producer statements, we conduct the reachability analysis by backward traversing the VFG from the seed. As blamed in [4], data structures are a major source of slice pollution. The precise reasoning of container semantics enables the slicer to obtain more precise slices for container-manipulating programs.

Example 7.1. Consider the variable q and the statement $ids.peek()$ at line 18 in Figure 2. We can obtain the set of the slices S as follows by a backward traversal from $q@\ell_{18}$:

$$\begin{aligned} S = \{\ & s_1 :\ \text{“}a\text{”}@\ell_4 \hookrightarrow i@\ell_7 \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18}, \\ & s_2 :\ \text{“}b\text{”}@\ell_6 \hookrightarrow j@\ell_{11} \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18}, \\ & s_3 :\ null@\ell_9 \hookrightarrow j@\ell_{11} \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18}\ \} \end{aligned}$$

If the analysis does not distinguish the objects in containers, the thin slice also includes the insertions at line 5, which is a spurious producer statement. Specifically, we have the set of the slices S':

$$S' = S \cup \{s_4 : null@\ell_5 \hookrightarrow i@\ell_7 \hookrightarrow ids\_arg1 \hookrightarrow q@\ell_{18}\}$$

7.2 Value-Flow Bug Detection

Value-flow bugs cover a wide category of program bugs, such as NPE [42], memory leak [59], and taint vulnerabilities [7, 60]. For example, detecting NPE suffices to perform a forward graph traversal, checking the reachability of the value-flow path from *null* to the dereferenced pointer. Similarly, taint vulnerability detection is essentially the problem of analyzing the reachability from the sources to the sinks specified in the taint specifications [29].

In the programs using containers, such as Web applications [8, 32], value flows are often propagated through containers, some of which may trigger the bugs. Our approach strengthens the bug detection for these programs.

Example 7.2. The NPE detector traverses the VFG in Figure 3 from the *null* values to the dereferenced pointers, i.e., p, q, and r at line 21. It discovers that the value flow is reachable from $null@\ell_5$ to $p@\ell_{21}$, forming the path

$$p_1 :\ null@\ell_5 \hookrightarrow hs\_args1 \hookrightarrow p@\ell_{17} \hookrightarrow p@\ell_{21}$$

Thus, the NPE detector reports the NPE without false positives. However, a container mashing-based analysis reports two false positives caused by the following spurious value flows even if it is path-sensitive:

$$\begin{aligned} & p_2 : null@\ell_5 \hookrightarrow hs\_args2 \hookrightarrow r@\ell_{19} \hookrightarrow r@\ell_{21} \\ & p_3 : null@\ell_5 \hookrightarrow i@\ell_7 \hookrightarrow id\_arg1 \hookrightarrow q@\ell_{18} \hookrightarrow q@\ell_{21} \end{aligned}$$

7.3 Summary

The VFG of a container-manipulating program precisely summarizes how values flow through containers, enabling the clients to solve the instances of the value-flow problem by a demand-driven reachability analysis. If necessary, the path constraints are collected on demand and then solved by an SMT solver. Our approach judiciously delays constructing and reasoning about the disjunctions until SMT solving in the reachability analysis. The solver only checks the constraints of certain paths and bypass irrelevant ones, further promoting the scalability of our approach. Also, the precision often goes arm in arm with scalability in the analysis [61, 62]. The strong updates in the memory orientation analysis reduce the facts in each subdomain, yielding a more sparse VFG. Moreover, it can decrease the number of the traversed paths in the reachability analysis, alleviating the overall overhead.

Table 2. List of containers

Framework	Name	Category	Framework	Name	Category
JCF	ArrayList	position	Legacy	Vector	position
JCF	LinkedList	position	Legacy	Stack	position
JCF	HashSet	value	Java EE	ServletContext	value
JCF	TreeSet	value	Java EE	ServletRequest	value
JCF	LinkedHashSet	value	Java EE	HttpServletRequest	value
JCF	HashMap	value	Java EE	HttpServletResponse	value
JCF	TreeMap	value	Java EE	HttpServletRequestWrapper	value
JCF	LinkedHashMap	value	Java EE	HttpServletResponseWrapper	value
Legacy	Properties	value	Java EE	HttpSession	value
Legacy	Dictionary	value	Java EE	JspContext	position, value
Legacy	Hashtable	value	Java EE	PageContext	position, value

8 IMPLEMENTATION

We have implemented Anchor based on the static analysis platform Pinpoint [41, 48] in Ant Group, using Z3 [63] as the SMT solver. Anchor supports a variety of value-flow analyses, such as the value-flow bug detection and thin slicing. Benefited from the original design of Pinpoint, Anchor achieves the context-, flow-, field-, and path-sensitivity in the client analysis. Our work extends the memory model of Pinpoint, enabling the precise reasoning of container memory layouts. In this section, we mainly present the details on the implementations of the memory orientation analysis and the reachability analysis.

Memory Orientation Analysis in Anchor. In the memory orientation analysis, Anchor analyzes the collections in JCF, Java legacy collections, and data structures in Java EE, which are widely utilized in real-world programs [32, 64]. Table 2 shows their names and categories. Specifically, we provide the container specification in a configuration file to specify the concrete semantics of each container interface, such as inserting at the end of a position-dependent container, and removing the pair at a specific key in a value-dependent container. In the memory orientation analysis, Anchor loads the configuration file to identify container interface calls, and updates abstract memory states by applying corresponding partial abstract transformers in Figure 9 and the witness operators in Figure 10. Particularly, Anchor is only concerned with the memory layouts of the containers and does not perform the reasoning of other sophisticated properties, such as the largest key of a TreeMap object and the first-inserted key in a LinkedHashMap object. It is worth mentioning that several kinds of Java EE containers are essentially a composition of multiple position-dependent containers or value-dependent containers. For example, a JspContext object maintains the JspWriter objects sequentially and the attribute objects with the keys. Anchor analyzes such container objects separately, e.g., it splits a JspContext object into two containers, regarding it an object with two container-typed fields.

Reachability Analysis in Anchor. In our reachability analysis, we collect and solve the constraints on demand in the traversal. Instead of leveraging a full-feature SMT solver, we also implement several semi-decision procedures as the intra-procedural preprocessing procedures, such as unit propagation, to determine unsatisfiable or valid constraints in a light-weighted manner [63], most of which can be achieved in linear time. For general cases, we model the variables in the program by bit vectors in the constraints, and set the length of a bit vector to the bit width. To avoid solving the formula in a large size, we adopt an eager strategy to prune the infeasible paths before they reach the sink nodes. Specifically, we solve the condition of a path at specific program locations in the traversal even if it has not reached a sink node. If the current path condition has been unsatisfiable, we can safely stop the traversal, as it can not form a feasible path.

9 EVALUATION

To demonstrate the utility, we address the following research questions:

- **RQ1:** How universe are anchored containers? (Section 9.1)
- **RQ2:** How efficient is Anchor in constructing the VFG? (Section 9.2)
- **RQ3:** How effective is Anchor in thin slicing? (Section 9.3)
- **RQ4:** How precise and scalable is Anchor in detecting value-flow bugs? (Section 9.4)

We conduct four experiments to answer the research question. First, we count anchored containers to show the universality of the concept. Second, we measure the time and memory overhead of the VFG construction for real-world programs. Third, we count the producer statements as the size of a thin slice to measure the precision of thin slicing. Finally, as a case study, we use taint vulnerabilities and null pointer exception to measure the precision and scalability in detecting value-flow bugs.

Subjects. We select 18 open-source Java projects on GitHub that are actively maintained and contain intensive usage of various containers. They cover different sizes (ranging from 19 KLoC to 5.12 MLoC) and diverse application domains (such as RPC frameworks, data management systems, etc.). Besides, many of these projects, such as MyBatis, HBase, and Hadoop, are the fundamental infrastructure of database-backed applications and big data processing systems. They are extensively and frequently scanned by academic static analyzers [10, 65, 66], and industrial tools, and thus expected to have every high quality. We also select several open-sourced projects in Ant Group, including sofa-rpc, atlas, and dubbo, to show our commercial value for the company. Particularly, we choose the OWASP benchmark projects [43] as the subjects to evaluate taint vulnerability detection, as configuring taint specifications for real-world programs might be subjective.

Environment. We evaluate Anchor on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4@2.20GHz and 512GB of physical memory. Following previous studies [41, 67], we set the time limit of an SMT call to 10 seconds. Any analysis is run with a limit of 6 hours and 150GB of memory.

9.1 Identifying Anchored Containers

To show the prevalence of anchored containers, we count the anchored containers at the exit of each project by the memory orientation analysis. Specifically, we examine the uniqueness domain **U** at the exit and count the number of the container objects o which is mapped 1 by **U**. Several container objects are anchored containers at specific program locations, e.g., the HashMap object o_2 at line 7 in Figure 2. However, we do not consider them in such a fine-grained manner, although they can still promote the value-flow analysis. We also count the non-anchored containers in each project and measure the distribution of anchored containers in different types and frameworks.

Table 3 shows the numbers of anchored containers in the column **#AC**. The result shows that anchored containers widely exist in real-world programs. On average, there are 1.13 anchored containers in 1 KLoC, which demonstrates their prevalence. Particularly, over 1 thousand anchored containers exist in the project NetBeans, posing the necessity of analyzing them precisely and efficiently. Besides, The column **#NAC** in Table 3 shows the number of non-anchored containers. Although non-anchored containers take up a larger proportion in real-world programs, the orientation analysis can still effectively improve the precision of many value-flow clients, which will be evidenced by the answers to other research questions.

Figure 13 shows more details of the proportions of various kinds of containers. First, Figure 13 (a) reveals that position-dependent containers, such as ArrayList and LinkedList, are slightly more frequently used than value-dependent containers, such as HashMap and Dictionary. Second, Figure 13 (b) shows that position-dependent containers take up 70.8% of all the anchored containers, while the proportion of value-dependent containers is only 29.2%. The phenomenon mainly comes from the common practice of developers, as they often add or remove

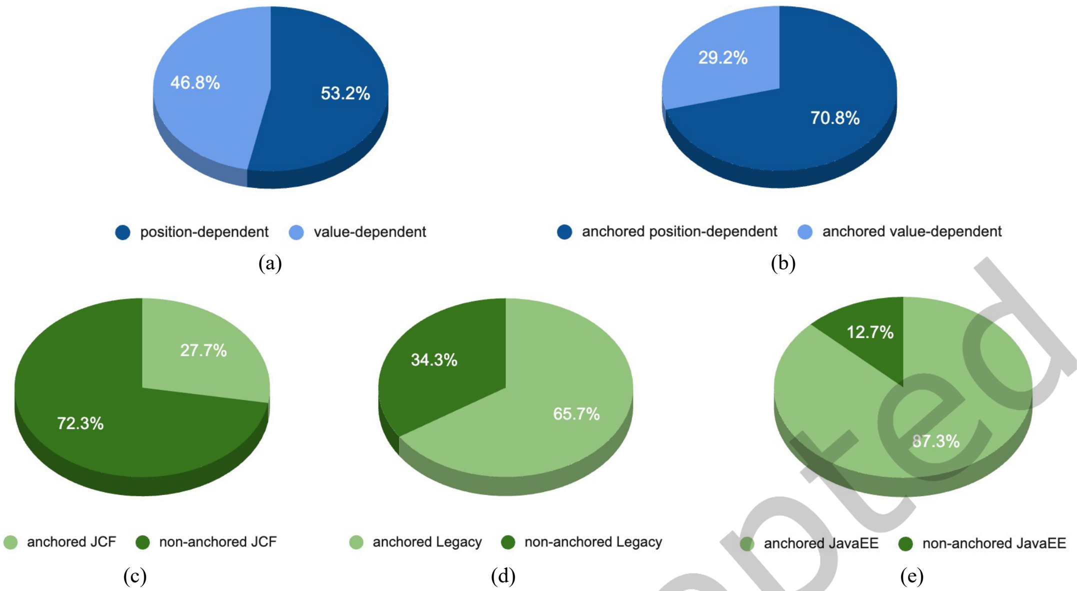


Fig. 13. Proportions of different kinds of containers. (a): Proportions of position-dependent and value-dependent containers; (b): Proportions of anchored position-dependent and anchored value-dependent containers; (c), (d), and (e): Proportions of anchored and non-anchored containers in different frameworks.

an element at the beginning or the end of a position-dependent container, and use more non-literal keys in a value-dependent container. Third, Figure 13 (c), (d), and (e) show the proportions of anchored and non-anchored containers in three frameworks. Specifically, the collections in JCF, which are general-purposed containers, are often used with non-literal keys or constant indexes in a flexible manner, and the proportion of non-anchored containers reaches 72.3%. In contrast, 83.7% Java EE data structures are anchored containers, as they are mostly used to store specific messages, e.g., network requests and responses, which often take literals as keys.

Answer to RQ1: Anchored containers widely exist in real-world programs. There are 1.13 anchored containers in 1 KLoC of the experimental subjects. The proportions of anchored containers in the JCF collections, Java legacy collections, and Java EE data structures are 27.7%, 65.7%, and 83.7%, respectively.

9.2 Constructing Value-Flow Graph

To evaluate the scalability of Anchor, we investigate the overhead of Anchor in the VFG construction. Specifically, we set up two configurations to construct the VFG. In the first configuration (`VFG-O`), we perform the memory orientation analysis to utilize the anchored containers, while in the second configuration (`VFG-S`), we smash the container objects as many existing value-flow analyzers [29, 49]. To better quantify the overhead of analyzing container semantics, we also add the configuration `VFG-N` as a blank control group, in which we do not analyze container interface calls.

The columns **VFG-N**, **VFG-O**, and **VFG-S** in Table 3 show the overhead of time and memory under the three configurations, respectively. We can find that

Table 3. The number of anchored containers and overhead of building the VFG

Project	**Description**	**Size**	**#AC**	**#NAC**	**VFG-N**		**VFG-O**		**VFG-S**	
		(KLoC)			Time	Mem	Time	Mem	Time	Mem
					(min)	(GB)	(min)	(GB)	(min)	(GB)
GraphJet	Graph processing system	19	35	85	0.1	0.4	0.1	0.4	0.1	0.4
mapper	Server application	22	18	77	0.3	0.6	0.4	0.7	0.3	0.6
light-4j	Microservice platform	44	48	182	0.3	1.5	0.3	1.6	0.3	1.6
roller	Server application	54	87	267	1.2	1.7	1.4	1.9	1.2	1.8
MyBatis	ORM framework	61	81	220	1.0	2.8	1.1	3.1	1.1	3.1
sofa-rpc	RPC framework	74	72	238	1.3	3.7	1.5	3.9	1.4	3.8
Glowstone	Server application	86	125	303	1.4	3.1	1.7	3.4	1.5	3.3
DolphinScheduler	Eventing infrastructure	90	132	457	1.3	2.7	1.6	2.9	1.5	2.9
atlas	Server application	142	226	551	1.7	4.1	2.1	4.7	2.0	4.6
Struts	Web framework	170	197	765	2.9	5.1	3.3	5.6	3.2	5.5
dubbo	RPC framework	184	173	736	3.0	5.5	3.5	6.1	3.2	5.9
IoTDB	Data management system	236	445	1,498	6.7	10.0	7.2	10.9	6.8	10.2
Spring-Boot	Web framework	346	396	1,152	7.1	10.4	7.9	12.1	7.5	11.8
Cassandra	Database system	538	534	1,250	11.3	15.3	13.1	18.4	12.5	17.7
Hibernate-ORM	ORM framework	787	452	1,288	13.4	20.8	16.6	23.1	14.5	22.3
HBase	Data management system	791	677	1,626	14.9	21.6	17.2	24.3	15.8	23.7
Hadoop	Data management system	1,811	769	3,041	25.7	34.5	28.6	38.6	27.5	37.4
NetBeans	IDE platform	5,122	1,273	5,029	54.8	81.5	61.1	86.2	57.3	84.9

- The memory orientation analysis introduces negligible overhead compared with the analysis based on smashing containers. Both of them finish the construction for any project in 62 minutes with 86.2G peak memory. For the projects with less than 1 MLoC, two analyses only demand around 18 minutes and 25G memory.
- Compared with the analysis under `VFG-N`, the memory orientation analysis consumes at most 24.9% more time and 20.3% more memory. When analyzing the project NetBeans with 5.12 MLoC, Anchor only spends extra 6.3 minutes and 4.7G peak memory on the precise reasoning of container semantics.
- We also adopt regression analysis to study the observed scalability under `VFG-O`, of which the result is shown in Figure 14. The R-squared values for time and memory are 0.9631 and 0.9691, respectively, which indicates the overhead grows nearly linearly at a gentle rate and shows the potential scalability of Anchor.

The scalability of Anchor in the VFG construction benefits from two major designs. First, we perform a delay reasoning of value-flow paths, and only encode the path condition symbolically in the VFG without any explicit solving process. Second, fewer program facts are generated in the abstract state **M** in the presence of the strong updates, which makes the abstract transformers possibly avoid more computation. The observed linear scalability promotes the practicality of Anchor to analyze the large-scale programs in the real world.

Lastly, it is worth mentioning that we construct the VFG for the whole program, which can support a variety of value-flow clients. When we focus on a specific client, e.g., the NPE detection, we could concentrate on particular value flows, and only analyze containers with specific features, such as the containers that may contain *null* values. However, our experimental data shown in Table 3 and Figure 14 has demonstrated the low overhead of the memory orientation analysis. Moreover, Anchor essentially analyzes non-anchored containers as pointers by container smashing and regards anchored containers as data structures with a set of fields, which does not introduce significant overhead theoretically.

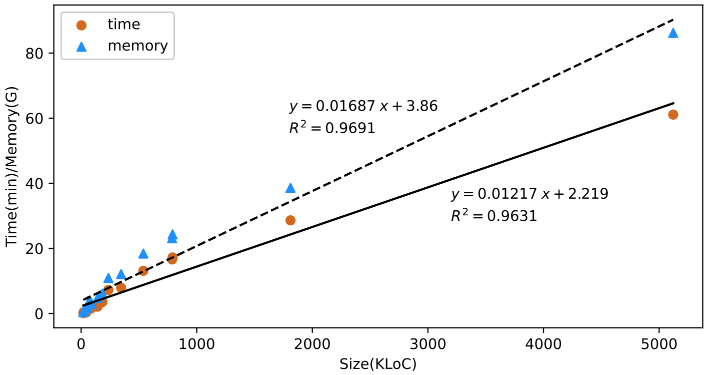


Fig. 14. Scalability of the VFG construction under the configuration VFG-O

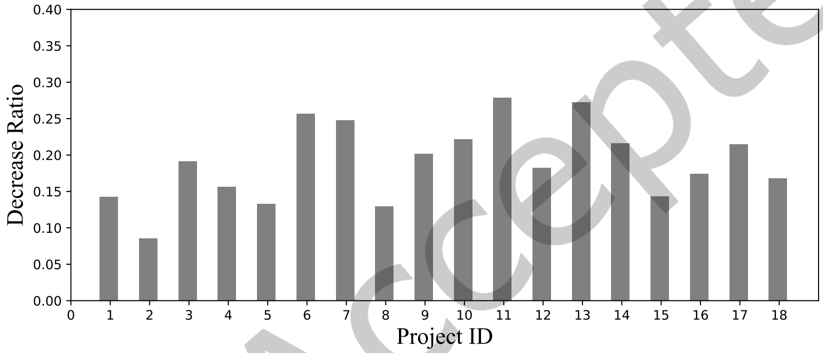


Fig. 15. Decrease ratio of slice sizes under TS-O over TS-S

Answer to RQ2: Anchor features the linear scalability in the value-flow graph construction for real-world programs and finishes the analysis in 62 minutes with 86.2 GB of peak memory for the program with 5.12 MLoC.

9.3 Answering Thin Slicing Queries

To show the precision improvement in the thin slicing, we follow the previous studies [4, 68] and compare the sizes of thin slices under two configurations, i.e., TS-O and TS-S, in which we perform the memory orientation analysis and smash the container objects, respectively. Due to the soundness of two thin slicers, a smaller average size of thin slices indicates a higher precision of the thin slicer. For each project, we randomly select 100 pairs of an access interface call and its return value as the seeds. Particularly, we do not solve constraints to make the analysis light-weighted in the scenario of program understanding.

Figure 15 shows the decrease ratio of thin slice sizes under TS-O over the ones of TS-S, in which project IDs are assigned based on the project sizes. It is shown that 17.1% fewer producer statements are discovered under TS-O than TS-S on average. Besides, we compare the thin slices generated under two configurations. It is found that the thin slices generated under TS-S contain all the statements in the slices generated under TS-O. Therefore, the

Table 4. NPE detection result

Project	NPE-O			NPE-S			Infer		
	Time (min)	Mem (GB)	#FP/#R	Time (min)	Mem (GB)	#FP/#R	Time (min)	Mem (GB)	#FP/#R
GraphJet	0.8	1.3	0/0	0.8	1.2	1/1	0.5	0.8	2/2
mapper	1.2	1.9	0/0	1.1	1.7	1/1	1.0	1.9	1/1
light-4j	3.8	2.3	0/2	3.1	2.7	0/2	4.9	1.6	2/4
roller	4.9	4.1	0/0	4.3	3.8	1/1	2.1	5.7	4/4
MyBatis	9.3	6.0	0/1	8.9	7.4	2/3	7.2	10.1	5/6
sofa-rpc	7.6	5.4	0/1	7.9	6.1	1/2	5.6	4.3	2/2
Glowstone	14.0	7.5	0/0	11.9	8.3	1/1	12.9	13.7	3/3
DolphinScheduler	12.8	6.3	0/0	12.5	6.5	0/0	10.2	8.9	1/1
atlas	14.6	8.1	0/0	13.5	9.3	2/2	NA	NA	NA
Struts	17.9	9.0	0/1	20.1	9.6	0/1	NA	NA	NA
dubbo	18.7	9.3	0/4	19.3	9.5	0/4	NA	NA	NA
IoTDB	26.5	14.5	0/2	26.2	17.8	1/3	28.4	12.1	5/7
Spring-Boot	31.3	16.8	1/1	30.5	16.1	2/2	34.6	20.7	3/3
Cassandra	52.4	21.9	0/2	49.8	21.4	3/5	43.2	33.9	6/7
Hibernate-ORM	66.5	35.2	0/0	69.6	38.1	1/1	51.1	46.3	4/4
HBase	69.9	42.4	0/0	67.1	39.5	1/1	NA	NA	NA
Hadoop	141.7	66.7	0/4	127.8	73.9	8/12	113.7	89.5	14/17
NetBeans	297.3	121.5	1/4	271.4	139.6	15/18	OOM	OOM	OOM
			2/22			**40/60**			**53/62**

NA means we fail to run INFER on the experimental projects.
OOM means INFER runs out of memory.

size decrease of the thin slices indicates that ANCHOR provides more precise slices under the configuration `TS-O` effectively, which benefits from the precise VFG constructed by the memory orientation analysis. Therefore, we conclude that the memory orientation analysis brings better precision in thin slicing than smashing containers. The more precise thin slices can provide better insight for the developers to understand and debug the program in the development.

Answer to RQ3: ANCHOR eliminates 17.1% producer statements that are spurious on average for thin slicing in real-world projects compared with the thin slicer smashing containers.

9.4 Detecting Value-Flow Bugs

Following the previous experiments, we set up the configurations `Taint-O`/`Taint-S` and `NPE-O`/`NPE-S` for the detections of the two types of bugs, respectively.

9.4.1 Detecting Taint Vulnerabilities. To evaluate the effectiveness of ANCHOR in the taint vulnerability detection, we choose the OWASP benchmark projects [43], which is a Java test suite with thousands of exploitable test cases of taint vulnerabilities. The vulnerabilities covered by OWASP benchmark projects include cross-site scripting attacks, information leakage, improper error-handling attacks, etc. Particularly, we select all the programs using containers as the experimental subjects, only concentrating on quantifying the benefit of analyzing container-manipulating programs.

We evaluate ANCHOR upon the collected subjects under two configurations `Taint-O` and `Taint-S`, in which we perform the memory orientation analysis and the container mashing, respectively. The results show that ANCHOR discovers all the 352 taint flows through containers with no false positive under `Taint-O`, while the false-positive ratio reaches 31.0% (158/510) under `Taint-S`. Therefore, the memory orientation analysis can

```
 1 public PropertiesConfiguration() {
 2   Set<String> propertiesNames = getInputStrsFromUser();
 3   List<PropertiesProvider> providers = new ArrayList<>();
 4   for (String name : propertiesNames) {
 5     if ("true".equals(name)) {
 6       providers.add(null);
 7     } else {
 8       providers.add(getExtension(name));
 9     }
10   }
11   Properties properties = ConfigUtils.getProperties();
12   for (PropertiesProvider provider : providers) {
13    properties.putAll(provider.initProperties());
14   }
15   ConfigUtils.setProperties(properties);
16 }
```

Fig. 16. A confirmed NPE in the project dubbo

significantly improve the precision of analyzing the OWASP benchmark projects, demonstrating the practical use in detecting taint vulnerabilities in the presence of containers.

9.4.2 Detecting Null Pointer Exception. To measure the precision and scalability of the NPE detection, we evaluate Anchor upon the real-world projects under the configurations NPE-0 and NPE-S. Particularly, we count the NPE reports caused by the value flows through containers to show the impact of reasoning container semantics. We run the prominent bug detector Infer to compare the overhead and precision [69, 70]. Table 4 shows the numbers of reported bugs, false positives, and the overhead, based on which we summarize the following findings.

Precision and Overhead. Overall, Anchor analyzes the experimental subjects with high efficiency and scalability. First, the reports of the analysis under NPE-0 are subsumed by the ones under NPE-S, and the false-positive ratios are 9.1% (2/22) and 66.7% (40/60), respectively. Upon the submission, 12 of the true positives had been confirmed by the developers [44]. Besides, it is worth noting that most of the bugs are discovered in the popular infrastructure projects, such as MyBatis, Struts and Hadoop, showing the impact of Anchor for infrastructure reliability.

Second, Anchor finishes analyzing the project NetBeans with 512.2 MLoC in 5 hours within 121.5 GB peak memory under NPE-0. An interesting finding in the evaluation is that the memory orientation analysis reduces the overhead in several projects, such as sofa-rpc, Struts, and Hibernate-ORM. The major reason is that the strong updates introduce fewer value-flow facts, which further form fewer value-flow paths, alleviating the overhead of the graph traversal and constraint solving. Also, our work can be further improved by reducing the time and memory overhead. In the evaluation, we only measure the extra overhead introduced by our approach. The techniques of reducing the overhead of static analyzers can benefit Anchor seamlessly in an orthogonal manner [48].

Case Study. We show two typical NPE reports discovered under the two configurations. Figure 16 shows a confirmed NPE bug in the project dubbo, which is reported under NPE-S and NPE-0. If the name of the property is equal to “true”, a *null* value is inserted into the ArrayList object at line 6 and finally dereferenced in the traversal. Under the configuration NPE-S, Anchor maintains the ownership of the container object pointed by providers, and thus finds that provider can be *null* in the traversal, which causes an NPE at line 13. Meanwhile, Anchor also reports the NPE under the configuration NPE-0 due to its soundness. It is also worth mentioning that Figure 16 shows a typical pattern of container usage. Specifically, the elements are dereferenced in the traversal under

```
1 public void action(String w) {
2   Map<String, Object> config = new HashMap<>();
3   config.put("WORD", w == null ? "" : w);
4   config.put("IS_WHOLE", Boolean.FALSE);
5   config.put("BLOCK", null);
6   run((String)config.get("WORD"),
7        (Boolean)config.get("IS_WHOLE"));
8 }
```

Fig. 17. Two false positives in NetBeans reported under the configuration NPE-S

conditions unrelated to their indexes. Even if the container is non-anchored, Anchor does not report a false positive under NPE-O as long as there exists a *null* value in the container.

Figure 17 shows two false positives in the project NetBeans reported under the configuration NPE-S. The values paired with "WORD" and "IS_WHOLE" are not *null*, while the analysis does not distinguish them from the *null* value paired with "BLOCK", causing two false positives at lines 6 and 7. In contrast, Anchor avoids reporting the false positives, as the HashMap object is an anchored container, of which the index-value correlation is precisely tracked by Anchor. Figure 17 demonstrates a common usage pattern of anchored value-dependent containers, which often exists in the configuration modules of a project. The values paired with literal keys are produced by different expressions introducing different program facts, which finally yields specific value-flow bugs at particular keys. Our memory orientation analysis effectively identifies this pattern and analyzes index-value correlations precisely, further supporting precise value-flow bug detection.

Comparison with Infer. The column **Infer** in Table 4 shows that Anchor and Infer share the similar overhead. Infer introduces 53 false positives in 62 reports, and all the true positives reported by Infer are also detected by Anchor. The fundamental reason of its high false-positive ratio is that Infer can not support precise container reasoning, e.g., it reports the two false positives in Figure 17. Also, Infer can not fully track path conditions when the path variables do not collude with preconditions, introducing the infeasible value flows in the NPE detection. After multiple attempts, Infer still fails to analyze several projects because of the crash and out-of-memory, denoted by NA and OOM, respectively. In contrast, Anchor finishes analyzing all the projects in the given budget of time and memory, showing the superiority in terms of scalability.

There are other static analyzers detecting NPEs in real programs [71]. Particularly, Infer analyzes the data structures with bi-abduction reasoning and achieves the field-, flow-, and context-sensitivity with good scalability, sharing the similar style of our work. Thus, we select Infer for comparison to show the advantages of Anchor. We also seek to evaluate Compass [35] while it is outdated for the operating systems we can set up. Also, Compass is proposed to analyze C/C++ programs, and its implementation does not support analyzing Java programs.

Answer to RQ4: Anchor detects all the taint vulnerabilities in the OWASP benchmark programs using containers with no false positive. Also, it uncovers 20 null pointer exceptions in 18 real-world Java programs with 9.1% as its false-positive ratio and finishes analyzing the program with 5.12 MLoC in 5 hours.

9.5 Threats to Validity

There are two major threats to the validity of our approach. The first threat to the validity is whether our approach can be generalized to the containers in a variety of third-party libraries. Actually, our memory orientation analysis only relies on the specifications of container interfaces. As explained in Section 8, the developers can classify the semantics of the interfaces and abstract them by the container interfaces in the language syntax defined in Figure 5. This process does not involve much expert knowledge, and thus supports the generality of our approach.

The second threat to the validity of our work is whether our approach supports a sound value-flow client analysis. Recall that Theorem 6.2 and Corollary 6.1 guarantee the soundness of value-flow analysis. For a specific client, such as the NPE detection, we can discover all the value-flow paths and collect the path conditions by traversing the VFG. However, we set the time budget of solving a constraint in the implementation, possibly discarding feasible paths when the solver fails to solve the path conditions in 10 seconds, which might introduce unsoundness to a specific value-flow client.

9.6 Discussion

This section presents more discussions on the benefit of anchored containers, the limitations, and future work.

Benefit of Anchored Containers. The experiments demonstrate that the anchored containers enable the analysis to obtain high precision with low overhead. Although precise reasoning of general containers requires constructing and solving the constraints in a more sophisticated logic theory, the modification patterns of the anchored containers permit us to specialize the container axioms [72] and acquire the value flows based on the modification history. The combined domain in Section 5 essentially extends the solving procedure and supports constructing more precise VFG in the memory orientation analysis. Meanwhile, the experiment of NPE detection shows that the anchored containers can make the precision go arm in arm with scalability in several analyses of the subjects. As guaranteed by Theorem 6.1, more facts are reduced by the strong updates in the subdomains in the presence of the anchored containers, which prevents the reachability analysis from examining spurious value-flow paths. The phenomena have been discussed in the recent static analyses [61, 62]. The anchored containers play an important role in unleashing the precision and efficiency of the analysis simultaneously.

Limitation of Anchor. Our study shows the effectiveness of Anchor in analyzing container-manipulating programs, but several limitations still exist in our current approach.

- Anchor only analyzes the memory layout of the containers, and it does not concern other properties of containers, such as the size, the emptiness, and insertion order, which also affect the value-flows in the program. Although we have not found the spurious results caused by the unawareness of these properties in the experiment, it can indeed decrease the precision and recall of Anchor theoretically.
- Anchor takes the manually-written container specifications as input to identify the container types and the behavior of their interfaces. For example, we manually investigate all the class definitions in JavaEE to collect the container-like data structures in Table 2. However, the manual annotation often involves a huge laborious effort, especially in the presence of numerous third-party libraries [32, 73].
- Anchor does not reason the index-value correlations of non-anchored containers, which introduces the precision loss when analyzing the value flows through them. As shown in Figure 13 (c), there is a large proportion of JCF collections that are non-anchored. Although the anchored containers can improve the precision of analyzing their ownership, their index-value correlations are still blurred in the memory orientation analysis. This prevents Anchor from further improving the precision of several value-flow analyses, such as thin slicing.

Future Work. It could be promising to explore the following directions to design an automatic, fast, and precise static analyzer for container-manipulating programs. First, it would be meaningful to track more container properties to obtain more precise value flows. For example, the size of a container could support more precise reasoning of the path condition if a branch condition involves its size. Second, the automatic inference of container specifications would enable the static analyzer to identify the container types and analyze client programs in a more automatic manner. More data structures in the third-party libraries, such as Apache Commons Collections [74], Trove [75], and Google Guava Collection [76], could be discovered and identified as containers, which benefits the subsequent analyses [31, 32, 35]. Third, it would be worth designing more advanced analyses to reason the non-anchored containers. The relations among the indexes of container interface calls could support discovering more precise value flows in general cases. For example, a must-not alias analysis would enable us to prune spurious value flows through non-anchored containers according to their keys, which could further increase the precision of value-flow analysis. Fourth, it would also be a promising direction to design an on-demand VFG construction algorithm. The values unrelated to a client can be skipped so that we can safely ignore specific value flows through containers, which can further decrease the overhead of the analysis.

10 RELATED WORK

There is a large body of literature touching the topic of our work, ranging from the heap abstraction to the SMT solving optimization. We discuss each line of the studies in detail as follows.

10.1 Heap Abstraction

Containers are mostly allocated dynamically in the heap. Analyzing heap-manipulating programs statically relies on a bounded abstract heap model [50]. A common heap abstraction is allocation site-based abstraction, which summarizes the heap objects allocated by the same statement into an abstract object [30, 77, 78]. We adopt allocation site-based abstraction to bound the number of abstract objects and guarantee the termination. Another heap abstraction is based on generic predicates, merging the objects which satisfy certain predicates [25, 26, 79–81]. One typical instantiation is the 3-valued logic analyzer (TVLA) [79, 80]. It defines *focus* and *coerce* operators for semantic reduction [38] to obtain more precise 3-valued structures. Anchor's witness operators bear similarities to the coerce operator [51]. In contrast, a coerce operator depends on predefined rules of fundamental inconsistency, while witness operators only utilize the facts in the subdomains and do not rely on manual configurations.

10.2 Value-Flow Analysis

Value-flow analysis resolves the program dependencies to improve the effectiveness of static analyses [29, 41, 59]. It was initially applied in program optimization and debugging by offering def-use relations. With the promotion of the research communities, it has become a fundamental technique of static program analysis and verification [82–84]. Its precision and efficiency are highly dependent on the underlying pointer analysis. To achieve better precision with lower overhead, many value-flow analyses adopt on-demand pointer analyses so that unnecessary pointer facts are not calculated in the value-flow propagation [30, 41, 78]. However, they do not support precise analysis of containers, and smash containers into the sets of objects [29, 31, 45]. Anchor is the first practical value-flow analysis supporting precise container analysis, and its principle can be applied to other techniques [29, 45, 84].

10.3 Data Structure Verification

Another line of efforts aims to verify whether the data structure implementation obeys its specification. Shape analyses, for example, address this problem by inferring or verifying advanced properties [14, 85, 86]. They utilize predicates to describe the linkage property, such as reachability, cyclicity, and treeness [36, 79, 87]. Other sophisticated properties, including the size [51, 88–90], inclusion relation [91, 92], and numeric values [93, 94], are also significant concerns of many shape analyses. The properties contribute to data structure verification, e.g., verifying if the *add* interface inserts an element at the end of the list [95–97]. In this work, we assume that the interfaces conform to their specifications and analyze the client-side programs with the specifications. Data structure verification and client-side program analysis are two orthogonal lines of research work, and the combination provides a more reliable guarantee of the correctness of the programs using containers.

10.4 Specification Inference

It has always been a popular and fundamental problem to infer the data structure specifications. Basically, there are two typical kinds of efforts tracking the problem. The first kind of approaches, including shape analyses [14, 86], abstracts the properties of the data structures by logical formulae, and reasons the specification of each interface of the data structures symbolically. The specifications obtained by symbolic reasoning are sound, while they often only concern the restrictive forms of the linkage properties [79] and numeric properties [95], and are also quite brittle in real-world programs. The second kind of approaches synthesizes the data structure specifications by leveraging learning techniques [73, 98]. Specifically, they utilize the runtime states of the data structure interfaces [98] or the usage of data structures in the client programs to infer the possible specifications [73]. However, they often rely on a given template for the inference, not considering the numeric properties in most cases. We believe it is meaningful to identify whether a data structure is a container and determine the more expressive behavior of its interfaces. A variety of container analyses, such as container-aware pointer analysis [31, 35], and container type optimization [99], can benefit from the specifications inferred automatically.

10.5 SMT Solving Optimization

There have been several studies optimizing the SMT solving process in the program analysis based on the program information. One typical category of the studies leverages the semantic information to assist the SMT solving [100–103]. For example, the interval and data-dependency information are utilized to guide the branching heuristic in the SAT solving phase [101]. Another typical category of the studies achieves the program transformations which preserve the semantic equivalence, and constructs the constraints based on the new program, unleashing the power of the optimized constraint solvers to improve the efficiency of the solving [72, 104]. For instance, when the array only contains the constant values, the array operations are transformed to the comparison of the indexes, which can be represented in the bit-vector theory [72]. The transformation is essentially the process of applying array axioms for the particular kind of arrays, making the analysis benefit from the efficiency of the solving procedure of the bit-vector theory. Anchor bears similarities to the idea in [72]. Specifically, it specializes the container axioms for the anchored containers to simplify the constraints and enables a precise value flow analysis without solving the sophisticated formulae in the combined theory of linear integer arithmetic and uninterpreted functions [35]. It is a promising direction to discover and leverage specific operation patterns to perform efficient and precise reasoning on the data structures.

11 CONCLUSION

We have described Anchor to analyze value flows for containers. Anchor identifies anchored containers automatically for strong updates in the memory orientation analysis and discovers the precise value flows through containers, promoting a variety of client analyses. As a result, it produces more precise thin slices and uncovers 20 NPEs with only two false positives. It outperforms the state-of-the-art value flow analyses and container reasoning techniques in terms of precision and scalability, and features in the ability to analyze millions of lines of code. We expect the underlying insight of Anchor to benefit various clients of analyzing container-manipulating programs, such as program understanding and bug detection.

ACKNOWLEDGMENTS

We thank Professor Eric Bodden and the anonymous reviewers for valuable feedback on earlier drafts, which helped improve its presentation. We also appreciate Dr. Gang Fan for insightful discussions. The authors are supported by the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei. This work was finished when Qingkai Shi was with Ant Group. He is currently with Purdue University and is available via email at shi553@purdue.edu. Peisen Yao is the corresponding author.

REFERENCES

[1] Microsoft. 2022. Containers-C++ Reference. https://www.cplusplus.com/reference/stl/. (2022). [Online; accessed 7-Sept-2022].

[2] Oracle. 2022. Collections Framework Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html. (2022). [Online; accessed 7-Sept-2022].

[3] JavaEE. 2022. Java(TM) EE 8 Specification APIs. https://javaee.github.io/javaee-spec/javadocs/. (2022). [Online; accessed 7-Sept-2022].

[4] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 112–122. DOI:http://dx.doi.org/10.1145/1250734.1250748

[5] Susan Horwitz, Thomas W. Reps, and David W. Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Program. Lang. Syst.* 12, 1 (1990), 26–60. DOI:http://dx.doi.org/10.1145/77606.77608

[6] 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 International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, Richard N. Taylor and Matthew B. Dwyer (Eds.). ACM, 63–72. DOI:http://dx.doi.org/10.1145/1029894.1029907

[7] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, Patrick D. McDaniel (Ed.). USENIX Association. https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static

[8] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 87–97. DOI:http://dx.doi.org/10.1145/1542476.1542486

[9] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: precise refutations for heap reachability. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 275–286. DOI:http://dx.doi.org/10.1145/2491956.2462186

[10] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 38:1–38:17. DOI:http://dx.doi.org/10.1145/3302424.3303972

[11] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: an evolving graph system for flow- and context-sensitive analyses of million lines of C code. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 914–929. DOI:http://dx.doi.org/10.1145/3453483.3454085

[12] Mark Marron, Cesar Sanchez, and Zhendong Su. 2010. High-level heap abstractions for debugging programs. (2010).

[13] Hiralal Agrawal. 1991. *Towards automatic debugging of computer programs.* Ph.D. Dissertation. Purdue University.

[14] Bor-Yuh Evan Chang, Cezara Dragoi, Roman Manevich, Noam Rinetzky, and Xavier Rival. 2020. Shape Analysis. *Found. Trends Program. Lang.* 6, 1-2 (2020), 1–158. DOI:http://dx.doi.org/10.1561/2500000037

[15] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer, 465–484. DOI:http://dx.doi.org/10.1007/978-3-319-26529-2_25

[16] 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 (2018), 140:1–140:29. DOI:http://dx.doi.org/10.1145/3276510

[17] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40. DOI:http://dx.doi.org/10.1145/3381915

[18] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, 661–667. DOI:http://dx.doi.org/10.1007/978-3-642-02658-4_52

[19] Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. 2010. Statically Inferring Complex Heap, Array, and Numeric Invariants. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot and Matthieu Martel (Eds.), Vol. 6337. Springer, 71–99. DOI:http://dx.doi.org/10.1007/978-3-642-15769-1_6

[20] Jiangchao Liu and Xavier Rival. 2015. Abstraction of Optional Numerical Values. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer, 146–166. DOI:http://dx.doi.org/10.1007/978-3-319-26529-2_9

[21] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science)*, Radhia

Cousot (Ed.), Vol. 2694. Springer, 1–18. DOI:http://dx.doi.org/10.1007/3-540-44898-5_1

[22] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2012. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4 (2012), 25:1–25:28. DOI:http://dx.doi.org/10.1145/2377656.2377662

[23] Pieter Hooimeijer and Margus Veanes. 2011. An Evaluation of Automata Algorithms for String Analysis. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science)*, Ranjit Jhala and David A. Schmidt (Eds.), Vol. 6538. Springer, 248–262. DOI:http://dx.doi.org/10.1007/978-3-642-18275-4_18

[24] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: automatic summarization for multipath string loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 188–198. DOI:http://dx.doi.org/10.1145/2771783.2771815

[25] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic Analysis of Open Objects in Dynamic Language Programs. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Markus Müller-Olm and Helmut Seidl (Eds.), Vol. 8723. Springer, 134–150. DOI:http://dx.doi.org/10.1007/978-3-319-10936-7_9

[26] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2015. Desynchronized Multi-State Abstractions for Open Programs in Dynamic Languages. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 483–509. DOI:http://dx.doi.org/10.1007/978-3-662-46669-8_20

[27] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[28] David L. Heine and Monica S. Lam. 2006. Static detection of leaks in polymorphic containers. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 252–261. DOI:http://dx.doi.org/10.1145/1134285.1134321

[29] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. DOI:http://dx.doi.org/10.1145/2594291.2594299

[30] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDE^{al}: efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 99:1–99:27. DOI:http://dx.doi.org/10.1145/3133923

[31] Pratik Fegade and Christian Wimmer. 2020. Scalable pointer analysis of data structures using semantic models. In *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, Louis-Noël Pouchet and Alexandra Jimborean (Eds.). ACM, 39–50. DOI:http://dx.doi.org/10.1145/3377555.3377885

[32] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 794–807. DOI:http://dx.doi.org/10.1145/3385412.3386026

[33] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. 2020. Scaling static taint analysis to industrial SOA applications: a case study at Alibaba. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1477–1486. DOI:http://dx.doi.org/10.1145/3368089.3417059

[34] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Fluid Updates: Beyond Strong vs. Weak Updates. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Andrew D. Gordon (Ed.), Vol. 6012. Springer, 246–266. DOI:http://dx.doi.org/10.1007/978-3-642-11957-6_14

[35] Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise reasoning for programs using containers. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 187–200. DOI:http://dx.doi.org/10.1145/1926385.1926407

[36] 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, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 32–45. DOI:http://dx.doi.org/10.1145/3009837.3009881

[37] Isil Dillig, Thomas Dillig, and Alex Aiken. 2009. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, 233–247. DOI:http://dx.doi.org/10.1007/978-3-642-02658-4_20

[38] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. DOI: http://dx.doi.org/10.1145/567752.567778

[39] Donglin Liang and Mary Jean Harrold. 2001. Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings (Lecture Notes in Computer Science)*, Patrick Cousot (Ed.), Vol. 2126. Springer, 279–298. DOI: http://dx.doi.org/10.1007/3-540-47764-0_16

[40] 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. Exp.* 44, 12 (2014), 1485–1510. DOI: http://dx.doi.org/10.1002/spe.2214

[41] 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, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 693–706. DOI: http://dx.doi.org/10.1145/3192366.3192418

[42] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the extensional scalability problem for value-flow analysis frameworks. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 812–823. DOI: http://dx.doi.org/10.1145/3377811.3380346

[43] OWASP. 2022. Open Web Application Security Project. http://www.owasp.org/. (2022). [Online; accessed 7-Sept-2022].

[44] Anchor. 2022. Bug reports of Anchor. https://containeranalyzer.github.io/. (2022). [Online; accessed 7-Sept-2022].

[45] 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, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. DOI: http://dx.doi.org/10.1145/2892208.2892235

[46] 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 (1991), 451–490. DOI: http://dx.doi.org/10.1145/115372.115320

[47] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. Computing summaries of string loops in C for better testing and refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 874–888. DOI: http://dx.doi.org/10.1145/3314221.3314610

[48] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 930–943. DOI: http://dx.doi.org/10.1145/3453483.3454086

[49] 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, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. DOI: http://dx.doi.org/10.1145/2892208.2892235

[50] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2 (2016), 29:1–29:47. DOI: http://dx.doi.org/10.1145/2931098

[51] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. 2009. A combination framework for tracking partition sizes. 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.). ACM, 239–251. DOI: http://dx.doi.org/10.1145/1480881.1480912

[52] Sumit Gulwani and Ashish Tiwari. 2006. Combining abstract interpreters. *ACM SIGPLAN Notices* 41, 6 (2006), 376–386.

[53] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. DOI: http://dx.doi.org/10.1145/512950.512973

[54] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Efficient Path-Sensitive Data-Dependence Analysis. *arXiv preprint arXiv:2109.07923* (2021).

[55] Thomas W. Reps. 1997. Program Analysis via Graph Reachability. In *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, Jan Maluszynski (Ed.). MIT Press, 5–19.

[56] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170. DOI: http://dx.doi.org/10.1016/0304-3975(96)00072-2

[57] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, Lori L. Pollock and Mauro Pezzè (Eds.). ACM, 133–144. DOI: http://dx.doi.org/10.1145/1146238.1146254

[58] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 480–491. DOI: http://dx.doi.org/10.1145/1250734.1250789

[59] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 72–82. DOI:http://dx.doi.org/10.1109/ICSE.2019.00025

[60] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 576–587. DOI:http://dx.doi.org/10.1145/2635868.2635869

[61] Ondrej Lhoták and Laurie J. Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1 (2008), 3:1–3:53. DOI:http://dx.doi.org/10.1145/1391984.1391987

[62] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 485–495. DOI:http://dx.doi.org/10.1145/2594291.2594320

[63] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. DOI:http://dx.doi.org/10.1007/978-3-540-78800-3_24

[64] Ohad Shacham, Martin T. Vechev, and Eran Yahav. 2009. Chameleon: adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 408–418. DOI:http://dx.doi.org/10.1145/1542476.1542522

[65] Rong Gu, Zhiqiang Zuo, Xi Jiang, Han Yin, Zhaokang Wang, Linzhang Wang, Xuandong Li, and Yihua Huang. 2021. Towards Efficient Large-Scale Interprocedural Program Static Analysis on Distributed Data-Parallel Computation. *IEEE Trans. Parallel Distributed Syst.* 32, 4 (2021), 867–883. DOI:http://dx.doi.org/10.1109/TPDS.2020.3036190

[66] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. 2022. Complexity-guided container replacement synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–31. DOI:http://dx.doi.org/10.1145/3527312

[67] Yichen Xie and Alexander Aiken. 2005. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 351–363. DOI:http://dx.doi.org/10.1145/1040305.1040334

[68] 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, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:27. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.15

[69] Facebook. 2022. Infer Static Analyzer. https://fbinfer.com/. (2022). [Online; accessed 7-Sept-2022].

[70] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. DOI:http://dx.doi.org/10.1145/3338112

[71] David A. Tomassi and Cindy Rubio-González. 2021. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 292–303. DOI:http://dx.doi.org/10.1109/ASE51524.2021.9678535

[72] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 68–78. DOI:http://dx.doi.org/10.1145/3092703.3092728

[73] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin T. Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 745–759. DOI:http://dx.doi.org/10.1145/3314221.3314640

[74] Apache Commons. 2022. Commons Collections. https://commons.apache.org/proper/commons-collections/. (2022). [Online; accessed 7-Sept-2022].

[75] Trove. 2022. High Speed Object and Primitive Collections for Java. http://trove.starlight-systems.com/. (2022). [Online; accessed 7-Sept-2022].

[76] Guava. 2022. Google Core Libraries for Java. https://github.com/google/guava. (2022). [Online; accessed 7-Sept-2022].

[77] 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 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 59–76. DOI:http://dx.doi.org/10.1145/1094811.1094817

[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)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:26. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.22

[79] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. DOI:http://dx.doi.org/10.1145/514188.514190

[80] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. 2010. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.* 32, 2 (2010), 5:1–5:52. DOI:http://dx.doi.org/10.1145/1667048.1667050

[81] Jiangchao Liu, Liqian Chen, and Xavier Rival. 2018. Automatic Verification of Embedded System Code Manipulating Dynamic Structures Stored in Contiguous Regions. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37, 11 (2018), 2311–2322. DOI:http://dx.doi.org/10.1109/TCAD.2018.2858462

[82] Thomas W. Reps. 1997. Program Analysis via Graph Reachability. In *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, Jan Maluszynski (Ed.). MIT Press, 5–19.

[83] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170. DOI:http://dx.doi.org/10.1016/0304-3975(96)00072-2

[84] Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, Eric Bodden, Laurie J. Hendren, Patrick Lam, and Elena Sherman (Eds.). ACM, 3–8. DOI:http://dx.doi.org/10.1145/2259051.2259052

[85] Thomas W. Reps. 1995. Shape Analysis as a Generalized Path Problem. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995.* 1–11. DOI:http://dx.doi.org/10.1145/215465.215466

[86] Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. 2007. Shape Analysis and Applications. In *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, Y. N. Srikant and Priti Shankar (Eds.). CRC Press, 12.

[87] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. DOI:http://dx.doi.org/10.1145/2049697.2049700

[88] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2010. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 211–222. DOI:http://dx.doi.org/10.1145/1706299.1706326

[89] Kuat Yessenov, Ruzica Piskac, and Viktor Kuncak. 2010. Collections, Cardinalities, and Relations. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Gilles Barthe and Manuel V. Hermenegildo (Eds.), Vol. 5944. Springer, 380–395. DOI:http://dx.doi.org/10.1007/978-3-642-11319-2_27

[90] Tianhan Lu, Pavol Cerný, Bor-Yuh Evan Chang, and Ashutosh Trivedi. 2019. Type-Directed Bounding of Collections in Reactive Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science)*, Constantin Enea and Ruzica Piskac (Eds.), Vol. 11388. Springer, 275–296. DOI:http://dx.doi.org/10.1007/978-3-030-11245-5_13

[91] Tuan-Hung Pham, Minh-Thai Trinh, Anh-Hoang Truong, and Wei-Ngan Chin. 2011. FixBag: A Fixpoint Calculator for Quantified Bag Constraints. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 656–662. DOI:http://dx.doi.org/10.1007/978-3-642-22110-1_53

[92] Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. 2013. QUIC Graphs: Relational Invariant Generation for Containers. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 7920. Springer, 401–425. DOI:http://dx.doi.org/10.1007/978-3-642-39038-8_17

[93] Zhoulai Fu. 2014. Targeted Update - Aggressive Memory Abstraction Beyond Common Sense and Its Application on Static Numeric Analysis. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 534–553. DOI:http://dx.doi.org/10.1007/978-3-642-54833-8_28

[94] Zhoulai Fu. 2014. Modularly Combining Numeric Abstract Domains with Points-to Analysis, and a Scalable Static Numeric Analyzer for Java. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science)*, Kenneth L. McMillan and Xavier Rival (Eds.), Vol. 8318. Springer, 282–301. DOI:http://dx.doi.org/10.1007/978-3-642-54013-4_16

[95] Karen Zee, Viktor Kuncak, and Martin C. Rinard. 2008. Full functional verification of linked data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 349–361. DOI:http://dx.doi.org/10.1145/1375581.1375624

[96] Deokhwan Kim and Martin C. Rinard. 2011. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 528–541. DOI:http://dx.doi.org/10.1145/1993498.1993561

[97] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer*

Science), Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 124–139. DOI: http://dx.doi.org/10.1007/978-3-642-54862-8_9

[98] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 678–692. DOI: http://dx.doi.org/10.1145/3192366.3192383

[99] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T Barr. 2018. Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 118–128.

[100] Jianhui Chen and Fei He. 2018. Control flow-guided SMT solving for program verification. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 351–361. DOI: http://dx.doi.org/10.1145/3238147.3238218

[101] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast bit-vector satisfiability. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 38–50. DOI: http://dx.doi.org/10.1145/3395363.3397378

[102] Hongyu Fan, Weiting Liu, and Fei He. 2022. Interference relation-guided SMT solving for multi-threaded program verification. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 163–176. DOI: http://dx.doi.org/10.1145/3503221.3508424

[103] Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. 2021. Type and interval aware array constraint solving for symbolic execution. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 361–373. DOI: http://dx.doi.org/10.1145/3460319.3464826

[104] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. 2021. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 651–664. DOI: http://dx.doi.org/10.1145/3453483.3454068