HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PHD QUALIFICATION EXAMINATION (PQE)

A Survey on Heap Analysis

Student:
Chengpeng WANG

Supervisor:
Dr. Charles Zhang

October 4, 2020

Abstract

Program heap is essentially a mathematical concept, i.e., a set of objects and a connectivity relation on them. In real-world programs, developers define various types of data structures and allocate the objects in the heap. Because of dynamic allocation and flexible heap manipulations, the size of heap-allocated objects is potentially unbounded and the connectivity relation can be extremely complex, which increase the difficulty of assuring memory safety and understanding heap-manipulating programs.

Static program analysis, a program analysis technique, achieves satisfactory performance in many program analysis tasks. Based on abstract state transformation, program states in actual execution can be approximated without executing programs. However, the unboundedness of heap and complex connectivity relation make heap-allocated objects difficult to be abstracted precisely, thus the precision of clients in static analysis is degraded in the presence of intensive use of heap. In order to support static analysis clients to analyze program precisely, heap analysis provides linkage properties about the heap, which reflects the connectivity relations of objects, such as reachability, ownership, etc.

According to the way to organize heap-allocated objects, existing works can be divided into two categories. The first category of works concentrate on the structural heap connected by pointers, while the second category of works focus on the structural heap organized by containers. Aiming to a particular structural heap, abstract heap model is established and linkage properties are inferred specifically by checking satisfiability of constraints or solving a CFL-reachability problem.

To show the impact of heap analysis, applications of heap analysis are introduced. The applications mainly include memory corruption detection, typestate verification, memory safety verification in multi-threaded programs, and heap-manipulating program understanding. We hope our survey will shed light on our future work on heap analysis in certain scenarios, including analyzing structural heap manipulated in a loop.

Contents

1 Introduction

Software security has great consequences in daily lives. Vulnerabilities can occur in many real-world software systems, such as the vulnerabilities in monitoring systems and financial management systems, which can cause aircraft crash and inestimable economic loss. Among these vulnerabilities, there are a considerable number of memory bugs, such as buffer overflow, null pointer dereference, and data race in multi-threaded programs. In contrast to conventional testing approaches, static program analysis attempts to over-approximate possible program states by not executing programs, and it obtains satisfactory performance in the analysis of memory issues [1].

Unfortunately, dynamic allocation and manipulation pose challenges in analyzing heap-manipulating programs [2, 3], in which heap-allocated objects are frequently used. Firstly, dynamic allocation facilitates creating data structures by statements in a program instead of variable declaration, therefore the size of the heap is not statically determinable. Secondly, developers can self-define data structures in real-world programs, which makes it difficult to analyze heap-allocated objects generally. Thirdly, heap-allocated objects can be manipulated by statements including pointer operations and library interfaces, and the forms of the statements are various, which yield points-to relation inevitably sophisticated. Therefore, the precision of static analysis can be degraded if precise and expressive heap properties are unavailable.

Heap analysis, at a generic level, provides heap properties to support precise analysis in static analysis clients. Due to the diversity of heap structure, it is impossible to apply a general theory to perform the analysis on an arbitrary form of heap with the best precision. To the best of knowledge, existing works mainly focus on two types of the heap, both of which are organized and formed into a regular structure. The first structural heap is organized by pointers forming self-similar structures, such as linked lists and binary trees. The heap is accessed and manipulated by stack pointers, which traverse the heap through pointer-valued fields of data structure. The second structural heap is organized by containers, in which heap-allocated objects are stored in sequence or according to keys, and it is accessed and manipulated by library interfaces of containers. In such two structural heaps, it is feasible to obtain precise linkage properties including but not limited to points-to relation and aliasing. These linkage properties are important to make top clients more precise.

In this survey, we summarize the works related to these two types of the structural heap. Section 2 introduces the preliminaries and background of heap analysis. Section 3 and section 4 review the works related to structural heap with pointers and containers respectively, and the properties are illustrated with applications. Section 5 discusses the applications of heap analysis, and the conclusion is followed in Section 6.

2 Background and Preliminary

In this section, we give an introduction of heap-allocated objects firstly, including heap allocation and the formed structural heaps. Based on the definitions of the structural heaps, the inference of linkage properties in structural heaps is formulated, which is the aim of heap analysis. Points-to graphs and heap abstractions, are discussed to prepare the preliminaries to model heap in a bounded size.

2.1 Heap Allocation

Memory allocation of programs, such as C/C++ programs, includes two parts, namely stack allocation and heap allocation. Stack allocation happens on the function call stack, and the size of allocated memory is known to compiler before the program execution. Function parameters and local variables are stored on the stack, and the memory is deallocated when the function call is finished. Different from stack allocation, the heap-allocated object is managed by programmers rather than compilers. The memory is allocated during the execution of allocation statements written by programmers, such as *malloc* and *new* in C/C++ programs. Moreover, the memory should be deallocated manually, such as *free* and *delete* statement.

Example 1. Figure 1 depicts an example C program. In the *main* function, the first two elements of a linked list are swapped after the creation. The local variables, including c, e, and p are stored on the stack while the nodes in linked lists, such as e and $e \to n$ point to, are allocated in heap.

```
1  typedef struct
2    node {
3    struct node *n;
4    int data;
5  } List;
6
7  main() {
8    List *c;
9    c = create_list();
10   c = swap(c);
11 }
12
13 List* create_list() {
14   List *e, *c;
15   int i, size;
16   c = NULL;
17   scanf("%d", &size);
18   for (i = 0; i < size; i++) {
19     e = malloc(sizeof(List));
20     e->data = i;
21     e->n = c;
22     c = e;
23   }
24   return c;
25 }
26
27 List* swap(List *c) {
28   List *p;
29   if (c != NULL) {
30     p = c;
31     c = c->n;
32     p->n = c->n;
33     c->n = p;
34   }
35   return c;
36 }
```

Figure 1: An example of linked list creation and swapping in C

```
1  vector<File*> v;
2  for (int k = 0; k <= 10; k++) {
3   if (k % 2) {
4    v.push_back(NULL);
5   } else {
6    File* a = malloc();
7    a->close();
8    v.push_back(a);
9   }
10 }
```

Figure 2: An example of inserting File pointers into C++ STL container

It should be noticed that there are three distinguished features of heap allocation.

- **Dynamic allocation:** Allocation statements might be executed multiple times in the loops and recursive functions, thus the size of allocated objects depends on the number of iterations and might be unbounded.
- **Various types of data structure:** Types of heap-allocated objects are various and the linkage between objects are also flexible. Programmers can define specific data structures by themselves and connect the objects of data structures by pointers in the way as they like.
- **Dynamic manipulation:** The ways to allocate heap objects and their linkage are both affected by the different forms of statements, such as pointer dereference and library interfaces of containers.

For example, the dereference of e at line 20 in Figure 1 and the *push_back* of vector v at line 6 and 8 in Figure 2. These three features distinguish heap-allocated objects from stack variables and make the analysis more challenging.

2.2 Linkage Properties in Structural Heap

In this survey, we restrict heap analysis into a subfield of static program analysis. Static program analysis aims to reason about the behavior of computer programs without actually running them [1]. It approximates the actual program behaviors by a domain, including symbolic domain in symbolic execution and abstract domain in abstract interpretation. Actual program behaviors can be approximated by computing the effect of each statement in a specific domain.

Due to the complexity of heap, it seems impossible to establish a general heap model for arbitrary forms of programs. According to the works we surveyed, we obtained an interesting observation as follows.

Observation 1. Heap-allocated objects can form two structural heaps $SH_P = (E, N_P, P, F_P)$ and $SH_C = (C, N_C, I, F_C)$, in which objects are organized by stack pointers and containers respectively.

More specifically, in Figure 1, the local variable in *create_list*, i.e., the stack pointers c and e access the objects allocated in the loop and link them into a linked list. Similarly, the heap-allocated objects are organized by enumerating the pointers pointing to them in a container. This observation sheds light on the inference of linkage properties of a collection of heap-allocated objects. We can give formal definitions of two structural heaps as follows.

Definition 1 (Structural heap with pointers)**.** The structural heaps with pointers, denoted by SH_P, can be formally defined by a 5-tuple $(P_S, N, E, P_F, R_{P_F})$.

- P_S: A set of stack pointers, which are the entry of structural heap.
- N: A set of heap-allocated objects of the same data type.
- $E \subseteq P_S \times N$: A relation between stack pointers and heap-allocated objects, which indicates the entry of the structural heap in programs.
- $P_F \subseteq N \times N$: A relation between two heap-allocated objects. It is induced by the pointer-valued field of heap-allocated objects, which links objects as a connected component.
- R_{P_F}: A restriction on P_F such that SH_P is self-similar, i.e., for each $(n_1, n_2) \in P_F$, the heap-allocated objects reachable from n_2 form a structural heap, which can be represented by a $(P'_S \cup \{p_{tmp}\}, N', E' \cup (p_{tmp}, n_2), P'_F, R'_{P_F})$, where p_{tmp} is an artifact stack pointers and P'_S, N', P'_F and R'_{P_F} are the subsets of corresponding components in SH_P.

Definition 2 (Structural heaps in containers)**.** The structural heaps organized by containers, denoted by SH_C, can be formally defined by a 4-tuple (C, I, N, F).

- $C = C_{pos} \cup C_{key}$: A set of containers in two categories, i.e., position-dependent containers and key-dependent containers.
- I: A set of index sets of the containers in C. An index is comparable value, such as integers and strings.
- N: A set of heap-allocated objects or the pointers which reach heap-allocated objects.
- F: A set of mappings from an index set I to $N' \subset N$, where the elements of N' have the same type.

For either of structural heaps, we can get the linkage properties of heap-allocated objects. According to our survey, the linkage properties for the first structural heap mainly include:

- Reachability of heap-allocated objects, i.e., given a heap-allocated object, whether it is accessed by moving stack pointer along an access path [4]. It can be formulated by checking the satisfiability of the formula for a first-order logic with transitive closure, i.e., for a given $p \in P_S$, $\exists n' \in N, E(p,n') \wedge P_F^+(n',n)$, where $P_F^+(n',n)$ is defined as follows

$$P_F^+(n_1,n_2) = (n_1 = n_2) \vee (\exists n_3, P_F^+(n_1,n3) \wedge P_F(n_3,n_2))$$

- Disjointness of two collections of heap-allocated objects [5, 6]. It can be formulated by checking whether $N_1 \cap N_2$ is an empty set where $N_1, N_2 \in N$. In many scenarios, N_1 and N_2 are the sets of heap-allocated objects reachable from two stack pointers.

For the other case, the linkage properties for the second structural heap can be categorized into these two types.

- Ownership of heap-allocated objects, i.e., whether a heap-allocated object belongs to a container [7]. More specifically, an object belongs to a container if and only if the object is stored in the container explicitly or the pointer which can reach to the heap-allocated object. It can be formulated by checking whether $\exists index \in I_c, f(index) = n$ holds for a given $n \in N, f \in F$ and $I_c \in I$.

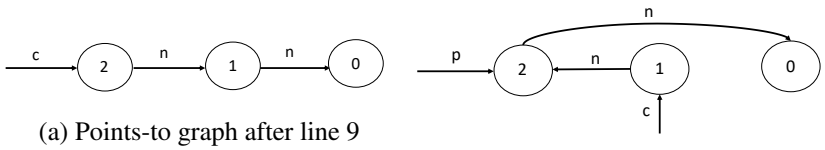
- Index-value correlations in containers, i.e., which objects are stored in a specific position or paired with a specific key [8]. It can be formulated by computing the value of $f(index)$ for a given $f \in F$, $I_c \in I$ and $index \in I_c$ and checking whether the result is equal to a given $n \in N$.

2.3 Points-to Graph

In order to describe the memory state, points-to graphs are proposed in heap analysis to describe the linkage information. The statements which operate on the heap can be modeled as the transformers on the points-to graphs. The points-to graphs we surveyed can be mainly categorized into two types, which are utilized to model the structural heaps discussed in the section .

Points-to graph in SH_P is defined as $(N, E \cup P_F)$. The nodes in the points-to graph correspond to the elements in N, which are the heap-allocated objects. The edges connects two heap-allocated objects if a pointer variable in one of them points to the other object. This type of edges correspond to the occurrence of the pairs in E. Meanwhile, there are the edges indicating that stack pointers point to the heap-allocated objects, which correspond to the occurrence of the pairs in P_F. All the edges are labeled with a symbol, i.e., the name of a stack pointer or a pointer-valued field, respectively.

In the example shown Figure 1, the memory state is depicted in the points-to graph shown in Figure 3. The leftmost edge of which the label is c indicates that the stack



(a) Points-to graph after line 9

(b) Points-to graph after line 31

Figure 3: Points-to graphs of the program in Figure 1 with the input *size* = 3

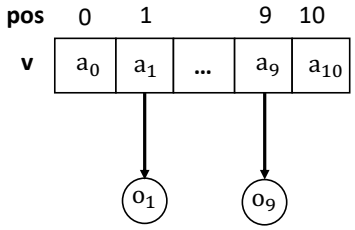


Figure 4: Points-to graph of the program in Figure 2

pointer *c* points to the object of which the *data* is equal to 2. The edges with *n* as its label connects two heap-allocated objects and indicate the linkage information through pointer-valued field *n*. It should be noticed that the content stored in the heap-allocated object can not be modeled in points-to graphs, such as the numeric data in this example. Here we denote the value in the circle for illustration of functionality only.

Points-to graph in SH_C depicts the correlation between the element in the key of containers and the heap objects it contains. For example, Figure 4 shows the points-to graph after the loop in Figure 2. The points-to graph displays the correlation of the key and the heap objects, in which the key is the position of an element.

2.4 Heap Abstraction

As discussed in Section 2.1, the allocation of heap-allocated objects is affected by the statements and might form into an unbounded size. Therefore it is not capable to apply points-to graphs to the analysis of the program directly. In order to assure the soundness, heap abstraction is a necessary and key part of heap analysis. In an abstract heap, several heap-allocated objects are regarded as a single abstract objects, which bounds the size of heap-allocated objects. Given a heap abstraction, a points-to graph discussed in Section 2.3 can be converted into an abstract points-to graph by merging multiple nodes as a summary node. Heap analysis can be regarded as linkage property inference based on the effect of statements in an abstract points-to graph.

Compared to a points-to graph without abstraction, an abstract points-to graph must lost the information to some degrees. For example, the points-to edges among merged heap-allocated objects do not occur in the abstract points-to graph. Heap abstractions

differ in different applications to trade-off between precision and efficiency. Heap abstractions can be based on

- Allocation site: summarizing the heap-allocated objects created by the same statement [9, 7]. Allocation-site based abstraction is commonly used in the analysis of Java programs.
- Generic instrumentation predicate: summarizing the heap-allocated objects based on parametric predicates in the configuration of analyzers [4, 5]. If a collection of heap-allocated objects satisfy the relation described by the predicates, they are summarized in the abstract heap.
- Type: summarizing the heap-allocated objects with the same type. It can be combined with allocation-site based abstraction to obtain a more fine-grained abstract heap [10, 11].

Heap abstraction is an orthogonal dimension to structural heap in heap analysis. For a certain form of structural heap, different heap abstractions can be applied, which is dependent to the kinds of linkage properties and the requirement of precision and efficiency. In Section 3 and Section 4, we will discuss technical details of abstract heap based on generic instrumentation predicates and allocation sites, respectively.

3 Structural Heap with Pointers

This section describes linkage properties inference of structural heap with pointers and its applications. There has been a long history of reasoning about unbounded data structures connected by pointers. Shape analysis is an evolutionary approach to infer the linkage properties of structural heap [12]. According to Definition 1 in Section 2, this type of the structural heap are a self-similar collection of heap-allocated objects, which are linked by the pointer-valued pointers and take stack pointers as the entry of heap memory. Typical instances include linked lists and binary trees, of which definitions are shown in Figure 5. In this section, we discuss how to obtain the linkage properties of the structural heap, such as reachability and disjointness defined in Section 2. Other instances, such red-black trees and sorted linked lists, have the restriction of data stored in each node, which are in the scope of linkage properties we discuss in the survey.

In Section 3.1, the syntax of heap manipulation and the basic concept of abstract interpretation are introduced. Section 3.2 and 3.3 discuss two typical shape analyzers, including TVLA and Xisa, both of which are based on abstract interpretation. The applications of properties are discussed in Section 3.4 following the comparison of TVLA and XISA.

```
struct List {
    List *next;
    int data;
};
```

```
struct Tree {
    Tree *left;
    Tree *right;
    int data;
};
```

Figure 5: An example of definitions linked lists and binary trees in C/C++

$$
\begin{aligned}
\textit{Address} \quad & A := \textit{NULL} \mid \textit{malloc}() \\
\textit{StackPtrExpr} \quad & P_s := p \\
\textit{HeapPtrExpr} \quad & P_h := P_s \rightarrow n \mid P_h \rightarrow n \\
\textit{PtrExpr} \quad & P := P_s \mid P_h \\
\textit{Statement} \quad & S := P_s = A \mid P_s = P \mid P_h = A \mid P_h = P
\end{aligned}
$$

Figure 6: Syntax of operations on structural heaps

3.1 Preliminary

3.1.1 Syntax of Heap Manipulation

In a structural heap with pointers, the linkage of heap-allocated objects is manipulated by pointer statements. The statements are defined by the following grammar in the Figure 6 as follows. For simplicity, we assume there is no pointer arithmetic operation and heap manipulation is performed through access paths from stack pointers, i.e., accessing heap-allocated objects by the pointer-valued fields recursively. In the grammar, v is a variable and p is a stack pointer. For short, we denote access paths through pointer-valued fields by heap pointers. Specifically, address includes NULL and the heap memory location created by *malloc()*. Stack pointers and heap pointers correspond to two relations in Definition 1, i.e., $E \subseteq P_S \times N$ and $P_F \subseteq N \times N$. $P_s = A$ and $P_s = P$ update the relation E and $P_h = A$ and $P_h = P$ update the relation P_F. The statements are organized in the program structures to perform a specific functionality in the structural heap, e.g., creating and swapping a linked list shown in Figure 1.

3.1.2 Abstract Interpretation

Abstract interpretation is a theory of sound approximation of semantics of computer programs, based on monotonic functions over ordered sets, especially lattices [1]. It is the partial execution of programs and applied to the automatic extraction of information about the possible execution of computer programs. The concrete domain and abstract domain are the two key components in abstract interpretation, which are order sets containing concrete states and abstract states in the program execution respectively. Transformers in two domains map a concrete or abstract state to another one.

More specifically, let L_1 and L_2 be a concrete domain and abstract domain. Each

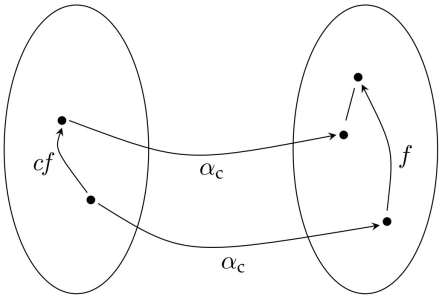


Figure 7: Soundness of abstract interpretation

program statement corresponds to a concrete transformer cf, which is the computation in the actual execution. The concrete transformer is a mapping from L_1 and L_1 and reflects the effect of program statement in the concrete domain. In order to over-approximate the program states without actual execution, i.e., cover possible program behavior, a mapping from L_1 to L_2, denoted by α_c, is defined to abstract concrete states by an abstract state. Meanwhile, an abstract transformer f transforms an abstract state to another, which models the effect of a statement in the abstract domain. These two transformers are both monotonic, i.e., for $\forall x_c, y_c \in L_1$ and $\forall x, y \in L_2$, if $x_c \sqsubseteq y_c$ and $x \sqsubseteq y$, then

$$cf(x_c) \sqsubseteq cf(y_c), f(x) \sqsubseteq f(y)$$

For a specific static analysis problem, the definitions of α_c and f should satisfy the following soundness condition:

$$\alpha_c(cf(S_c)) \sqsubseteq f(\alpha_c(S_c))$$

The soundness condition can be depicted illustratively shown in Figure 7. The abstraction of the concrete state after transforming S_c is approximated by the abstract state after transforming the abstract state $\alpha_c(S_c)$.

A fixed point is obtained by executing all the statements, in a iterative way if there is a loop or a recursive function. The soundness condition assures that the fixed point covers all the possible program behaviors. Due to the transformers are the monotonic functions over the lattices, the finite height of the L_2 guarantees the termination of the computation in the abstract domain. For the abstract domain with an infinite height, the termination can not be assured without defining a widening operator for acceleration. The widening operator will be discussed briefly in Section 3.3.

Abstract interpretation is a fundamental theory of many static heap analyses. The abstract domain is defined to establish an abstract heap model, which tackles unboundedness of heap-allocated objects. Meanwhile, an abstract transformer of each statement in Figure 6 needs to be defined along with the abstract heap model in order to model the effect of dynamic manipulation. Section 3.2 and 3.3 will discuss two abstract heap models and the abstract transformers in two typical works.

3.2 TVLA: TVL based Shape Analysis

Three Valued Logic Analyzer(TVLA) is an evolving research vehicle for abstract interpretation and naturally suited for checking properties of the structural heap. It is the collection of a series of works[4, 14, 15]. TVLA encodes the points-to graph by first-order logic and expresses the properties by first-order logic with transitive closure(FO^{TC}). In order to guarantee the bounded size of points-to graph to establish abstract heap model, the truth values are extended by introducing an indeterminate truth value denoted by 1/2, which the name of TVLA comes from. The logic operators in TVL are defined as follows.

a	$\neg a$
1	0
0	1
1/2	1/2

$a \vee b$		a		
		0	1/2	1
b	0	0	1/2	1
	1/2	1/2	1/2	1
	1	1	1	1

$a \wedge b$		a		
		0	1/2	1
b	0	0	0	0
	1/2	0	1/2	1/2
	1	0	1/2	1

In this section, we will discuss the details of the TVL based memory model and the transition of memory state.

3.2.1 Memory Model based on TVL

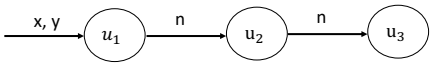
In TVLA, the points-to graph is encoded by **core predicates**, which are shown in Figure 1. The predicate $q(v)$ encodes the points-to edge from stack pointer q to a heap-allocated objects. The true evaluation of $q(v)$ corresponds to an element in P_F. The predicate $n(v_1, v_2)$ encodes the points-to edge of two heap-allocated objects connected by pointer-valued field n. The true evaluation of $n(v_1, v_2)$ corresponds to an element in E. It is obvious that it is precise to encode the heap memory manipulated by the language in the syntax defined in Figure 6 as long core predicates for all stack pointers and pointer-valued fields are defined.

Table 1: Core predicates in TVL

Predicate	Intended Meaning
$q(v)$	Does pointer variable q point to heap-allocated object v?
$n(v_1, v_2)$	Does the n filed of v_1 point to v_2?
$sm(v)$	Does element v represent more than one concrete elements?

Example 2. Figure 8(b) shows the truth values of the predicates $x(v)$ and $y(v)$, which store the point-to information of the stack pointers x and y. More specifically, $x(v)$ and $y(v)$ both evaluate to 1 when v is interpreted as u, indicating the fact that the stack pointer variables x and y both point to u_1.

Figure 8(c) shows the truth values of the predicates $n(v_1, v_2)$, and it records the linkage relation of the heap individuals v_1 and v_2 through the field n of v_1. Only $n(u_1, u_2)$ and $n(u_2, u_3)$ evaluate to 1, which reflects the linear structure of the linked list.



(a) points-to graph

	x	y
u_1	1	1
u_2	0	0
u_3	0	0

(b) $q(v)$

n	u_1	u_2	u_3
u_1	0	1	0
u_2	0	0	1
u_3	0	0	0

(c) $n(v_1, v_2)$

Figure 8: An example of points-to graph and core predicates

Based on the points-to graph, the linkage properties can be expressed by FO^{TC} formulae, which are **instrumentation predicates**. Figure 2 shows the common used instrumentation predicates.

Table 2: Instrumentation predicates in FO^{TC}

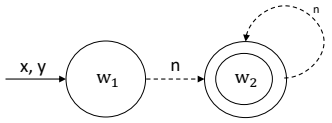
Predicate	Intended Meaning
$is(v) = \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$	Sharing: Is v pointed by two different objects v_1 and v_2?
$r_x(v) = x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)$	Reachability: Is v reachable from stack pointer x?
$c(v) = \exists u : n(v, u) \wedge n^+(u, v)$	Cyclicity: Is v on a directed cycle?

To summarize the unbounded structural heap, a finite number of unary predicates, named **abstraction predicates**, are defined to partition heap-allocated objects into disjoint sets. For example, unary instrumentation predicates in Table 2, can be chosen as abstraction predicates. The heap-allocated objects are summarized into a summarized node in an abstract heap by abstraction predicates if all the abstraction predicates evaluate to the same truth value over these objects. The finite number of abstraction predicates guarantees the bounded size of abstract heap.

More specifically, given a set of unary predicates $A = \{p_1, p_2, ..., p_n\}$, where $p_i (1 \leq i \leq n)$ is named as an abstraction predicate, a points-to graph is folded into a more compact one with summary nodes, named an abstract points-to graph. Specifically, if v_1 and v_2 are two heap-allocated objects such that $\forall p_i \in A : p_i(v_1) = p_i(v_2)$, v_1 and v_2 can be summarized. The predicates related to the summary node evaluate to 1/2 to over-approximate concrete linkage relation. It can be proven that the set of all abstract points-graphs is of a bounded size as long as the number of the abstraction predicates is bounded, which guarantees the finite height of abstract domain.

Example 3. If A is set as $\{x, y, r_x, r_y\}$, the abstract points-to graph in Figure 9(a) can be obtained as the abstract graph of Figure 9(a). It is obvious that $x(u_2) = x(u_3)$ and $y(u_2) = y(u_3)$, so these two objects are summarized into the summary node v_2.

After merging u_2 and u_3 into a summary node w_2, $n(w_1, w_2)$ evaluate to 1/2 to show that w_1 might point to w_2. It over-approximates the concrete information that u_1 points to u_2 but does not point to u_3 in concrete points-to graph.



(a) abstract points-to graph

	x	y
w_1	1	1
w_2	0	0

(b) $q(v)$

n	w_1	w_2
w_1	0	1/2
w_2	0	1/2

(c) $n(v_1, v_2)$

Figure 9: An example of abstract points-to graph in TVLA

It should to be notice that the granularity of abstraction depends on the set of abstraction predicates A. If more abstraction predicates are added in A, fewer objects are summarized and abstract points-to graph is less compact. Therefore, less linkage relation is lost in the abstraction, which makes the abstract domain stores more precise linkage relation.

Based on the abstract points-to graph, the properties can be checked by evaluating the FO^{TC} formulae. If the formula evaluate to 1(1/2), the property must(may) hold. Otherwise, the property must not hold in the structural heap.

Example 4 (Reachability, Cyclicity)**.** In Figure 9(a), w_2 is a summary node and has a dotted self-cycle. Based on the definition of $c(v)$, it is easy to get the truth value of $c(w_2)$ to 1/2, which matches the fact shown by dotted self-cycle of w_2. Similarly, based on the definition of $r_x(v)$, it is easy to compute $r_x(w_1)$ as 1 and $r_x(w_2)$ as 1/2, which indicate the fact that x must reach the heap-allocated object abstracted by w_1 and may reach the heap-allocated object abstracted by w_2 respectively.

Example 5 (Aliasing)**.** Based on core predicates in Table 1, other heap properties can be self-defined. Two pointers, i.e., p and q, are aliased if they point to the same heap cell. Aliasing can be encoded by $\exists n : p(n) \wedge q(n)$.

3.2.2 Transformer in TVL

As discussed in Section 3.2.1 , the truth values of the core predicates determine the abstract memory state uniquely. Therefore, the transformers of states in abstract heap model can be defined by the predicate-update formulae. As long as the values of core predicates after the statement are computed, the new abstract points-to graph are obtained.

Consider the statement $x = x \to n$ for an example, the predicate-update formula of the core predicate $x(u)$ is $x'(u) = \exists v : x(v) \wedge n(v,u)$. Other core predicates are not updated, because no edge except the points-to edge starting from x is changed in the heap state transformation.

One difficulty is that in the cases where the some edges are uncertain. More specifically, the predicate-update formula might evaluate to to 1/2 over some objects, which means that the corresponding core predicate has the indefinite value, which blurs the truth of point-to relation in points-to graph. To obtain higher accuracy, Mooly Sagiv and Thomas Reps proposed two additional operations and combine them into a transformer, namely *focus* and *coerce*[4].

Focus operation assures the predicate-update formula evaluates to a definite value, and is guided by a set of focus formulae related to the statement. For example, if we consider the statement $y \to n = x \to n$, the set of focus operation is $\{\exists v : x(v) \wedge n(v,u), \exists v : y(v) \wedge n(v,u)\}$. If either has the value of 1/2, the node will be split, which yields logic structures representing disjoint abstract states, and then performs a precise update, i.e., *strong update*, upon them based on the predicate-update formula respectively.

Coerce operation removes the impossible logic structures based on the patterns of inconsistencies. For example, the summary nodes can not make the unary predicates in the form of r_x evaluate to the definite values. Additionally, the correlations between the abstract predicates provide several useful patterns of inconsistencies. If an inconsistency occurs, the logic structures should be removed, or the summary node should be reduced to a concrete heap-allocated object and the summary edge should be deleted. The details of *focus* and *coerce* are discussed in section 6.3 and 6.4 in [4] and and more details are not discussed in the survey.

Example 6. Figure 10 shows an example of state transition at the statement $x = x \to n$. The abstract predicates are $\{x, y, r_x, r_y\}$. The predicate-update formula of $x = x \to n$ is $x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$, and the set of focus formulae is $\{\exists v_1 : x(v_1) \wedge n(v_1, v)\}$. S_0 is split into three logic structures $S_{1,0}$, $S_{1,1a}$ and $S_{1\,1b}$, and the union of the concrete memory states they contain are equal to those which S_0 represents. Based on the predicate-update formula, strong updates can be performed respectively. In the phase of *coerce* operation, $S_{2,0}$ is removed, for the value of $r_x(u_2)$ is 1 while u_2 in not in the transitive closure of x in the graph. The uncertainty in $S_{2,1a}$ and $S_{2,1b}$ is removed based on the basic patterns discussed above in a similar way.

For each statement in the program, a transformer is constructed in the above way,

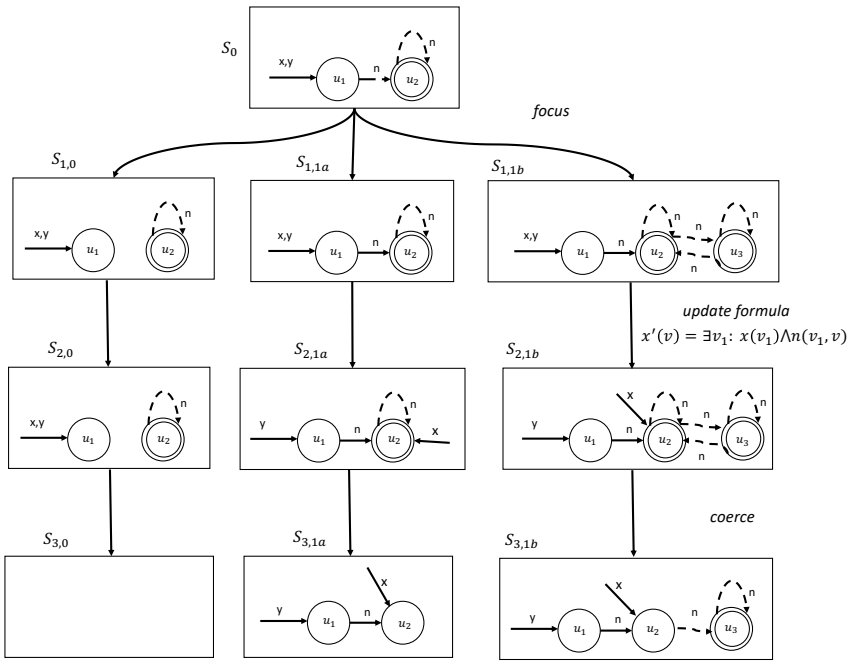


Figure 10: An example of state transition

and constraint solvers are invoked to update the truth values of predicates in each transformation. Due to the bounded size of abstract domain, a fixed point can be obtained by applying the transformers. The fixed point covers all the possible points-to graphs and over-approximates the program behaviors. The properties of structural heaps can be obtained by checking the values of instrumentation predicates, which has been shown in Figure 4.

3.2.3 Overhead of Predicate Abstraction

TVLA is a general framework suited for checking properties of heap-allocated objects based on predicate abstraction. The users encode the properties by FO^{TC} formulae and evaluate them in the abstract points-to graph after the fixed-point reaches, which makes TVLA extensive.

The time cost in state transformation makes TVLA inefficient. Some of the instrumentation predicates are used as abstraction predicates, such as r_x and c_n, and they should be updated in each transformation. A simple and straightforward way is to evaluate the instrumentation predicates based on their definitions and the latest values of core predicates. However, it is time consuming due to the time cost brought by the constraint solving in the process. In the work [16], a finite differencing-based approach

$< assert >$	::=emp	empty heap
	$\mid < exp > \rightarrow < exp >$	singleton heap
	$\mid < assert > * < assert >$	separating conjunction
	$\mid < assert > -\!\!* < assert >$	separating implication

Figure 11: Assertions in separation logic

is proposed to reduce time overhead.

The heap abstraction in TVLA depends on the abstraction predicates, thus the selection of the abstract predicates affect the precision of TVLA. Fewer abstraction predicates are used, more heap-allocated objects are likely to be merged which yields more loss of points-to edge, thus weaker properties are generated. However, excessive abstraction predicates make the space of points-to graphs explosively large, in which the nodes have $3^{|A|}$ possibilities. Even if the number is finite and termination of analysis is guaranteed, the heavy burden of time cost, caused by formula update in the state transformation, prevents it from analyzing large functions in real-world programs.

3.3 Xisa: SL based Shape Analysis

TVLA updates the abstract heap in a global way, i.e, when a transformer updates the points-to edge of a node, it requires the updating of the values of instrumentation predicates, which depends on the whole heap. A single cell might effect the values of many instrumentation predicates. However, a specific transformer only alters the linkage relation near the objects it manipulates. According to this observation, another leading shape analysis is proposed to perform a local reasoning about the heap based on Separation Logic(SL).

The assertions expressed in separation logic are listed in Figure 11. emp represents empty heap, in which no object is allocated. $e \rightarrow e'$ asserts that the heap contains one cell, at address e with contents e'. $p_1 * p_2$ asserts that the heap can be split into two disjoint parts in which p_1 and p_2 hold respectively. $p_1 -\!\!* p_2$ asserts the fact that p_2 will hold in the heap extended from the current heap and a disjoint part in which p_1 holds. Separating conjunction and implication make the separation logic naturally suitable for the local reasoning of heap [17, 18].

There are several typical works on SL based on shape analysis [6, 11, 19], including Xisa, Sling and Infer. This section give an introduction of Xisa to illustrate how to analyze the structural heap from a collection of objects in isolation [6, 20, 21, 22].

3.3.1 Memory Model based on SL

In order to describe the heap much in an isolated manner, **inductive predicates** over separation logic are defined to describe the linkage relation in the collection of heap-

allocated objects. For example, the inductive predicate $list(\alpha)$ is defined to describe the structural heap storing a linked list as follows:

$$list(\alpha) := (\alpha = null) \vee ((\alpha \neq null) \wedge (\exists \beta : \alpha \rightarrow \beta \wedge list(\beta)))$$

Similarly, for the structural heap storing a binary tree, the inductive predicate $tree(\alpha)$ can be defined as follows:

$$tree(\alpha) := (\alpha = null) \vee ((\alpha \neq null) \wedge (\exists \beta\ \gamma : \alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma \wedge tree(\beta) \wedge tree(\gamma)))$$

α, β and γ are the symbols in N and P_S in Definition 1. It is obvious that the points-to edges in a component of a separating conjunction are abstracted in the inductive predicates. By combining the inductive predicates to form a separating conjunction, the linkage properties of local heap yield a global property of large heap.

In the work [6], Chang and Rival observe that the checking functions in the program can guide heap abstractions. The checking function is originally designed for testing or dynamic analysis and reflects the intention of the developers, and it is the actual properties which the developers suppose the program should hold at some points.

For example, in the program shown in Figure 12, all the nodes in the list l are blue nodes at the entry of *make_all_purple* and the function colors all the nodes purple. The checking function *bluelist* and *purplelist* in the assert statement check whether the nodes accessed by their parameters are in blue or purple respectively. This property reflects the intention of the developers that the nodes of the linked list l are colored in purple if they are all in blue initially. It provides an inspiration to establish the abstract heap model by splitting the heap based on the definition of two assert statements.

The inductive predicates corresponding to the checking functions are as follows:

$$bluenode(\alpha) = \alpha \text{ is blue}$$

$$bluelist(\alpha) := (\alpha = null) \vee ((\alpha \neq null) \wedge (\exists \beta : \alpha \rightarrow \beta \wedge bluenode(\alpha) \wedge bluelist(\beta)))$$

where α and β are the symbols of heap memory locations. The predicate $purplelist(\alpha)$ can be defined similarly. Figure 12 shows an example of heap abstraction based on the checking function *bluelist*. The bold edge in Figure 12(b) shows a collection of heap-allocated objects which satisfy the inductive predicate $bluenode(\alpha)$.

To express the state of entire heap, several inductive predicates are combined together in a separating conjunction A_i, then the abstract memory can be easily established by a finite disjunctions as follows:

$$< A_1; P_1 > \vee \ldots \vee < A_n; P_n >$$

Each disjunction represent a specific case of structure P_i is a pure formula of pointers.

Example 7. Consider the list l which assures that the checking result of $bluelist(l)$ is $true$. The memory region which stores the list l is shown in Figure 11(a). It can be described by the logic formula as follows:

$$bluelist(l) :=< emp; l = null > \vee < (l.next \rightarrow l') * bluelist(l'); l' \neq null \wedge blue(l) >$$

It should be notice that inductive predicates used to form separating conjunctions can be defined according to other specifications [5, 19], including the definition of data structure. The process of defining inductive predicates requires manual efforts in most scenarios, and there are several works on the automatic generation of heap abstraction template [23].

```
1  typedef struct List {
2    Label color;
3    List* next;
4  } List;
5  bool bluelist(List* l) {
6    if (l == null) return true;
7    else return (l->color==blue)
8      && bluelist(l->next);
9  }
10 bool purplelist(List* l) {
11   if (l == null} return true;
12   else return (l->color==purple)
13     && purplelist(l->next);
14 }
15 void make_all_purple(List*l) {
16   List* cur = l;
17   List* last = null;
18   while (cur!=null) {
19     assert(bluelist(cur));
20     make_purple(cur);
21     last = cur;
22     cur = cur->next;
23   }
24   assert(purplelist(cur));
25 }
```

Figure 12: An example of checking function of linked lists in C

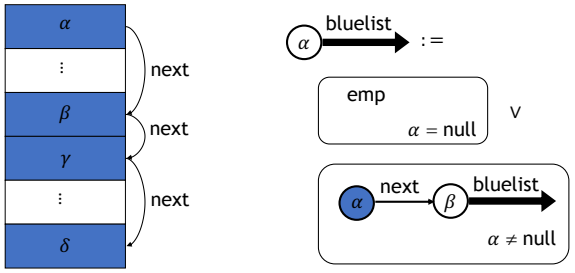


Figure 13: Memory abstraction based on checking functions in Figure 12

3.3.2 Transformer in SL

To reflect memory updates in the graph, Xisa simply modifies the appropriate points-to edges in the points-to graph. Based on the SL based abstract memory model, the statements defined in Figure 6 act on a component of a separating conjunction, i.e., Xisa supports the local reasoning. The soundness is guaranteed by the frame rule in SL [18].

When the points-to edge is in a component of a separating conjunction, the objects

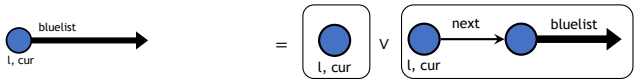


Figure 14: An example of unfolding

in the component abstracted by an inductive predicate are materialized by unfolding the definition of an inductive predicate so that the points-to edge can be exposed. Unfolding can be regarded as a reverse process of the memory abstraction, in which the component is replaced by the points-to edges and a smaller component abstracted by the same predicate [6].

Example 8. Figure 14 shows the example of unfolding the list *l*. Assume that the list *l* is not an empty list. We can notice that there is a deference of *cur* at line 22 in the function *make_all_purple*. According to the definition of the checker edge labeled *bluelist* as above, the unfolding can yield two disjunctives. After the unfolding, these two disjunctives can be updated by coloring the color of objects *cur* points to by purple.

Widening Different from TVLA, the termination of Xisa can not be assured directly. Although unfolding permits strong updates in the abstract domain, potentially infinite use of the unfolding operator increases to the points-to graph to an unbounded size. For example, Figure 15 shows the possible points-to graphs at the beginning of the first four iterations. As the iteration continues, more point-to edges labeled *next* are generated, which makes it is impossible to use a formula of separation logic with a finite number of existence quantifications. Moreover, the number of disjunctives can be unbounded, thus we can not find the finite number of formulae to describe the abstract heap state. These two problems make the analysis terminable in the presence of the loop and recursive functions.

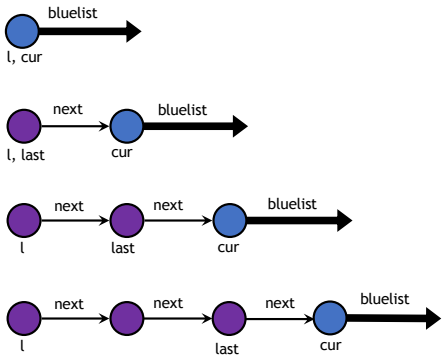


Figure 15: Points-to graphs at the loop entry in the first four iterations

To guarantee the termination, widening operator in the abstract interpretation should

be instantiated. The basic idea of widening in Xisa derives from the fact that the inductive predicate abstracts the points-to edges in the disjunctive generated by unfolding. This means that the disjunctives can be merged into a single one by summarizing the points-to edges by an inductive predicate. Figure 16 shows the widening operator over the abstract heap memory after line 20 in the first two iterations. The edge labeled *purplelist* contains the union of the concrete states represented by $l = last$ and $l \rightarrow next = last$. The detailed definition of widening is given in Section 4.2 in [6] and is not gone further discussion in the survey.

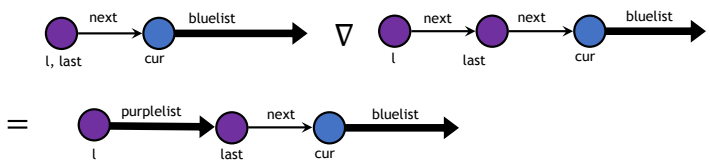


Figure 16: An example of widening

At last, the termination can be assured when there are a finite number of inductive predicates. The edges in the abstract points-to graph have finite possibilities, so there are a finite number of possible disjunctives, which assure the boundedness of abstract domain is bounded.

Example 9 (Reachability)**.** Figure 16 shows that the stack pointer l can reach the heap-allocated objects that *last* points to. Moreover, the inductive predicate *purplelist* indicates that the objects between them are both in purple.

3.3.3 Disjunctive Clumping

Disjunctive explosion Although the number of disjunctives is finite, it is memory-consuming if all the disjunctives are maintained. To reduce the space complexity, Huisong proposes the clumping of several similar disjunctives $d := m \vee ... \vee m$ into an abstract points-to graph in a coarse granularity [22], where $m := p{*}...{*}p$.

The disjunctives of d are sorted into abstract states $m_0, ..., m_n$, such that all the abstract states in m_i are similar. The similarity is measured based on the silhouette, which stores the structural order encoded in the regular expression. The similar abstract states which have similar silhouette are merged into one abstract state.

Example 10. Figure 17 shows an example of the clumping of the binary trees. The silhouette extracted from the abstract shape graph records the relative locations of the cursors x, y, and t. s_1, s_2 and s_3 all record the fact that the height of the node which t points to is not smaller than the height of the node x points to and the similar relation between x and y. The three silhouettes can be merged into the silhouette shown in (b), and m_1, m_2 and m_3 are clumped into the weakened abstract state in (c).

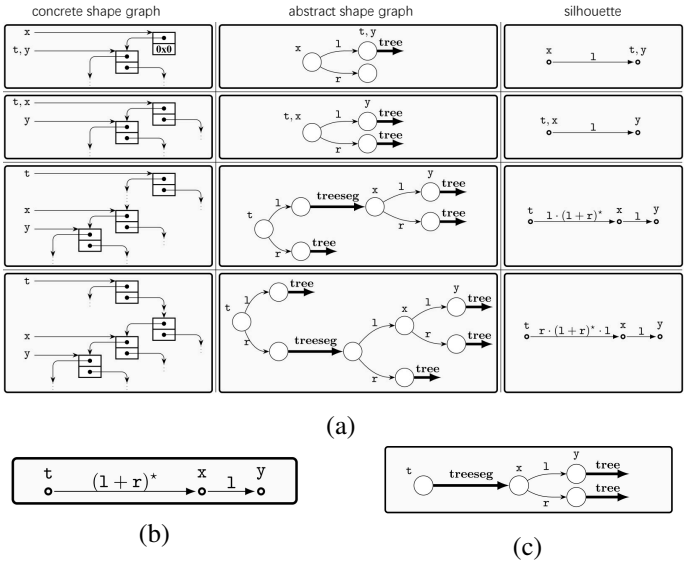


Figure 17: Semantic clumping of abstract states [22]

It should be noticed that the silhouette is another heap representation different from the logic based heap abstraction. It focuses on the relative position of stack pointers in the structural heap and might be applied to get more scalable heap analysis of the program with a specific form of structures, such as loops and recursive functions.

3.4 Summary

3.4.1 Comparison

Predicates play an important role in heap abstraction of shape analysis in the framework of abstract interpretation. However, TVLA and Xisa establish the abstract memory model from two different aspects. In TVLA, the heap is encoded by core predicates $q(v)$ and $r_x(v)$ and the abstract heap is established based on predicate abstraction. In Xisa, the heap is split into disjoint components and the combination of the properties of the components describes the heap property, which is a separating conjunction of inductive predicates in separation logic. The different views in memory abstraction distinguish them in terms of generality, usability, and efficiency.

Generality TVLA is more general than Xisa. Instrumentation predicates are defined to track the heap properties of interest, which is not limited to be linkage properties. For example, it can be extended to infer the ordering properties in a sorted linked list when the predicate for order relation is defined. Different form TVLA, Xisa takes advantage of checking functions to form inductive predicates. Although it is more program-specific abstraction, it can not store points-to edges as precisely as TVLA,

because the points-to information in each component is blurred.

Usability The natural difference between two methods of abstraction is that the heap predicates in TVLA and Xisa are in different levels of abstraction. The abstract predicates in TVLA take advantage of low-level and generic relations, such as concerning reachability. They should be specified as the prerequisites of the analysis, thus the precision is highly dependent on expert knowledge. In contrast to TVLA, Xisa is based on the checking functions in the source code and does not depend on the expert guidance for abstraction. However, Xisa might perform not well due to the imprecise memory abstraction when the checking functions are not available. In this case, inductive predicates might have to be defined based on types of data structure.

Efficiency Due to the advantages of separation logic in local reasoning, Xisa outperforms TVLA in terms of efficiency. Meanwhile, the techniques, such as semantic clumping [22], can merge the similar disjunctives and simplify the abstract memory state efficiently, which can cut down the total time cost in the transition of each disjunctive. In contrast, the state transition in TVLA suffers a heavy burden of time because the predicate updating is time-consuming in spite of the optimization [16]. The interesting thing is that TVLA does not need widening operator while the time cost in Xisa mainly comes from widening operation, while this advantage does not outperform the one brought from local reasoning in separation logic.

At last, TVL and SL are two orthogonal of logics and there is a work to combine them for more precise structural heap model. In the work [24], SMASLTOV exploits the infrastructure of TVLA and uses unary domain predicates to pick out regions of the heap that are of interest in the state that a logical structure models. To the best of knowledge, it is the only work which combines these two logics and connects the TVLA community and separation logic community.

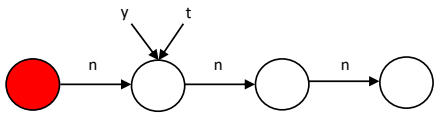
3.4.2 Applications

The applications of structural heap with pointers have a wide range of applications, including memory bug detection, concurrent program verification and automatic parallelization. In this section, we discuss two applications based on reachability and disjointness respectively. Section 5 will discuss more applications.

Memory leak detection is one of the applications of reachability. If the heap-allocated objects are not reachable from any of stack pointers, the objects should be freed immediately. Otherwise, memory leak detection occurs. Figure 3.4.2 shows the example of linked list traversal in C program. Based on the heap analyzer, such as TVLA, it can be concluded that the heap object represented by the red node in Figure 19 is not reachable from x and y. Due to the lack of *free* statement, a memory bug occurs in this example.

```
1  int main() {
2    List* y = createList();
3    List* t = NULL;
4    while (y != NULL) {
5      t = y->n;
6      y = t;
7    }
8    return 0;
9  }
```

Figure 18: An example of linked list traversal in C program



(a)

Figure 19: Points-to graph after the first iteration

Concurrent program verification benefits from the disjointness. More specifically, if two threads access disjoint heap regions, there must not be no data race bug. Moreover, the threads can be parallelized if the heap regions accessed are disjoint. For example, the heap-allocated objects reachable from x and y are disjoint in Figure 20, therefore the computation in two threads does not count a node in the binary tree more than one time, and the parallelization is safe.

```
1  void cntTree(Tree* root) {
2    Tree* x, y;
3    if (root != NULL) {
4      x = root->l;
5      y= root->r;
6
7      auto handle = async(launch::async,cntTree,y);
8      int nx = cntTree(x);
9      int count = nx+handle.get() + 1;
10
11     return count;
12   }
13   return 0;
14 }
```

Figure 20: An example of counting the nodes in a binary tree by two threads

4 Structural Heap in Containers

This section discusses property inference of structural heap with containers. According to Definition 2, it has two major differences compared with the structural heap in Section 3. Firstly, objects are organized by a mapping F that maps the key to the value its stored, and the key set contains comparable elements for indexing. Secondly, instead of manipulating structural heap by pointer operations directly, the manipulation of containers, including *add* and *get* operations, are performed by invoking a handful of library interfaces, such as *push_back* and the index operator. These two differences make us concern different properties in this structural heap from the one with pointers and they create challenge to establish the unified model for various types of containers.

In this section, we discuss how to obtain the properties of the structural heap, namely ownership and index-value correlation defined in Section 2. Section 4.1 categorizes the containers from client slides. Section 4.2 and 4.3 introduce property inference based on two memory model. At last, the comparison and the applications are appended and discussed in Section 4.4.2.

4.1 Preliminary

According to the definition in the Dictionary of Algorithms and Data Structures [25], the container, i.e., abstract data type, is a set of data values and associated operations that are precisely specified independent of any particular implementation. The implementations of containers can be based on the manipulation of structural heap in Section 3, and they are used through the interfaces from the client-side. For example, a C++ STL map can be implemented by organizing its key set based on a binary search tree. However, the underlying implementation details are not the concerns of the users, and they do not affect the understanding the contents of containers as long as the library interfaces of containers are implemented according to the same specification. The specification provides the information that whether the content is read from or written to the container, therefore the ownership can be obtained based on the specification of library interfaces.

Another important observation is that the containers, from the client-side, can be be divided into two categories, namely position-dependent containers and value-dependent containers.

- **Position-dependent containers** It stores the values sequentially, which defines the correlation of positions and values. In the structural heap, the values can be heap-allocated objects or their pointers. In C++ STL library, the position-dependent containers include *vector*, *list*, *stack*, *queue* and *deque*.

- **Value-dependent containers** It stores the values according to the keys, which defines the key-value correlation. In the structural heap, the comparable keys correspond to the heap-allocated objects or their pointers. In C++ STL library,

the value-dependent containers include *set*, *map* and their variants including *multi_set* and *hash_map*.

From the view of program analyzers, position-dependent containers can be regarded as a special case of value-dependent containers, and the position-value correlation and key-value correlation, which are two specific properties in the two types of containers, can be generalized by index-value correlation. Based on this observation, it is promising to provide a per-element understanding of container contents by analyzing index-value correlation.

```
vector<A*> v; //o0
A* a = new A(); //o1
v.push_back(a);
A* c = v.back();
```

```
map<string, S*> scores;
u["Alice"]=new S(90);
u["Bob"]=new S(80);
```

Figure 21: An example of two types of containers in C++

Example 11. Figure 21 shows the examples of a position-dependent container and value-dependent container. The index-value correlation implies that: (1) a and c points to the same heap-allocated object. (2) The score of *Alice* is 90 and the score of *Bob* is 80.

Compared with ownership, index-value correlation provides more precise properties because it reflects the layout of the storage in the containers. Section 4.2 and 4.3 discuss the inference of ownership and index-value correlation, which are based on flow graph and symbolic heap respectively.

4.2 Flow Analysis

According to the definition of ownership in Section 2, the ownership is determined by the objects flowing in and out of the container. Therefore, the ownership inference can converted into a graph reachability problem if the graph tracks the objects flow in the program.

4.2.1 Flow Graph

Despite of the variety of container instances, the library interface can be regarded as one of these three types of operations, namely allocate, load and store, which trigger the object flow of the container. The objects are abstracted based on allocation-site abstraction, which is also common used in pointer analysis in Java programs [9]. Based on this heap abstraction and the semantics of library interfaces, the flow graph can be constructed to model the flow in and out of the containers [7].

Definition 3 (Flow Graph)**.** Flow graph $G = (V_c \cup V_n, E)$ encodes object flow caused by library interfaces of containers.

- V_c and V_n model the abstract memory of the containers and the content of containers respectively. Each container c corresponds to $v_c \in V_c$. And if a container is the element of a nested container, there is also a corresponding node in V_n.
- E models the semantics of library interfaces to record the object flow. Based on the effect of library interfaces, the edge is added between a container and another object. Table 3 shows typical library interfaces and their edges in the flow graph.

Table 3: Edges in flow graph modeling interface semantics

interface	semantics	edge in flow graph
$vector < A* > v$	allocate	$o \xrightarrow{new} v$
$A * a = new\ A()$	allocate	$o \xrightarrow{new} v$
$v.push\_back(a)$	store	$a \xrightarrow{store} v$
$v[i] = a$	store	$a \xrightarrow{store} v$
$A * a = v.back()$	load	$v \xrightarrow{load} a$
$A * a = v[i]$	load	$v \xrightarrow{load} a$

Example 12. Figure 22 shows the flow graph of the first code snippet in Figure 21. The interfaces *push_back* and *back* perform a store and load operation on the container v respectively, which is allocated at line 1.

Figure 22: Flow graph of the first code snippet in Figure 21

4.2.2 Ownership Inference

In this section, it is assumed that containers can store heap-allocated objects directly or their pointers. A heap-allocated object belongs to a container if they are contained in the container or their pointers are stored. Based on the flow graph constructed in Section 4.2.1, ownership inference can be formulated by a special reachability problem in a flow graph. The ownership is determined by the statements in a sequence and edges in the flow graph are labeled based on semantics, therefore the sequences of the labels establish the ownership and the effect of the statements. According to this observation, it can be found that ownership can be discovered by solving a CFL-reachability problem, i.e., searching all the reachable paths in which the sequence of labels belongs to a context-free language. In the ownership inference problem, the context-free grammar is defined as follows [7].

Definition 4. The context-free grammar in the formulation of ownership inference is $G = (V, \Sigma, R, S)$, where the set of nonterminal character $V = \{flowsTo, ownership\}$, the finite set of terminals $\Sigma = \{new, assign, store, load\}$, and the start variable $S = ownership$. The production rule R is defined as follows.

$$\begin{aligned} flowsTo &\rightarrow new\ (assign \mid store\ load)* \\ ownership &\rightarrow \varepsilon \\ ownership &\rightarrow flowsTo\ store\ \overline{flowsTo} \end{aligned}$$

A pointer a points to a object o_1 if the sequence of labels in the path which begins with o_1 and ends with a can be derived from *flowsTo*. In the third production rule, $\overline{flowsTo}$ is a reverse language in which the sentences are derived from $flowsTo$. After searching all the pairs of the nodes in a flow graph, we can filter the pairs of which the corresponding sentences are derived from *ownership* and then the ownership is obtained.

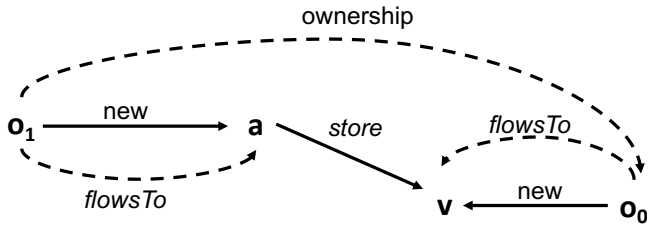


Figure 23: An example of CFL-reachability in flow graphs

Example 13. Based on the context-free grammar in Definition 4, we can figure out three paths shown in Figure 23. According to the first production rule, there is a *flowsTo* path from o_1 and a indicating that a points to o_1. Similarly, v points to the inside objects of the container. According to the third production rule, there is a *ownership* path from o_0 to o_1, which indicates that o_0 contains o_1. For nested containers, the ownership can be obtained due to the transitivity of the third production rule.

4.2.3 Imprecision of Flow Analysis

Although CFL-reachability problems can be solved at a considerable cost, flow graphs are not expressive enough to model control structure precisely. Firstly, the statement order is blurred in the flow graph. If there are more than one store operations, the object loaded out of the container can not be determined precisely. Secondly, flow graphs only record the operations, while path conditions and control structure including loops are not encoded. The flow and path insensitivity degrades the precision of flow-in and flow-out information obtained based on ownership.

```
1        vector<A*> v;                    // o0
2        v.push_back(new A());            // o1
3        for (int i = 0; i < N; i++) {
4            A* p = new A();
5            v.push_back(p);
6        }
7        A* c = v.front();
8        cout << c->data << endl;
```

Figure 24: An example of storing elements in a container within a loop

Example 14. Figure 25 shows the flow graph of the program in Figure 24. Due to the statement order is blurred in the flow graph, we can only get the fact that o_1 and o_2 flow in o_0, thus c points to o_1 or o_2, whereas the second case is spurious.

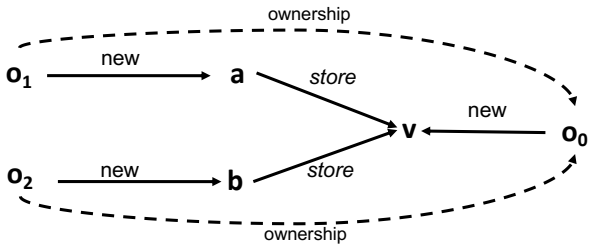


Figure 25: An example of flow graphs blurring statement order

4.3 Symbolic Heap Analysis

This section summarizes a symbolic heap model for containers and introduces how to infer index-value correlation based on the model. At the end of the section, the comparison of the symbolic heap and the abstract heap in Section 3 is discussed to illustrate the relationship between the analysis of the two types of structural heaps.

4.3.1 Symbolic Heap Model

According to Definition 2, a container is essentially a mapping from a key set to a value set. Therefore, a container can be modeled as an abstract location, in which the elements are distinguished by a function converting a key to an index [26, 8]. More specifically, the function is exactly identity function for position-dependent containers, as the key and the index are both the position of the values in the containers. For value-dependent containers, the function is invertible uninterpreted, which is used to establish the correlation between key and values. Based on this abstraction, a symbolic heap

model can be constructed to model the heap-allocated objects organized by containers and the operations in the heap.

In the symbolic heap model, a container is encoded by an abstract value set $\theta = \{(\pi_k, \phi_k)\}$, which is a set of the pairs of an abstract value π and a constraint on index variables ϕ.

- Abstract value π is the abstract location corresponding to the element of N or C in Definition 2. The abstraction is based on allocation site, and the allocation sites in each iteration of the loop are distinguished by a loop counter.
- A constraint on index variables ϕ encodes the content in abstract locations. It establishes the correlation of the abstract locations and the values at a certain position or with the same key in the container.

Figure 26 shows a general symbolic heap model for containers. X is an abstract value representing a container, and Y is an abstract values representing containers or the memory created in a loop. i_1 and i_2 are the index of the containers or the loop counter. The container can be encoded by $< X >_{i_1} = \{(o_0, \phi_1(i_1)), (< Y >_{i_2}, \phi_2(i_1, i_2))\}$, and the symbolic heap represented in this form depicts index-value correlation directly.

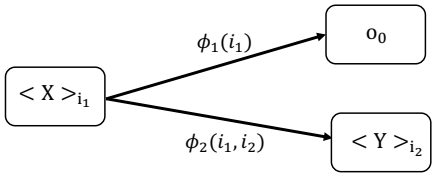


Figure 26: A symbolic heap model for containers

4.3.2 Index-Value Correlation Inference

Based on the symbolic heap, the index-value correlation can be obtained by finding all the feasible edge for a given index variable, therefore the symbolic heap at each program point needs to be computed according to the statement semantics.

Specifically, an abstract semantics of an operation is a transformation of an abstract value set θ. For the load operation, such as $front$ and $a = v[i]$, it retrieves the abstract value by checking the satisfiability of each constraint based on the key in the operation. For the store operation, such as *push_back* and $v[i] = a$, it manipulates the abstract value set by updating the constraints of index variables and adding a new abstract value pair.

Figure 27 shows a symbolic heap of the program in Figure 24. The details of load and store operations are illustrated in the next two examples.

Example 15 (Load operation)**.** For the statement $A * a = v[1]$, the loaded object can be obtained by searching along the edges with feasible constraints. In the symbolic heap, two edges from $< v >_{v_i}$ induce two invocations of a constraint solver.

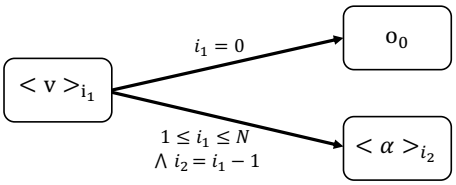


Figure 27: The symbolic heap of Figure 24

In the first invocation, $i_1 = 0 \wedge i_1 = 1$ is checked and determined as an unsatisfiable formula in integer theory, which means that a does not point to o_0. In the second invocation, $1 \le i_1 \le N \wedge i_2 = i_1 - 1 \le i_1 = 1$ is checked. It is a satisfiable formula and the solution is $i_1 = 1 \wedge i_2 = 0$. Based on the result, a is equal to the pointer which points to the heap-allocated object in the first iteration of the loop.

```
1        for (int j = 0; j < v.size(); j++) {
2          if (j % 2 == 0) {
3            v[j] = NULL;
4          }
5        }
```

Figure 28: An example of storing elements of a container

Example 16 (Store operation)**.** Figure 28 shows an example of updating the content of containers by store operations. The constraint of store operation is $i_1 \% 2 = 0$, where the literal i_1 corresponds to the loop counter j. Similar to load operation, the constraint labeled in each edge from the container v is conjuncted with the negation of constraint of indexed variable yielding the new constraint in the original abstract value pair. At last, a new abstract value pair $(NULL,\ j\%2 = 0)$ is added to the symbolic heap.

After the transformations, the symbolic heap is shown in Figure 29, which can be expressed as follows.

$$\theta = \{(o_0,\ i_1 = 0 \wedge i_1 \% 2 \le 0),\ (NULL,\ i_1 \% 2 = 0),\\ (< \alpha >_{i_2},\ 1 \le i_1 \le N \wedge i_2 = i_1 - 1 \wedge i_1 \% 2 != 0)\}$$

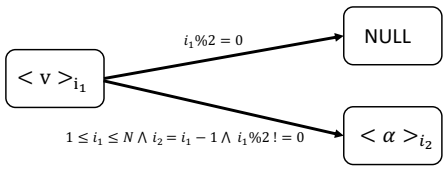


Figure 29: The symbolic heap of Figure 28

It should be noticed that Example 15 and 16 do not explain the details related to value-dependent containers. The difference is that an invertible uninterpreted function m is involved in the invocation of constraint solvers. Technically, the index-value correlation can be inferred for value-dependent containers by simply feeding the following axiom to the solver

$$\forall k_1, k_2 \in I, k_1 != k_2 \rightarrow m(k_1) != m(k_2)$$

4.3.3 Comparison with Shape Analysis

The core insight of symbolic heap analysis is that the concrete objects updated by a statement can be abstracted by the indexed constraints and the flow sensitivity is assured. This is similar to shape analysis based on abstract interpretation in Section 3, which also infers a flow-sensitive heap properties. However, there are several differences between them.

Firstly, symbolic heap analysis discussed in this section concentrates on analyzing the contents and client-side uses of these data structures, while the shape analysis aims to verify the implementations of data structure. In shape analysis, the heap-allocated objects are of the same type and they are abstracted based on linkage properties represented by some instrumentation predicates. In contrast, containers might form a hierarchical heap structure which contains different types of objects. A container is partitioned and abstracted according to the constraint on index variables. This abstraction can be applied to other data structures, such as arrays and strings. Therefore, symbolic heap analysis is complementary to shape analysis and capable of reasoning points-to target in various data structures [27].

Secondly, *bracket constraints* are proposed to perform an over-approximation and under-approximation of the heap simultaneously to deal with the uncertainty caused by complex branch conditions in the program [28]. In TVLA framework, the uncertainty of linkage property is dealt with the intermediate value 1/2, and disjunctives of three-valued logic structures are generated to get precise may and must points-to edge [4]. Compared with disjunctives of three-valued logical structures, bracket constraints in symbolic heap analysis achieve the same precision as the disjunctives of three-valued logical structure. However, symbolic heap analysis reasons about only one abstract heap per program point, which delays reasoning the disjunctives until constraint solving, thus the scalability is promising with the benefit of the optimization of a constraint solver in the satisfiability checking [26].

4.4 Summary

This section summarizes the difference of ownership obtained by flow analysis and index-value correlation obtained by symbolic heap analysis. The applications are discussed at the end, which include taint analysis and typestate analysis.

4.4.1 Comparison

Flow analysis and symbolic heap analysis aim to get the ownership and index-value correlation respectively. Index-value correlation can be regarded as a more precise ownership which maintains the position or key information. The superiority of index-value correlation is owing to two reasons. Firstly, the heap abstraction is based on constraints on index variables in symbolic heap analysis so that the positions of objects are tracked when the objects are added in the container. However, flow analysis does not distinguish the objects in the container and just takes them as a set.

Secondly, symbolic heap is generated per program location. The symbolic heap is transformed based on the constraints encoding each operation of containers, so the result is flow- and path-sensitive. However, flow graphs in Section 4.2 can not encode statement order and path condition, so the imprecision is caused in the program with multiple store operation, such as the program in Figure 24.

4.4.2 Applications

Taint analysis tracks the sensitive data as tainted information by starting with a predefined source until it reaches a given sink. In the programs with a frequent use of containers, the precision of taint analysis can not be easily assured because there are objects flow in and out of containers, which makes the propagation of tainted information sophisticated. In these scenarios, ownership and index-value correlation help to identify the tainted variable [29, 30].

In Figure 30, the first element of *v* is tainted by the return value of *source*. According to Table 3 and Definition 4, the tainted value belongs to *r* and may flow to both of *a* and *b*. Compared with the ownership, index-value correlation supports more precise taint analysis and obtain the result that only the tainted value flows in the container as the first element and then sinks in *a*.

```
1        vector<A*> r;
2        r[0] = source();
3        r[1] = NULL;
4        A* a = r[0];
5        A* b = r[1];
```

Figure 30: An example of taint analysis

Typestate analysis addresses the temporal safety properties of object, most of which are heap-allocated objects, by encoding usage rules for library interfaces, such as APIs, and these objects are commonly organized by the container [31]. The example shown in Figure 31 demonstrates the importance of index-value correlation. Ownership blurs the fact that where the value is stored in the container, thus it causes a false positive at line 10. In contrast, index-value correlation can precisely detect the bug at

line 9 without reporting false positives.

In the presence of aliasing, the combination of over- and under-approximations supports a more precise update of typestate when the bracket constraints are utilized [32]. The similar works include *strong update* in general data flow problem [33].

```
1   vector<File*> v;
2   for (int k = 0; k <= 10; k++) {
3     File* a = malloc();
4     v.push_back(a);
5     if (k % 2) {
6       a->open();
7     }
8   }
9   v[0]->write("error");
10  v[1]->write("ok");
```

Figure 31: An example of typestate analysis

5 Applications

In the section, we discuss four clients which take advantage of heap analysis. The works listed in this section show the importance of heap analysis in the bug detection and program understanding of heap manipulating program.

5.1 Memory Corruption

Memory corruption is a secure bug type and commonly exists in real-world programs, especially in the programs with intensive use of heap. Pointer manipulation and abstract interfaces, which makes programming effective, also increases the difficulty of detection. Imprecision of heap model are the root cause of false positives and negatives. In the structural heap organized by pointers, precise heap properties, including sharing and aliasing relation, can improve the precision of memory leak detection by performing flow-sensitive analysis [4, 34]. In the structural heap organized by containers, ownership and index-value correlation can applied to prove memory safety properties because precise object flow is obtained [28]. Moreover, function summaries are proposed in many works to improve the scalability of the analysis [19, 35].

5.2 FSM State Error

A faulty finite state machines (FSM) state error occurs when the manipulations are driven an object into an undesirable state, such as use of expired file descriptor and use after free [36]. In the analysis of the resource maintained in the heap-allocated objects, it is necessary to cut through the tangle of aliasing by which a pointer manipulates the

objects of interest to assure the precision. Typestate analysis provides a formal solution to this problem [32, 33]. Example 5 introduces an FO^{TC} formula in three-valued logic to encode aliasing, and the related applications include garbage collection based on temporal heap safety properties verification [37].

5.3 Memory Safety of Multi-threaded Program

Thread interference bug commonly occurs in multi-threaded programs. For example, if two threads access the shared memory concurrently and at least one performs a write operation, then data race occurs, which is a typical thread interference error. However, it is difficult to detect thread interference error in a multi-thread program if it mutates data structure through pointer manipulation, and it is challenging to identify shared memory in multi-threaded program. Heap properties, such as points-to relation and aliasing, are necessary to carve out the shared memory, but they might not be statically determinable because of dynamic heap manipulation.

Fortunately, heap abstraction in Section 3 provides the inspiration to approximate the shared memory. On the one hand, three-valued logic formula is powerful to abstract unbounded structure. In order to get the shared memory for each thread, resource invariants can be computed to carve out the shared memory [38]. For the program with dynamic thread creation, the threads and memory are both abstracted in a uniform model to assure the soundness of analysis [39]. On the other hand, separation logic formula is nature to describe the heap memory accessible from the parameter of the function, so it is more suitable for scalable concurrency program analysis [18, 40].

5.4 Heap-manipulating Program Understanding

Dynamic manipulation on the objects in the heap improves the difficulty of debugging when an error occurs. Moreover, the various types of data structure make it challenging to understand the semantics of the program with complex heap manipulations, which pose the obstacle to program maintenance. DOrder [23] generates the specification of interface, which reflects the structural and ordering relations, such as the relationship of input and output of in-order traversal of a binary tree. SLING [11] synthesizes and validates a separation logic formula describing the memory invariants at a certain program location, which can help to explain bugs and identify false positives. These two works both reply on runtime heap configurations and are implemented via a CEGIS(counterexample guided inductive synthesis) loop, which is a new trend of shape analysis [41, 42].

Compared with DOrder and SLING, static analysis techniques, including TVLA framework and XISA, are also able to provide important memory information by generating the program invariants. However, the whole heap memory model prevents them being applying in the understanding in real-world programs. Demanded-driven approaches are proposed to infer the specification according to particular scenarios, such

as identifying statement-achieving statements in the detection of bloat containers [7].

6 Conclusion

Heap analysis is a general collection of analyses that aim to infer linkage properties of heap-allocated objects. Different from other analyses, such as pointer analysis, it focuses on more high-level properties including reachability and ownership. In the survey, we have presented heap analysis techniques for heap property inference in two types of the structural heap organized by pointers and containers respectively.

Static shape analysis addresses the analysis of structural heap with pointers in the framework of abstract interpretation. The structural heap contains the heap-allocated objects which form a self-similar structure. Heap properties, such as reachability and disjointness, are obtained in the transition of the abstract heap, which is encoded in logics. Although shape analysis is capable of expressing precise and general heap properties, it is still limited in scalability because of time cost in constraint solving and the need for manual work in encoding predicates for heap abstraction.

Flow analysis and symbolic heap analysis analyze structural heap with containers, in which the objects are not necessarily in the same type and might in a hierarchical structure. Ownership and index-value correlation are heap properties of interest and useful to get object flow information. Symbolic heap analysis performs a flow-sensitive analysis to obtain index-value correlation, and its practicality relies on the optimization of constraint solving techniques. In contrast, flow analysis produces a flow-insensitive result to infer ownership, and the index of the values in containers are blurred.

The heap properties benefit many clients, such as memory corruption detection and typestate verification. Points-to relation and typestate of objects can be obtained and updated precisely according to the properties. For some specific programs, including multi-threaded programs, separation logic-based shape analysis outlines the border of shared memory, which is meaningful for data race detection. Besides, the properties can also help developers understand program better and promote program maintenance and debugging.

Flow-sensitive heap analysis assures the precision but suffers the heavy time burden. Complex program structures, such as loops and recursive functions, make it even more challenging. With the inspiration of separation logic in concurrent program verification, it is promising to propose a new form of representation to model structural heap in a specific program. In the future, it might be feasible to analyze the structural heap manipulated in a restricted form of program, such as structural heap manipulated in a loop, and this would generate strong implications of stack pointers.

References

[1] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[2] Earl T Barr, Christian Bird, and Mark Marron. Collecting a heap of shapes. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 123–133, 2013.

[3] Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, 2016.

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

[5] Dino Distefano, Peter W Ohearn, and Hongseok Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer, 2006.

[6] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pages 384–401, 2007.

[7] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 160–173, 2010.

[8] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. *ACM SIGPLAN Notices*, 46(1):187–200, 2011.

[9] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *ACM SIGPLAN Notices*, 40(10):59–76, 2005.

[10] Tian Tan, Yue Li, and Jingling Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–291, 2017.

[11] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. SLING: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 788–801, 2019.

[12] Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. Shape analysis and applications. In *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, page 12. 2007.

[13] Wikipedia. Abstract interpretation. `https://en.wikipedia.org/wiki/Abstract_interpretation`. Accessed Feb 10, 2020.

[14] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2):5:1–5:52, 2010.

[15] Noam Rinetzky and Shmuel Sagiv. Interprocedural shape analysis for recursive programs. In *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 133–149, 2001.

[16] Thomas W. Reps, Shmuel Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 380–398, 2003.

[17] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002.

[18] Peter W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019.

[19] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[20] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 247–260, 2008.

[21] Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 173–186, 2011.

[22] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. 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*, pages 32–45, 2017.

[23] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 491–507, 2016.

[24] Aditya V. Thakur, Jason Breck, and Thomas W. Reps. Satisfiability modulo abstraction for separation logic with linked lists. In *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*, pages 58–67, 2014.

[25] Dictionary of Algorithms and Data Structures. Abstract data type. `https://xlinux.nist.gov/dads/HTML/abstractDataType.html`. Accessed Aug 31, 2020.

[26] Ayse Isil Dillig. *Precise and Automatic Verification of Container-manipulating Programs*. PhD thesis, Stanford University, 2011.

[27] Isil Dillig, Thomas Dillig, and Alex Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. *ACM Sigplan Notices*, 45(10):397–410, 2010.

[28] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Symposium on Programming*, pages 246–266. Springer, 2010.

[29] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[30] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 725–735. IEEE, 2016.

[31] Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. Arc++: effective and lifetime dependency analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 116–126, 2014.

[32] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 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*, pages 133–144, 2006.

[33] Johannes Späth, Karim Ali, and Eric Bodden. Ideal: efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):99–1, 2017.

[34] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. Checking cleanness in linked lists. In *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*, pages 115–134, 2000.

[35] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 567–577. ACM, 2011.

[36] Common Weakness Enumeration. Cwe-1245: Improper finite state machines (fsms) in hardware logic. `https://cwe.mitre.org/data/definitions/1245.html`. Accessed Sept 7, 2020.

[37] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Shmuel Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pages 483–503, 2003.

[38] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 266–277, 2007.

[39] Eran Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 27–40, 2001.

[40] Stephen Brookes and Peter W O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.

[41] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. Learning shape analysis. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, pages 66–87, 2017.

[42] Long H. Pham, Jun Sun, and Quang Loc Le. Compositional verification of heap-manipulating programs through property-guided learning. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, pages 405–424, 2019.

[43] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. Abstract acceleration of general linear loops. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 529–540, 2014.

[44] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 61–72, 2016.

[45] Ke Wang and Zhendong Su. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 2020.