"What's in a Name?" Going Beyond Allocation Site Names in Heap Analysis

Vini Kanvar * and Uday P. Khedker
Indian Institute of Technology Bombay, India
{vini,uday}@cse.iitb.ac.in

Abstract

A points-to analysis computes a sound abstraction of heap memory conventionally using a *name-based* abstraction that summarizes runtime memory by grouping locations using the names of allocation sites: All concrete heap locations allocated by the same statement are grouped together. The locations in the same group are treated alike i.e., a pointer to any one location of the group is assumed to point to every location in the group leading to an over-approximation of points-to relations.

We propose an *access-based* abstraction that partitions each name-based group of locations into equivalence classes at every program point using an additional criterion of the sets of access paths (chains of pointer indirections) reaching the locations in the memory. The intuition is that the locations that are both allocated and accessed alike should be grouped into the same equivalence class. Since the access paths in the memory could reach different locations at different program points, our groupings change flow sensitively unlike the name-based groupings. This creates a more precise view of the memory. Theoretically, it is strictly more precise than the name-based abstraction except in some trivial cases; practically it is far more precise.

Our empirical measurements show the benefits of our tool Access-Based Heap Analyzer (ABHA) on SPEC CPU 2006 and heap manipulating SV-COMP benchmarks. ABHA, which is field-, flow-, and context-sensitive, scales to 20 kLoC and can improve the precision even up to 99% (in terms of the number of aliases). Additionally, ABHA allows any user-defined summarization of an access path to be plugged in; we have implemented and evaluated four summarization techniques. ABHA can also act as a front-end to TVLA, a parametrized shape analyzer, in order to automate its parametrization by generating predicates that capture the program behaviour more accurately.

CCS Concepts • **Theory of computation** → **Program analysis**; • **Software and its engineering** → *Software verification*

General Terms Design, Algorithms, Verification, Languages

Keywords access path, alias, allocation site, heap abstraction, static points-to analysis, summarization

* Partially supported by a TCS fellowship.

ISMM'17, June 18, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5044-0/17/06...$15.00
http://dx.doi.org/10.1145/3092255.3092267

1. Introduction

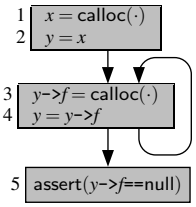
Pointers allow indirect manipulation of data and hence obscure program understanding as well as the results of program analyses. An accurate pointer analysis drives the precision, and consequently, the effectiveness of many program optimizations, reduces the number of false positives in program verification, and also helps in constructing a precise control flow graph by resolving the callees in indirect calls. Static heap analysis is also useful for discovering heap memory bugs like memory leaks, null dereferences, dangling pointers, multiple deallocation of the same memory location, etc. It is also helpful in identifying inefficient usages of heap like delayed deallocation, unnecessary use of heap instead of stack, poor cache utilization, and others [10].

Pointer information in a program is discovered by constructing abstractions of the concrete memory created by the pointer assignments in the program. A popular abstraction of memory views it as a set of interconnected chains of pointer indirections naturally represented by a *points-to graph* [4] in which nodes denote concrete memory locations and directed edges denote concrete memory links representing the address of a target location stored in a pointer location. Here onwards, we use locations and links to mean concrete memory locations and links, respectively. An alternative abstraction views memory as a set of *access paths* [16]. Semantically, it is a static representation of paths in the concrete memory created by a program. Syntactically, an access path is a variable followed by a sequence of field dereferences. Variables indicate both stack and global variables.

Formally, $\mathbb{\Sigma} \subseteq \mathbb{V} \times \mathbb{F}^*$ denotes a set of access paths where $\mathbb{V}$ and $\mathbb{F}$ are the sets of variables and fields, respectively. For example, access path $x.f.f.f.g$ represents a memory location reachable from variable x via 3 indirections of field f and then that of field g. In the presence of dynamic allocations, the number of access paths is potentially infinite and the length of access paths is unbounded. Therefore, we need to summarize the access paths. We represent summarized access paths by a regular expression, for example, $x.f(.f)^*.g$ is used as a shorthand for the following access paths: $x.f.g, x.f.f.g, x.f.f.f.g, \ldots$. Therefore, we can say $x.f.g \in \{x.f(.f)^*.g\}$.

A *points-to analysis* computes pointer information in terms of a points-to graph whereas an *alias analysis* computes sets of aliased access paths. Points-to analysis of heap manipulating programs is typically based on a *name-based abstraction* in which all concrete memory locations created by the same statement id or site (for example, $x = \mathsf{calloc}(\cdot)$) are grouped together and are *treated alike* i.e., a pointer to any one location of the group is over-approximated with a pointer to every location in the group [4, 9, 24].

Let $\mathbb{S}$ denote the set of program statements (or sites). For a flow-sensitive points-to analysis, we define $\mathbb{Q} = \{\mathsf{in}, \mathsf{out}\} \times \mathbb{S}$, as the set of program points. in_s and out_s denote program points before and after statement s, where $s \in \mathbb{S}$. The concrete memory and its name-



Execution snapshots	Figure 2a
Name-based graph at in_5	Figure 2b
Access-based graphs	Figures 2b, 4, 5
Mapping concrete locations to abstract nodes	Figure 2c

Figure 1: Our motivating example. Loop conditions in the program are not analyzed and are not shown for simplicity. Different aspects of our analysis and the figures that illustrate them are listed on the right.

based abstraction is modelled in terms of points-to graphs. Initially we assume that only heap locations have fields and the variables are not addressable. These assumptions are relaxed in Section 5.3.

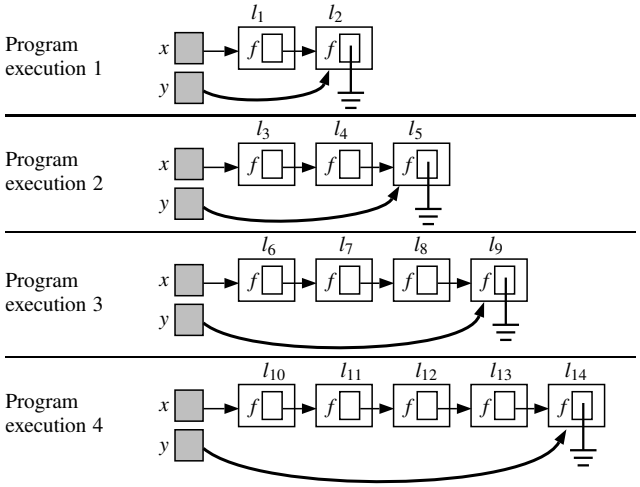
Example 1. We use the program in Figure 1 as a running example to illustrate flow-sensitive points-to analysis of heap using name-based and access-based abstractions. The program creates a null-terminated linked list rooted at variable x connected via field f. The execution snapshots of the grounded lists created at in_5 are shown in Figure 2a in terms of heap locations from l_1 to l_{14} where x and y point to the head and the tail, respectively of the list. Variable y is used to traverse the list and append a new node at the end of the list. The assertion in statement 5 checks that the list must be null-terminated. This holds because the fields of all nodes are initialized to null by calloc(·). The concrete heap locations l_1, l_3, l_6, l_{10} are allocated at site 1 of the program. The remaining heap locations like l_2, l_4, l_5, l_7 are allocated at site 3 of the program. □

A name-based graph is a points-to graph where each heap node (or name-based heap node, when needed for clarity) represents a group of concrete heap locations that are allocated at the same site. We show below the imprecision of the name-based abstraction.

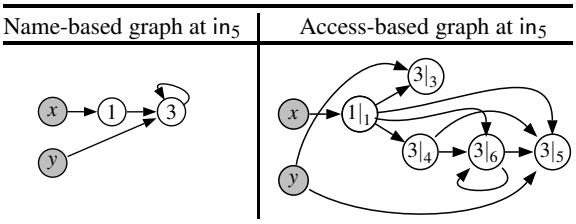
Example 2. The result of a flow-sensitive points-to analysis using name-based abstraction at in_5 of our motivating example is shown in Figure 2b. Node 1 represents locations l_1, l_3, l_6, l_{10}, which are allocated at site 1 of the program. Node 3 represents the remaining locations allocated at site 3 of the program thereby merging the last node of the list shown in Figure 2a with other nodes resulting in a self-loop over node 3. This suggests that $y\text{->}f$ may be non-null. Hence, the assertion in statement 5 of the program cannot be verified using name-based abstraction although no execution of the program would violate it. □

We propose an *access-based abstraction* for nodes in a points-to graph. It partitions the set of memory locations represented by a name-based heap node s into equivalence classes based on the set of access paths α reaching the locations in the memory. It represents each equivalence class by an access-based node $\langle s, \alpha \rangle$ or alternatively by $s|_i$, where i uniquely identifies α.

Example 3. The result of a flow-sensitive points-to analysis using access-based abstraction at in_5 of our motivating example is shown in Figure 2b. Observe that it discovers that the locations representing the internal nodes of the linked list are not pointed to by y; only the last node is pointed to by y suggesting that $y\text{->}f$ is null. Figure 2c shows how nodes $1|_1$ and $3|_3$ to $3|_6$ in the access-based graph are created by partitioning the set of locations represented by name-based nodes 1 and 3, respectively. Access-based node $1|_1$ represents an equivalence class $\{l_1, l_3, l_6, l_{10}\}$ in a partition of the set of locations represented by name-based node 1, and reachable by set of access paths $\{x\}$. Access-based node $3|_3$ represents an equivalence class $\{l_2\}$ in a partition of the set of locations represented by name-based node 3, and reachable by set of access paths $\{x.f, y\}$. Similarly,



(a) Execution snapshots of the heap memory at in_5 for up to four iterations of the loop body. Gray boxes denote concrete locations for variables x and y, white boxes (l_1 to l_{14}) denote concrete heap locations, and f represents field.



(b) Name-based and access-based abstractions at in_5. All out-edges of heap nodes represent field f; their labels are omitted to avoid clutter.

Name-based heap node	Access-based heap node		Equivalence class of concrete heap locations
s	$s\|_i$	$\langle s, \alpha \rangle$	
1 (Site 1)	$1\|_1$	$\langle 1, \{x\} \rangle$	$\{l_1, l_3, l_6, l_{10}\}$
3 (Site 3)	$3\|_3$	$\langle 3, \{x.f, y\} \rangle$	$\{l_2\}$
	$3\|_4$	$\langle 3, \{x.f\} \rangle$	$\{l_4, l_7, l_{11}\}$
	$3\|_5$	$\langle 3, \{x.f(.f)^*, y\} \rangle$	$\{l_5, l_9, l_{14}\}$
	$3\|_6$	$\langle 3, \{x.f(.f)^*\} \rangle$	$\{l_8, l_{12}, l_{13}\}$

(c) Name-based group s of locations is partitioned into access-based group $s|_i$ of locations based on the access paths α reaching the locations. Other partitions of locations are shown in Figures 4 and 5.

Figure 2: Analysis of the program in Figure 1.

access-based nodes $3|_4, 3|_5, 3|_6$ represent other equivalence classes in a partition of the set of locations represented by name-based node 3 based on the access paths reaching the locations. The access-based graph has no out-edge from nodes $3|_3$ and $3|_5$, that are pointed to by y. Thus, access-based abstraction is able to verify the assertion because of increased precision. □

Our main contributions are: *(1)* Our heuristic makes distinctions between heap locations using access paths reaching them (and not just their allocation sites or variables pointing to them [29]). Thus, it avoids distinctions between the locations that may not add value to a static analysis and makes those that may be meaningful. *(2)* We present a novel way of using allocation sites to summarize access paths more precisely. Besides, our tool Access-Based Heap Analyzer (ABHA[1]) allows any user-defined

[1] The word ABHA also means splendour in Sanskrit and derived languages.

summarization of access path to be plugged in. *(3)* ABHA captures the program behaviour automatically and we show how it can be used as a front-end for a parametric shape analyzer, TVLA.

Section 2 presents our key ideas which are formulated as a data flow analysis in Section 3. Convergence of our analysis is ensured in Section 4. Section 5 extends access-based abstraction to allow strong updates, perform interprocedural analysis, and handle the addressof operator and non-pointer fields. Section 6 shows how ABHA can be used as a front-end for shape analysis. Implementation and measurements are described in Section 7. Section 8 presents the related work. Section 9 concludes the paper.

2. Access-Based Abstraction: Key Idea

Here we define our criterion of abstracting locations.

2.1 Concrete Semantics: Notations and Terminologies

Let $\mathbb{\Pi}$ denote the set of execution paths of a program. Each execution path is a valid sequence of possibly repeating program statements starting from the start statement. Let $\pi_q \in \mathbb{\Pi}$ denote an execution path that reaches program point $q \in \mathbb{Q}$. Let $\mathbb{H}^{\mathsf{cm}}$ and $\mathbb{E}^{\mathsf{cm}}$ denote sets of all possible heap nodes (or locations) and edges (or links), respectively in the concrete memory. We define a *concrete memory graph* at q created by an execution path π_q, denoted $\mathsf{G}^{\mathsf{cm}}(\pi_q) = \langle \mathbb{V} \cup \mathsf{H}^{\mathsf{cm}}(\pi_q), \mathsf{E}^{\mathsf{cm}}(\pi_q) \rangle$, as a pair of sets of nodes and edges. The node set $\mathbb{V} \cup \mathsf{H}^{\mathsf{cm}}(\pi_q)$ is a discriminated union of the set of variables $\mathbb{V}$, and the set of heap nodes $\mathsf{H}^{\mathsf{cm}}(\pi_q) \subseteq \mathbb{H}^{\mathsf{cm}}$ at q. The edge set $\mathsf{E}^{\mathsf{cm}}(\pi_q) \subseteq \mathbb{E}^{\mathsf{cm}}$ for $\mathsf{E}^{\mathsf{cm}}(\pi_q) = \mathsf{EV}^{\mathsf{cm}}(\pi_q) \cup \mathsf{EH}^{\mathsf{cm}}(\pi_q)$ at q is a discriminated union of

- $\mathsf{EV}^{\mathsf{cm}}(\pi_q) : \mathbb{V} \to \mathsf{H}^{\mathsf{cm}}(\pi_q)$ is the set of edges of the kind $v \to l$ with a variable as their source node, and
- $\mathsf{EH}^{\mathsf{cm}}(\pi_q) : \mathsf{H}^{\mathsf{cm}}(\pi_q) \times \mathbb{F} \to \mathsf{H}^{\mathsf{cm}}(\pi_q)$ is the set of labelled edges of the kind $l \xrightarrow{f} l'$ with their source node in the heap.

Since we treat each execution as distinct, each execution path reaching a program point uses unique heap memory locations:

$$\big(\pi_q \neq \pi'_q \ \wedge\ l \in \mathsf{H}^{\mathsf{cm}}(\pi_q) \ \wedge\ l' \in \mathsf{H}^{\mathsf{cm}}(\pi'_q)\big) \Rightarrow l \neq l' \qquad (1)$$

Definition 1. *Allocation site of a location.* Each heap location is mapped to its allocation site by function $\mathsf{HST}^{\mathsf{cm}} : \mathbb{H}^{\mathsf{cm}} \to \mathbb{S}$. □

Definition 2. *Location reachable by an access path.* Access path $\sigma \in \mathbb{\Sigma}$ is said to *reach* location $l \in \mathsf{H}^{\mathsf{cm}}(\pi_q)$, if $l = \mathsf{Tgt}^{\mathsf{cm}}(\sigma, \pi_q)$ where,

$$\mathsf{Tgt}^{\mathsf{cm}}(\sigma, \pi_q) = \begin{cases} l & v \in \mathbb{V}.\ (\sigma = v \ \wedge\ v \to l \in \mathsf{EV}^{\mathsf{cm}}(\pi_q)) \\ l & f \in \mathbb{F}.\ (\sigma = \sigma'.f \ \wedge\ l' = \mathsf{Tgt}^{\mathsf{cm}}(\sigma', \pi_q) \ \wedge \\ & \quad l' \xrightarrow{f} l \in \mathsf{EH}^{\mathsf{cm}}(\pi_q)) \end{cases}$$

$\mathsf{Tgt}^{\mathsf{cm}}(\cdot)$ is a partial function because it returns a location reachable by σ only if all dereferences in σ reach a non-null location. □

Definition 3. *Access paths reaching a location.* Function $\mathsf{HAP}^{\mathsf{cm}}(l, \pi_q)$ returns a set of access paths that reach location l at program point q. Since these access paths reach the same location, they are aliased to each other.

$$\mathsf{HAP}^{\mathsf{cm}}(l, \pi_q) = \{\sigma \mid l = \mathsf{Tgt}^{\mathsf{cm}}(\sigma, \pi_q)\}$$ □

The conventional name-based abstraction groups concrete heap locations into abstract nodes using allocation sites (Section 2.2). For our abstraction, we first explain how to group them using accesses (Section 2.3), and then using both allocation sites and accesses (Section 2.4). We follow a recent dichotomy of *heap abstraction* [10] that views the grouping of concrete locations into possibly infinite abstract nodes as *heap modelling* (Section 2), and grouping of abstract nodes into a finite number of abstract nodes as *summarization* (Section 4.1).

2.2 Grouping Locations Using Allocation Sites

The conventional name-based abstraction is formalized below.

Definition 4. *Equivalence using allocation sites.* Locations l and l' are equivalent at program point q based on allocation sites, denoted $l \overset{st}{\simeq}_q l'$, if they are allocated at the same site. Using Definition 1,

$$l \overset{st}{\simeq}_q l' \Leftrightarrow \big(l \in \mathsf{H}^{\mathsf{cm}}(\pi_q) \ \wedge\ l' \in \mathsf{H}^{\mathsf{cm}}(\pi'_q) \ \wedge \\ \mathsf{HST}^{\mathsf{cm}}(l) = \mathsf{HST}^{\mathsf{cm}}(l')\big)$$ □

Relation $\overset{st}{\simeq}_q$ is an equivalence relation because it is reflexive ($l \overset{st}{\simeq}_q l$), symmetric ($l \overset{st}{\simeq}_q l' \Leftrightarrow l' \overset{st}{\simeq}_q l$), and transitive ($l \overset{st}{\simeq}_q l', l' \overset{st}{\simeq}_q l'' \Rightarrow l \overset{st}{\simeq}_q l''$). Therefore, an equivalence class of locations created using $\overset{st}{\simeq}_q$ is uniquely identified by its allocation site s and is represented by name-based node s.

2.3 Grouping Locations Using Accesses

Here, we introduce how to group locations that are *accessed alike*.

Definition 5. *Equivalence using accesses.* Locations l and l' are equivalent at program point q based on accesses, denoted $l \overset{ac}{\simeq}_q l'$, if: Whenever l is accessed using access path σ along some execution path reaching q, l' is also accessed using σ along some execution path also reaching q, and vice-versa. Using Definition 2,

$$l \overset{ac}{\simeq}_q l' \Leftrightarrow \big(l \in \mathsf{H}^{\mathsf{cm}}(\pi_q) \ \wedge\ l' \in \mathsf{H}^{\mathsf{cm}}(\pi'_q) \ \wedge \\ \big(\forall \sigma \in \mathbb{\Sigma}.\big(l = \mathsf{Tgt}^{\mathsf{cm}}(\sigma, \pi_q) \Leftrightarrow l' = \mathsf{Tgt}^{\mathsf{cm}}(\sigma, \pi'_q)\big)\big)\big)$$ □

There may be over-approximation in the equivalence relation because of summarization of access paths $\sigma \in \mathbb{\Sigma}$ (Section 4.1) which over-approximates the access paths.

Example 4. Assume that only four heap locations l_1, l_2, l_3, and l_4 are created along execution paths reaching program point q. Further assume that these locations are reachable by only three access paths σ_1, σ_2, and σ_3 such that $\mathsf{Tgt}^{\mathsf{cm}}(\sigma_1, \pi^1_q) = l_1$, and $\mathsf{Tgt}^{\mathsf{cm}}(\sigma_1, \pi^2_q) = l_2$ (i.e. at q, σ_1 reaches l_1 for execution path π^1_q and l_2 for execution path π^2_q). Similarly, let access path σ_2 reach locations l_1, l_2, l_3, l_4 at q, and let σ_3 reach location l_3 at q for their respective execution paths. In order to identify locations that could possibly be accessed by a statement immediately after program point q, consider the access paths that the statement could traverse in the memory:

- If it traverses σ_1, it could reach locations l_1 and l_2.
- If it traverses σ_2, it could reach locations l_1, l_2, l_3, and l_4.
- If it traverses σ_3, it will reach location l_3.

This covers all possible ways of accessing a memory at q. Observe that l_1 and l_2 either appear together or neither of them appears in the possibly accessed locations. Therefore, the effect of the statement at q on l_1 and l_2 is identical. □

The relation $\overset{ac}{\simeq}_q$ is an equivalence relation because it is reflexive ($l \overset{ac}{\simeq}_q l$), symmetric ($l \overset{ac}{\simeq}_q l' \Leftrightarrow l' \overset{ac}{\simeq}_q l$), and transitive ($l \overset{ac}{\simeq}_q l', l' \overset{ac}{\simeq}_q l'' \Rightarrow l \overset{ac}{\simeq}_q l''$). For Example 4, the equivalence classes created by $\overset{ac}{\simeq}_q$ at q are: $\{l_1, l_2\}$, $\{l_3\}$, and $\{l_4\}$.

Observe that the above equivalence is not easy to compute since it requires enumerating all heap locations. A more amenable way of creating equivalence classes of locations (subject to summarization of access paths) is as follows.

Definition 6. *Equivalence using accesses.* The equivalence relation $\overset{ac}{\simeq}_q$ at program point q is alternatively defined below using Definitions 3 and 5.

$$l \overset{ac}{\simeq}_q l' \Leftrightarrow \big(l \in \mathsf{H}^{\mathsf{cm}}(\pi_q) \wedge l' \in \mathsf{H}^{\mathsf{cm}}(\pi'_q) \wedge \mathsf{HAP}^{\mathsf{cm}}(l,\pi_q) = \mathsf{HAP}^{\mathsf{cm}}(l',\pi'_q)\big)$$ □

Locations that are accessed alike at program point q are statically indistinguishable at q based on the set of access paths reaching them in the memory at q.

Example 5. The access paths for locations at q in Example 4 are:

$$\mathsf{HAP}^{\mathsf{cm}}(l_1,\pi^1_q) = \{\sigma_1,\sigma_2\} \qquad \mathsf{HAP}^{\mathsf{cm}}(l_2,\pi^2_q) = \{\sigma_1,\sigma_2\}$$
$$\mathsf{HAP}^{\mathsf{cm}}(l_3,\pi^3_q) = \{\sigma_2,\sigma_3\} \qquad \mathsf{HAP}^{\mathsf{cm}}(l_4,\pi^4_q) = \{\sigma_2\}$$

The equivalence classes $\{l_1,l_2\}$, $\{l_3\}$, and $\{l_4\}$ are uniquely identified by the set of access paths reaching them viz., $\{\sigma_1,\sigma_2\}$, $\{\sigma_2,\sigma_3\}$, and $\{\sigma_2\}$, respectively. □

2.4 Grouping Locations Using Allocation Sites and Accesses

The criteria based on which locations are grouped and therefore treated alike in access-based abstraction is formalized below.

Definition 7. *Equivalence using allocation sites and accesses.* Locations l and l' are equivalent at program point q under access-based abstraction, denoted $l \simeq_q l'$, if they are equivalent based on both allocation sites and accesses. Using Definitions 4 and 6,

$$l \simeq_q l' \Leftrightarrow l \overset{st}{\simeq}_q l' \wedge l \overset{ac}{\simeq}_q l'$$ □

Since both $\overset{st}{\simeq}_q$ and $\overset{ac}{\simeq}_q$ are equivalence relations, their intersection $\simeq_q$ is also an equivalence relation. Each equivalence class of locations created using $\simeq_q$ is uniquely identified by an allocation site $s \in \mathbb{S}$ and a set of access paths $\alpha \subseteq \mathbb{\Sigma}$ of the locations. We represent each equivalence class by an access-based node $\langle s,\alpha\rangle$ or alternatively by $s|_i$, where i uniquely identifies the set of access paths α. Our key idea (Definition 7) using the role of accesses and the conventional role of allocation sites is as follows:

> *Access-based abstraction at program point q groups the following locations together: the locations that have been allocated at the same site and are being accessed alike at q.*

Example 6. Figure 2a shows heap locations l_1 through l_{14} created at in_5. Using Definition 7, the equivalence classes of locations, their associated access-based nodes, allocation sites $s \in \mathbb{S}$, and sets of access paths $\alpha \subseteq \mathbb{\Sigma}$ are shown in Figure 2c. Although access paths *x.f.f* and *x.f.f.f* reach l_{12} and l_{13}, respectively (Figure 2a), these are summarized to $x.f(.f)^*$. Thus, l_{12} and l_{13} are put in the same equivalence class represented by access-based node $3|_6$ at in_5. □

Grouping locations reduces the amount of information while keeping locations in different equivalence classes preserves precision. We attempt to avoid partitions that may not add value to a static analysis and make those that may be meaningful.

Although it is possible to group the locations based on accesses alone, we also use allocation sites for summarization (Section 4.1).

3. Computing Access-Based Abstraction

Here we compute the abstraction using data flow analysis as per Definition 7. We use the following basic pointer assignments in C: $x = \&y$, $x = y$, $x = *y$, $*y = x$, $x = y\text{->}f$, $x\text{->}f = y$, $x = \&(y\text{->}f)$, $x = y.f$, $x.f = y$, $x = \&(y.f)$, $x = \mathsf{null}$, and $x = \mathsf{calloc}(\cdot)$. For simplicity of exposition, in this section, we exclude the addressof operator "&" and non-pointer fields. Besides, we model the indirection operator "$*$" as another field. Section 5.3 handles the features excluded here.

Stmt s	Lptr_s	Lfield_s	Rpte_s
$x = \mathsf{null}$	$\{x\}$	$\emptyset$	$\emptyset$
$x = \mathsf{calloc}(\cdot)$	$\{x\}$	$\emptyset$	$\{s\|_0\}$
$x\text{->}f = \mathsf{calloc}(\cdot)$	$\mathsf{T}_s(x)$	$\{f\}$	$\{s\|_0\}$
$x = y$	$\{x\}$	$\emptyset$	$\mathsf{T}_s(y)$
$x = y\text{->}f$	$\{x\}$	$\emptyset$	$\bigcup_{n \in \mathsf{T}_s(y)} \mathsf{T}_s(n,f)$
$x\text{->}f = y$	$\mathsf{T}_s(x)$	$\{f\}$	$\mathsf{T}_s(y)$

$$\mathsf{T}_s(v) = \{n \mid v \to n \in \mathsf{Ein}_s\} \qquad \mathsf{T}_s(m,g) = \{n \mid m \xrightarrow{g} n \in \mathsf{Ein}_s\}$$

Figure 3: Extractor functions for statement s. Lptr_s and Rpte_s denote l-value and r-value of s. Other C statements are handled in Section 5.3.

3.1 Abstract Semantics: Notations and Terminologies

Recall that the concrete memory created by an execution path π_q is a graph $\mathsf{G}^{\mathsf{cm}}(\pi_q) = \langle \mathbb{V} \cup \mathsf{H}^{\mathsf{cm}}(\pi_q), \mathsf{E}^{\mathsf{cm}}(\pi_q)\rangle$. Let $\mathbb{H}$ and $\mathbb{E}$ denote the sets of all possible access-based heap nodes and edges. Analogously, we define an *access-based graph* at program point q, denoted $\mathsf{G}_q = \langle \mathbb{V} \cup \mathsf{H}_q, \mathsf{E}_q\rangle$, as a pair of sets of nodes and edges (called access-based nodes and edges, when needed for clarity) as follows. The node set $\mathbb{V} \cup \mathsf{H}_q$ is a discriminated union of the set of variables $\mathbb{V}$ and the set of heap nodes $\mathsf{H}_q \subseteq \mathbb{H}$. The edge set $\mathsf{E}_q \subseteq \mathbb{E}$, where $\mathsf{E}_q = \mathsf{EV}_q \cup \mathsf{EH}_q$ is a discriminated union of:

- $\mathsf{EV}_q \subseteq \mathbb{V} \times \mathsf{H}_q$ is the set of edges of the kind $v \to n$ with a variable as their source node, and
- $\mathsf{EH}_q \subseteq \mathsf{H}_q \times \mathbb{F} \times \mathsf{H}_q$ is the set of labelled edges of the kind $m \xrightarrow{f} n$ with their source node in the heap.

Recall that $\mathbb{\Sigma}$ represents the set of all access paths. We construct program-point specific restriction of $\mathbb{\Sigma}$ denoted $\Sigma_q \subseteq \mathbb{\Sigma}$ for a program point q which contains summarized access paths discovered in the static analysis at q. Overall, we construct H_q, E_q, and Σ_q on a need basis to incorporate the effect of a statement at program point q during the analysis, and do not compute the full range of values in the sets $\mathbb{H}$, $\mathbb{E}$, and $\mathbb{\Sigma}$.

$\mathsf{H}_q, \mathsf{E}_q, \Sigma_q$ at program point $q \in \mathbb{Q}$ are denoted as $\mathsf{Hin}_s, \mathsf{Ein}_s, \Sigma\mathsf{in}_s$ and $\mathsf{Hout}_s, \mathsf{Eout}_s, \Sigma\mathsf{out}_s$ at $q = \mathsf{in}_s$ and $q = \mathsf{out}_s$, respectively.

Recall that $\mathsf{HST}^{\mathsf{cm}}(\cdot)$ and $\mathsf{HAP}^{\mathsf{cm}}(\cdot)$ (Definitions 1 and 3) are extractor functions for a location. HST and HAP in Definitions 8 and 9 below are extractor functions for an access-based node.

Definition 8. *Allocation site of an access-based node.* Each access-based heap node is mapped to its allocation site. $\mathsf{HST} : \mathbb{H} \to \mathbb{S}$. □

Definition 9. *Access paths reaching an access-based node.* $\mathsf{HAP}(n,\mathsf{E})$ returns a set of summarized access paths for access-based node n using the sequences of edges reaching n in edge set $\mathsf{E} \subseteq \mathbb{E}$. A helper function $\mathsf{EAP}(m \xrightarrow{f} n, \mathsf{E})$ returns a set of summarized access paths computed by appending f to each summarized access path σ that reaches m in E. For summarization, the allocation site of n ($\mathsf{HST}(n)$) is passed as an argument to function $\mathsf{Summ}(\cdot)$ which is defined in Section 4.1.

$$\mathsf{HAP}(n,\mathsf{E}) = \{\sigma \mid \sigma \in \mathsf{EAP}(m \xrightarrow{f} n, \mathsf{E}), m \xrightarrow{f} n \in \mathsf{E}\} \cup \{\sigma \mid \sigma \in \mathsf{EAP}(v \to n, \mathsf{E}), v \to n \in \mathsf{E}\}$$
$$\mathsf{EAP}(v \to n, \mathsf{E}) = \{v\}$$
$$\mathsf{EAP}(m \xrightarrow{f} n, \mathsf{E}) = \{\mathsf{Summ}(\sigma, f, \mathsf{HST}(n)) \mid \sigma \in \mathsf{HAP}(m,\mathsf{E})\}$$ □

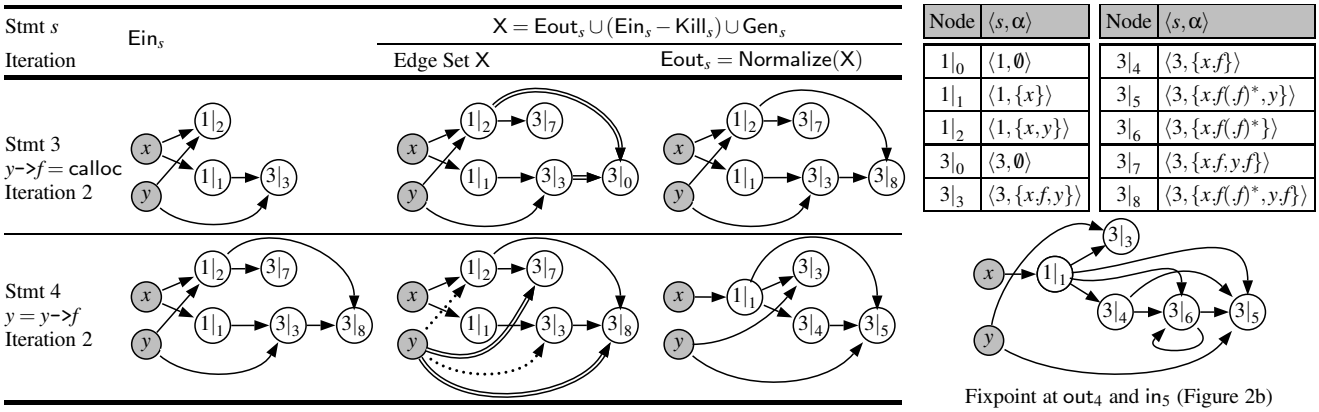


Figure 4: Iteration 2 and fixed point computation at in_5 of access-based analysis for the program in Figure 1. Iteration 1 of the analysis is in Figure 5. Double lined and dotted edges in temporary edge set X denote Gen_s and Kill_s edges, respectively.

3.2 Node Properties Based on Allocation Sites and Accesses

Each access-based node n is uniquely identified by an allocation site $s \in \mathbb{S}$ and a set of access paths $\alpha \subseteq \mathbb{\Sigma}$. By abuse of notation, we use $n \in \mathsf{E}_q$ to mean that some access-based edge in E_q involves access-based node n as the source or the target. For all $q \in \mathbb{Q}$, each access-based node $n \in \mathsf{E}_q$ is uniquely identified by $\langle s, \alpha \rangle$ denoted as node $s|_i$ for brevity i.e., $n \equiv s|_i$. Using Definitions 8 and 9, $s = \mathsf{HST}(n)$, $\alpha = \mathsf{HAP}(n, \mathsf{E}_q)$, and i uniquely identifies α. In other words, the set of access paths α identified by n at q is equal to the set of access paths computed using sequences of edges in E_q that reach n at q. As a special case, $\alpha = \emptyset$ is identified by $i = 0$.

Example 7. Ein_5 denotes a set of access-based edges at program point in_5 (Figure 2b). The allocation sites and access paths of its nodes are tabulated in Figure 2c. $\mathsf{HST}(1|_1)$ is 1 and $\mathsf{HST}(3|_4)$ is 3. $\mathsf{HAP}(1|_1, \mathsf{Ein}_5) = \{x\}$ since $\mathsf{EAP}(x \to 1|_1, \mathsf{Ein}_5)$ is $\{x\}$. $\mathsf{HAP}(3|_4, \mathsf{Ein}_5) = \{x.f\}$ since $\mathsf{EAP}(1|_1 \xrightarrow{f} 3|_4, \mathsf{Ein}_5)$ is $\{x.f\}$. □

3.3 Access-Based Heap Analysis

Our data flow analysis computes access-based graphs in terms of edge sets Ein_s and Eout_s at in_s and out_s. The node sets at in_s and out_s are $\mathbb{V} \cup \mathsf{Hin}_s$ and $\mathbb{V} \cup \mathsf{Hout}_s$, respectively where Hin_s and Hout_s are derived from the corresponding edge sets as follows: $\forall q \in \mathbb{Q}.(n \in \mathsf{E}_q) \Rightarrow n \in \mathbb{V} \cup \mathsf{H}_q$. The solution of our analysis is the set of access-based edges in the least fixed point computation of the data flow equations 2 and 3 below where start denotes the starting statement of the program, $\mathsf{Pred}(s)$ returns the set of control flow predecessors of statement s in the control flow graph, and Ein_s and Eout_s are initialized to $\emptyset$.

$$\mathsf{Ein}_s = \begin{cases} \emptyset & s = \mathsf{start} \\ \bigcup_{k \in \mathsf{Pred}(s)} \mathsf{Eout}_k & \text{otherwise} \end{cases} \quad (2)$$

$$\mathsf{Eout}_s = \mathsf{Normalize}\big(\mathsf{Eout}_s \cup (\mathsf{Ein}_s - \mathsf{Kill}_s) \cup \mathsf{Gen}_s\big) \quad (3)$$

The accumulation of Eout_s in the right hand side of Equation 3 is explained in Section 4.2. $\mathsf{Normalize}(\cdot)$ is defined in Equation 6.

Sets Kill_s and Gen_s are defined below in Equations 4 and 5 using extractor functions Lptr_s and Rpte_s (Figure 3), which denote the l-value and the r-value of statement s. In Figure 3, access-based node $s|_0$ is used when statement s uses $\mathsf{calloc}(\cdot)$. Node $s|_0$ denotes free heap locations unreachable from any access path and allocated at site s. Set Kill_s performs weak updates by removing edges only from the nodes in $\mathbb{V}$ (Section 5.1 performs strong updates on the heap). Set Gen_s contains edges originating either from a variable or in heap depending on the kind of statement.

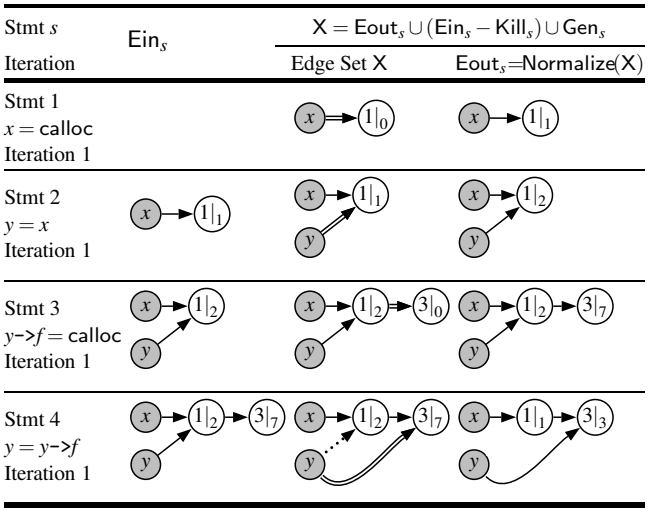


Figure 5: Iteration 1 of access-based analysis for the program in Figure 1. Iteration 2 and fixed point at in_5 is in Figure 4. Double lined and dotted edges in the intermediate edge set X denote Gen_s and Kill_s edges.

$$\mathsf{Kill}_s = \{v \to n \mid v \to n \in \mathsf{Ein}_s, v \in \mathsf{Lptr}_s\} \quad (4)$$

$$\mathsf{Gen}_s = \begin{cases} \mathsf{Lptr}_s \times \mathsf{Rpte}_s & \mathsf{Lfield}_s = \emptyset \\ \mathsf{Lptr}_s \times \mathsf{Lfield}_s \times \mathsf{Rpte}_s & \text{otherwise} \end{cases} \quad (5)$$

Example 8. The row for Stmt 1 in Figure 5 shows that when statement $1 : x = \mathsf{calloc}(\cdot)$ is visited in iteration 1, edge $x \to 1|_0$ (shown with a double lined edge in the column titled "Edge Set X") is generated implying that variable x points to a free heap node for computing Eout_1. The row for Stmt 4 shows that when $4 : y = y\text{->}f$ is visited in iteration 1, edge $y \to 3|_7$ is generated and edge $y \to 1|_2$ (shown with a dotted line in the column titled "Edge Set X") is killed from Ein_4 to compute Eout_4. □

Whenever an edge is killed or generated, the access paths for all nodes reachable from the edge may change thereby requiring an update of the access-based nodes. $\mathsf{Normalize}(\mathsf{X})$, for $\mathsf{X} \subseteq \mathbb{E}$,

updates each node $n \in \mathsf{X}$ with n' where n' uniquely identifies allocation site $\mathsf{HST}(n)$ and updated set of access paths $\mathsf{HAP}(n,\mathsf{X})$.

$$\begin{aligned}\mathsf{Normalize}(\mathsf{X}) = \ & \{m' \xrightarrow{f} n' \mid m \xrightarrow{f} n \in \mathsf{X}, \\ & m' = \langle \mathsf{HST}(m), \mathsf{HAP}(m,\mathsf{X})\rangle, n' = \langle \mathsf{HST}(n), \mathsf{HAP}(n,\mathsf{X})\rangle\} \\ & \cup \{v \rightarrow n' \mid v \rightarrow n \in \mathsf{X}, n' = \langle \mathsf{HST}(n), \mathsf{HAP}(n,\mathsf{X})\rangle\}\end{aligned} \quad (6)$$

Note that access-based nodes are newly created on a need basis by function $\mathsf{Normalize}(\cdot)$. Therefore, in Equation 3, when Eout_s is defined, any node that is newly created is added to Hout_s and its summarized access paths are added to $\Sigma\mathsf{out}_s$.

Example 9. As shown in the row for Stmt 4 in Figure 5, when statement $4: y = y\text{->}f$ is visited in iteration 1, $\mathsf{Normalize}(\mathsf{X})$ is called for $\mathsf{X} = \{x \rightarrow 1|_2, y \rightarrow 3|_7, 1|_2 \xrightarrow{f} 3|_7\}$ shown in the column titled "Edge Set X". Since $\mathsf{HST}(1|_2) = 1$, $\mathsf{HAP}(1|_2,\mathsf{X}) = \{x\}$ and $1|_1 \equiv \langle 1, \{x\}\rangle$, $\mathsf{Normalize}(\mathsf{X})$ updates node $1|_2$ of the column titled "Edge Set X" to $1|_1$ in the column for Eout_s. Also, since $\mathsf{HST}(3|_7) = 3$, $\mathsf{HAP}(3|_7,\mathsf{X}) = \{x.f, y\}$ and $3|_3 \equiv \langle 3, \{x.f, y\}\rangle$, $\mathsf{Normalize}(\mathsf{X})$ updates node $3|_7$ of the column titled "Edge Set X" to $3|_3$ in the column for Eout_s. □

3.4 Soundness and Precision

Assume that all the functions and sets used for access-based abstraction exist for name-based abstraction, and denoted using nm in superscript. Let nodes $l, l' \in \mathbb{H}^{\mathsf{cm}}, a, a' \in \mathbb{H}, n, n' \in \mathbb{H}^{\mathsf{nm}}$. Functions $\mathcal{N}^{\mathsf{c2a}}(l)$ and $\mathcal{N}^{\mathsf{a2n}}(a)$ below respectively map l and a to their corresponding set of access-based and name-based nodes.

$$\mathcal{N}^{\mathsf{c2a}}(l) = \{a \mid \mathsf{HST}^{\mathsf{cm}}(l) = \mathsf{HST}(a) \wedge \mathsf{HAP}^{\mathsf{cm}}(l) \subseteq \mathsf{HAP}(a)\}$$

$$\mathcal{N}^{\mathsf{a2n}}(a) = \{n \mid \mathsf{HST}(a) = \mathsf{HST}^{\mathsf{nm}}(n) \wedge \mathsf{HAP}(a) \subseteq \mathsf{HAP}^{\mathsf{nm}}(n)\}$$

For program point $q = \mathsf{out}_s$, let $\mathsf{Gen}^{\mathsf{cm}}(\pi_q), \mathsf{Gen}_s, \mathsf{Gen}_s^{\mathsf{nm}}$ and $\mathsf{Kill}^{\mathsf{cm}}(\pi_q), \mathsf{Kill}_s, \mathsf{Kill}_s^{\mathsf{nm}}$ denote sets of generated and killed edges.

Theorem 1. *Soundness.* Consider concrete memory edge $e^{\mathsf{cm}} = v \rightarrow l$ or $e^{\mathsf{cm}} = l \xrightarrow{f} l'$, and access-based edge $e = v \rightarrow a$ or $e = a \xrightarrow{f} a'$, respectively, such that $a \in \mathcal{N}^{\mathsf{c2a}}(l)$ and $a' \in \mathcal{N}^{\mathsf{c2a}}(l')$. For $q = \mathsf{out}_s$, access-based abstraction is sound because generation is over-approximated and killing is under-approximated compared to that in the concrete memory.

$$\forall \pi_q.\ \mathsf{E}^{\mathsf{cm}}(\pi_q) \overset{s}{\rightsquigarrow} \mathsf{E}_q \Leftrightarrow (e^{\mathsf{cm}} \in \mathsf{Gen}^{\mathsf{cm}}(\pi_q) \Rightarrow e \in \mathsf{Gen}_s) \wedge (e \in \mathsf{Kill}_s \Rightarrow e^{\mathsf{cm}} \in \mathsf{Kill}^{\mathsf{cm}}(\pi_q))$$ □

Theorem 2. *Precision.* Consider access-based edge $e = v \rightarrow a$ or $e = a \xrightarrow{f} a'$, and name-based edge $e^{\mathsf{nm}} = v \rightarrow n$ or $e^{\mathsf{nm}} = n \xrightarrow{f} n'$, respectively, such that $n \in \mathcal{N}^{\mathsf{a2n}}(a)$ and $n' \in \mathcal{N}^{\mathsf{a2n}}(a')$. For $q = \mathsf{out}_s$, access-based abstraction under-approximates generation and over-approximates killing as compared to the name-based abstraction and hence is more precise.

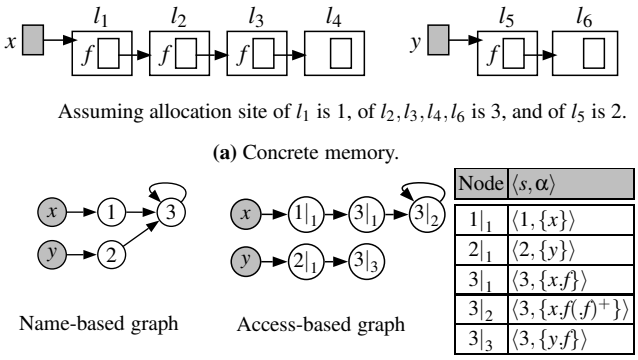
$$\mathsf{E}_q \overset{p}{\rightsquigarrow} \mathsf{E}_q^{\mathsf{nm}} \Leftrightarrow (e \in \mathsf{Gen}_s \Rightarrow e^{\mathsf{nm}} \in \mathsf{Gen}_s^{\mathsf{nm}}) \wedge (e^{\mathsf{nm}} \in \mathsf{Kill}_s^{\mathsf{nm}} \Rightarrow e \in \mathsf{Kill}_s)$$ □

4. Ensuring Convergence

In the presence of loops, we need to handle the following two issues for guaranteeing termination: Bound the potentially infinite access-based nodes into a finite set (Section 4.1) and handle the higher periodicity of flow functions of access-based analysis (Section 4.2).

4.1 Summarization of Access Paths

In the presence of loops, the number of access paths may be infinite in an access-based abstraction. Since an access-based node is identified using sets of access paths, the number of access-based nodes may be infinite. Access-based abstraction uses allocation sites to summarize access paths. This automatically creates a finite number of nodes. We use allocation sites in a novel way for a more precise summarization of access paths as follows: we merge *repeating* fields in the *same* access path that point to locations allocated at the *same* site. In contrast, name-based abstraction groups locations allocated at the *same* site *irrespective* of the field and *irrespective* of the access path.



(b) Summarized access paths for locations allocated at site 3 in name-based graph: $x.f(.f)^*, y.f(.f)^*$. Summarized access paths for locations allocated at site 3 in access-based graph: $x.f, x.f(.f)^+, y.f$.

Figure 6: Contrasting the use of allocation sites for summarization in name-based abstraction and in access-based abstraction.

Example 10. Figure 6 shows that access-based abstraction identifies that access path $y.f$ has no repeating field. On the other hand, name-based abstraction over-approximates $y.f$ to $y.f(.f)^*$ because allocation site 3 of the location reachable by $y.f$ is repeating along another access path, $x.f.f$.

Here (as also in our motivating example in Figure 2), the potentially infinite number of access paths, $x.f.f$, $x.f.f.f$, and so on with repeated occurrences of the same field, are summarized in access-based abstraction as $x.f(.f)^+$ since they all hold the addresses of locations allocated at site 3. The allocation sites are saved in the access paths as follows: $x.f_3.f_3$, $x.f_3.f_3.f_3$, and so on; therefore, summarized as $x.f_3(.f_3)^+$.

An arbitrary access path, $w.g_2.g_3.g_2$, is summarized as $w.g_2(.g_3.g_2)^+$. Here $w.g_2.g_3$ denotes that $w.g$ and $w.g.g$ point to locations allocated at sites 2 and 3, respectively. Summarized access path $w.g_2(.g_3.g_2)^+$ is obtained by representing $w.g_2.g_3.g_2$ as a finite automata and merging the two occurrences of g_2. □

It can be seen that the definition of set of access paths $\Sigma \subseteq \mathbb{V} \times \mathbb{F}^*$ (Section 1) is updated to $\Sigma \subseteq \mathbb{V} \times (\mathbb{F} \times \mathbb{S})^*$ to identify the allocation sites of the locations pointed to by the corresponding field. Function $\mathsf{Summ}(\cdot)$ summarizes each access path as described in the examples above. Its use in Definition 9 is illustrated below.

Example 11. The column for Eout_s in the row for Stmt 3 of Figure 4 shows that node $3|_8 \in \mathsf{Eout}_3$ is reachable by summarized access path, $x.f(.f)^*$. It is obtained when $\mathsf{Normalize}(\mathsf{X})$ updates node $3|_0 \in \mathsf{X}$ of the column titled "Edge Set X" to $3|_8 \in \mathsf{Eout}_3$ in the column for Eout_s due to the generated edges $1|_2 \xrightarrow{f} 3|_0$ in X and $3|_3 \xrightarrow{f} 3|_0$ in X of the column titled "Edge Set X". From Definition 9, set of summarized access paths computed using $\mathsf{EAP}(1|_2 \xrightarrow{f} 3|_0, \mathsf{X})$ is $\{\mathsf{Summ}(x,f,3)\} \cup \{\mathsf{Summ}(y,f,3)\} = \{x.f_3, y.f_3\}$, and that computed using $\mathsf{EAP}(3|_3 \xrightarrow{f} 3|_0, \mathsf{X})$ is $\{\mathsf{Summ}(x.f_3,f,3)\} \cup \{\mathsf{Summ}(y,f,3)\} = \{x.f_3(.f_3)^+, y.f_3\}$ in the column for "Edge Set X". Union of the $\mathsf{EAP}(\cdot)$ values gives

$\mathsf{HAP}(3|_0, \mathsf{X}) = \{x.f_3(.f_3)^*, y.f_3\}$. These access paths are identified by $3|_8 \equiv \langle 3, \{x.f_3(.f_3)^*, y.f_3\}\rangle$ in the column for Eout_s. □

Depending on the precision and efficiency requirements, any user-defined summarization of an access path can be plugged in as the definition of function $\mathsf{Summ}(\cdot)$. We have implemented and evaluated four summarization techniques; their descriptions and our empirical observations can be found in Section 7.

4.2 Handling Higher Periodicity of Flow Functions

Most flow functions have a fixed point which is a periodic point[2] with period $k = 1$. However, some functions have a periodic point with period $k > 1$ [12]. For such a periodic point, the values computed by the function can be equal only with a periodicity of k. Hence detecting convergence requires comparing the values of iteration i with iteration $i - k$, $i > k$; a comparison between successive iterations may lead to non-termination and the value of k may not be known.

For convergence, our analysis needs to maintain a subgraph relationship over the access-based graphs computed at a program point in successive iterations. If this relationship is defined as a subset relationship of node and edge sets where each node is treated as distinct, the graphs in the successive iterations may be found incomparable (Example 12), and may have a periodic point with period $k > 1$ (Example 13). We overcome this problem in the following manner: instead of replacing the data flow value Eout_s with a new computation every time, we accumulate the values in each iteration at the program point. Equation 3 shows that Eout_s is computed by taking a union with Eout_s of the previous iteration in the analysis. Since incomparable graphs are now ruled out, our data flow equations have a fixed point rather than a periodic point with period $k > 1$. This simplifies detecting convergence.

Example 12. Assume that we do not accumulate the values over iterations i.e., node $3|_7$ in Eout_3 of iteration 1 (in the column for Eout_s of the row for Stmt 3 in Figure 5) is not carried over to Eout_3 of iteration 2 (in the column for "Edge Set X" of the row for Stmt 3 in Figure 4). Observe that $3|_7$ is not generated in iteration 2. This is because $3|_0$ in the column titled "Edge Set X" which is updated to $3|_7$ in the column for Eout_s in iteration 1, is updated to $3|_8$ (and not $3|_7$) in the column for Eout_s in iteration 2. Semantically, $3|_7 \equiv \langle 3, \{x.f, y.f\}\rangle$ of iteration 1 is subsumed in $3|_8 \equiv \langle 3, \{x.f(.f)^*, y.f\}\rangle$ of iteration 2 because their sets of access paths have a subset relation. However, node id $3|_7$ in the column for Eout_s of iteration 1 is not found in the column for Eout_s of iteration 2 which makes the graphs incomparable.

By accumulating $3|_7$ from iteration 1 to iteration 2, the graph of iteration 1 is found to be subsumed in the graph of iteration 2. □

Example 13. Appendix A provides an example and its data flow value for the overall flow function with period $k = 2$ when data flow values are not accumulated over iterations. □

5. Extensions of Access-Based Abstraction

In this section, we describe how we allow strong updates (Section 5.1), perform interprocedural analysis (Section 5.2), and handle address expressions and non-pointer fields (Section 5.3).

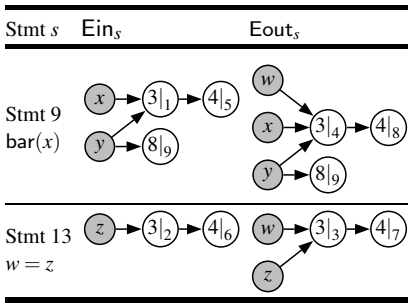
5.1 Strong Updates

Equation 4 performs killing of access-based edges when a variable is redefined. Here we handle strong updates of access-based edges

[2] For a function f, let $f^0(x) = x$ and $f^{i+1}(x) = f(f^i(x))$. A periodic point of f is a value x such that for some k, $f^k(x) = x$ and $f^j(x) \neq x$ for $0 < j < k$. It has a period k because for all $i \geq 0$, $f^{i \cdot k}(x) = x$. A fixed point is a value x such that $f(x) = x$. Hence a fixed point is a periodic point with period $k = 1$.

```
1  void main(){
2    struct abc *x,*y;
3    x=calloc(...);
4    x->f=calloc(...);
5    if ...
6        y=x;
7    else
8        y=calloc(...);
9    bar(x);
10 }
```

```
11 struct abc *w;
12 void bar(struct abc *z){
13   w=z;
14 }
```



Node	$\langle s, \alpha\rangle$
$3\|_1$	$\langle 3, \{x, y\}\rangle$
$3\|_2$	$\langle 3, \{z\}\rangle$
$3\|_3$	$\langle 3, \{w, z\}\rangle$
$3\|_4$	$\langle 3, \{x, y, w\}\rangle$
$4\|_5$	$\langle 4, \{x.f, y.f\}\rangle$
$4\|_6$	$\langle 4, \{z.f\}\rangle$
$4\|_7$	$\langle 4, \{w.f, z.f\}\rangle$
$4\|_8$	$\langle 4, \{x.f, y.f, w.f\}\rangle$
$8\|_9$	$\langle 8, \{y\}\rangle$

Figure 7: Interprocedural analysis using reachability based localization.

when $x\text{->}f$ is defined. A strong update on access-based field edge f can be performed by statement s defining $x\text{->}f$ if both of the following conditions hold at program point $q = \mathsf{in}_s$:

- x should neither point to a summary node nor hold a null or undefined value [14]. We identify a node as a summary node if its set of access paths contains a summarized access path.
- All access paths that are aliases of x, should be must aliases of x. An access path σ is must-aliased to x if whenever an access-based node contains x in its set of access paths, it also contains σ and vice versa.

$$\forall n \in \mathsf{E}_q, \forall \sigma \in \Sigma_q.(x \in \mathsf{HAP}(n, \mathsf{E}_q) \Leftrightarrow \sigma \in \mathsf{HAP}(n, \mathsf{E}_q))$$

5.2 Interprocedural Analysis

We perform a value-based context-sensitive analysis [13, 26], which records the context of a call by a pair of input-output data flow values generated on a need basis by traversing the call graph top-down. Since we summarize access paths, our data flow values, and hence the number of contexts of a procedure, is finite. This enables full context-sensitivity even in the presence of recursion. We perform a reachability-based localization [28] for efficiency which bypasses the information that is not reachable from global variables and callee's parameters. Basically, we use the intuition of separation logic [6] without using their formalizations. The main challenge in our method is that when unreachable information is bypassed, there is a change in the access paths for the nodes passed to the callee. After the call, the bypassed information in the caller needs to be connected with the computation of the callee. For this, we need to map the nodes in the input data flow value of the callee with their corresponding updated nodes in the output data flow value of the callee. This mapping may be different for different calls of the same function. This is different from any other analysis where the mapping of data flow values before and after a call is straightforward.

Example 14. Figure 7 illustrates this bypassing. In the column for Ein_s of the row for Stmt 9 i.e., before the function call $\mathsf{bar}(x)$, nodes, x, y, and $8|_9$—unreachable from the formal parameter z—are bypassed. In other words, they are disconnected from nodes that are reachable from z viz., $3|_1$ and $4|_5$ to obtain the graph in the column

Statements	Smallest set of access paths for nodes for s			
	lt'	lt	rt	rt'
$s: x = y$		$\{x.\overline{\&}\}$	$\{y.\overline{\&}.\overline{\star}\}$	$\{x.\overline{\&}.\overline{\star},\quad y.\overline{\&}.\overline{\star}\}$
$s: x = \&y$		$\{x.\overline{\&}\}$	$\{y.\overline{\&}\}$	$\{x.\overline{\&}.\overline{\star},\quad y.\&\}$
$s: x.f = \&y$	$\{x.\overline{\&}\}$	$\{x.\overline{\&}.f\}$	$\{y.\overline{\&}\}$	$\{x.\overline{\&}.f.\overline{\star},\quad y.\overline{\&}\}$
$s: x.f = y$	$\{x.\overline{\&}\}$	$\{x.\overline{\&}.f\}$	$\{y.\overline{\&}.\overline{\star}\}$	$\{x.\overline{\&}.f.\overline{\star},\quad y.\overline{\&}.\overline{\star}\}$
$s: x = \&(y.f)$		$\{x.\overline{\&}\}$	$\{y.\overline{\&}.f\}$	$\{x.\overline{\&}.\overline{\star},\quad y.\overline{\&}.f\}$
$s: x = y.f$		$\{x.\overline{\&}\}$	$\{y.\overline{\&}.f.\overline{\star}\}$	$\{x.\overline{\&}.\overline{\star},\quad y.\overline{\&}.f.\overline{\star}\}$
$x = \&y$ $s: z = \&(x\text{->}f)$		$\{z.\overline{\&}\}$	$\{x.\overline{\&}.\overline{\star}.f,$ $y.\overline{\&}.f\quad\}$	$\{x.\overline{\&}.\overline{\star}.f,\quad y.\overline{\&}.f,$ $z.\overline{\&}.\overline{\star}\qquad\}$
$x = \mathsf{calloc}(\cdot)$ $s: z = \&(x\text{->}f)$		$\{z.\overline{\&}\}$	$\{x.\overline{\&}.\overline{\star}.f\}$	$\{x.\overline{\&}.\overline{\star}.f,\quad z.\overline{\&}.\overline{\star}\}$

Figure 8: Examples of access-based nodes lt', lt, rt, and rt' for handling the addressof operator & and structure fields. For each statement s, $lt \xrightarrow{\overline{\star}} rt \in \mathsf{Gen}_s$ and $lt \xrightarrow{\overline{\star}} rt' \in \mathsf{Eout}_s$, where rt' identifies updated access paths for rt.

for Ein_s of the row for Stmt 13. At the end of function $\mathsf{bar}(\cdot)$, since nodes $3|_1$ and $4|_5$ have been updated to $3|_3$ and $4|_7$, respectively in the column for Eout_s of the row for Stmt 13, the bypassed nodes need to be reconnected to the latter nodes. This reconnection after the call i.e., at out_9 produces Eout_9 in the row for Stmt 9. □

5.3 Address Expressions and Non-Pointer Fields

We extend our principle of partitioning heap locations based on access paths to partitioning locations of address escaped variables across execution paths. This allows us to precisely compute that variables x and y are not aliased at the end of the code: $\mathsf{if}(\cdot)\ x = \&z;\ \mathsf{else}\ y = \&z;$. This is because the access paths reaching location z are different along the two control flow paths in the code. Address escaped variables are modeled as heap. Thus, strong updates are performed in the lines of heap (Section 5.1).

Access-based abstraction of locations of the same variable requires us to handle the addresses accessed by the addressof operator "&" and accesses such as $x.f.g$ where x and f are not pointers (g may or may not be a pointer). Practical C code (e.g. SPEC benchmarks) is replete with address expressions such as $\&(x.f.g)$. These two features are related and handled as follows:

1. For every variable x, we create an access-based node reachable by a special access path $x.\overline{\&}$.
2. For every field f, we create a field node representing the location reached by adding the offset of f to the corresponding base location represented by $n \in \mathsf{E}_q$. The field node is identified by the set of access paths, $\{\sigma.f \mid \sigma \in \mathsf{HAP}(n, \mathsf{E}_q)\}$.
3. For every address expression $\&(e)$, we create an access-based node identifying access paths corresponding to expression e.
4. The access-based edges (Section 3.1) at program point q are generalized to $\mathsf{E}_q = \mathsf{N}_q \times \mathbb{F} \times \mathsf{N}_q$. Here N_q represents a set of access-based nodes and E_q represents:
   - *Field edges*: Each field node can be reached from the node, representing the base of the corresponding location, via a field edge labelled with the field offset. Field edges are represented by $\mathsf{N}_q \times \mathbb{F} \times \mathsf{N}_q$.
   - *Points-to edges*: A node can hold the address of other nodes. Such edges are represented by $\mathsf{N}_q \times \{\overline{\star}\} \times \mathsf{N}_q$, where $\overline{\star} \in \mathbb{F}$ models the indirection operator "$*$" or "->" (because $x\text{->}f$ is equivalent to $(*x).f$).

Example 15. For each statement s in Figure 8, we create access-based nodes lt (for the l-value) and rt (for the r-value) and generate

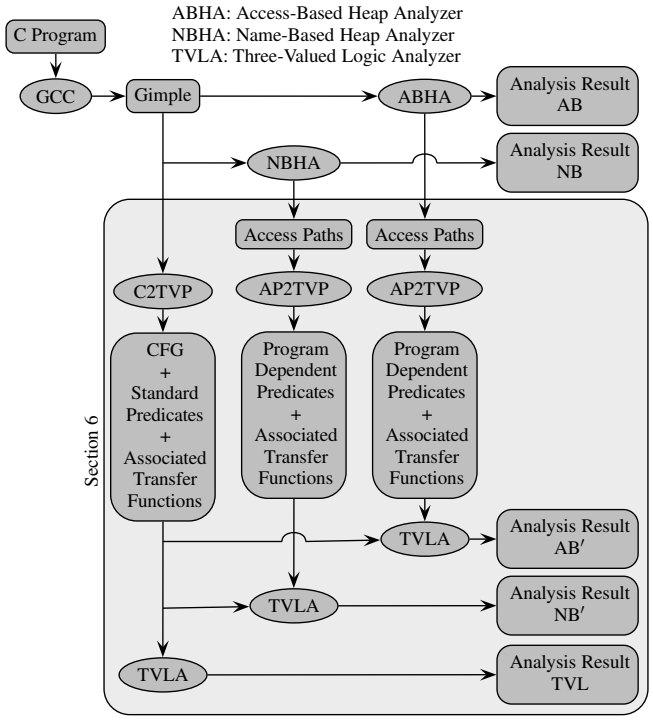


Figure 9: Using tools, ABHA and NBHA, both independently and also as front-ends to TVLA for automatically generating inputs required by TVLA. The circular nodes denote tools, the rectangular nodes denote information, and the edges denote flow of information. Our tool C2TVP converts Gimple (of a C program) to TVP (Three-Valued Program). Our tool AP2TVP constructs predicates and transfer functions (of TVP) from access paths.

$lt \xrightarrow{\overline{\star}} rt$. This changes the access paths for rt. $\mathsf{Normalize}(\cdot)$ creates a new node rt' and adds $lt \xrightarrow{\overline{\star}} rt'$ in Eout_s. $x.f = \&y$ adds $lt' \xrightarrow{f} lt$ also in Eout_s where lt' denotes a node with the access path $x.\overline{\&}$. In each case, since lt, lt', rt, and rt' nodes may be reachable by additional access paths, we could have multiple choices for these nodes. □

At the end of the code, $\mathsf{if}(\cdot)\ x = \&z;\ \mathsf{else}\ y = \&z;$, access-based nodes are uniquely identified by the following set of access paths: $\{x.\overline{\&}\}, \{y.\overline{\&}\}, \{x.\overline{\&}.\overline{\star}, z.\overline{\&}\}$, and $\{y.\overline{\&}.\overline{\star}, z.\overline{\&}\}$ which are precise—x and y are not marked as aliases.

6. ABHA as a Front-end to TVLA

ABHA (Access-Based Heap Analyzer) can also act as front-end to a well known tool, TVLA (Three-Valued Logic Analyzer) [29] which is a parametric shape analyzer. ABHA can export its analysis predicates to TVLA in order to automatically parametrize it with predicates that capture the program behaviour more accurately. Here we also translate C to TVP or Three Valued Program, which includes the control flow graph, predicates, and transfer functions (called *predicate update formulae* [29]).

TVLA is parameterized by predicates viz. *(i) core predicates*, and *(ii) instrumentation predicates* (derived from the core predicates) like "reachable from a variable via a field name" and "is a shared node". Many routinely used core and instrumentation predicates, which we call *standard predicates*, do not capture the program behaviour as accurately as would be desired in some applications (see Section 1). Unfortunately, TVLA cannot itself infer predicates. However, it allows a user to write better instrumentation predicates for program dependent and domain specific results. We call such instrumentation predicates as *program-dependent predicates*. However, for every different

program and also for every different set of properties that are desired to be captured, a user needs to rewrite or fine tune these predicates. For example, for writing program-dependent predicates like those based on access paths, a user needs to know all patterns of sequences of field names that could be created by the given program. Further, the user needs to define a transfer function for each such predicate on each type of statement of the program which is non-trivial specially for real world programs with several thousand lines of code. This impedes the applicability of TVLA.

In contrast to TVLA, ABHA covers the entire spectrum of programs and unravels the program behaviour by automatically generating access paths created by the given program.

As a TVLA front-end, ABHA generates a set of access paths created by the program. Our tool AP2TVP (Figure 9) creates a predicate and a transfer function for each such access path without user intervention. Therefore, the use of ABHA as a front-end to TVLA facilitates an automation of TVLA. In a similar way, as shown in the figure, classical NBHA (name-based heap analyzer) can also be set up as a front-end to TVLA. Overall, the figure shows that for large real world programs, the results of ABHA (or NBHA) can be directly used (denoted as AB (or NB) in the figure). For small programs fitting the TVLA limit, either *(i)* TVLA could be used with standard predicates which may not capture the program behaviour accurately (denoted as TVL), or *(ii)* the results of ABHA (or NBHA) could be exported to TVLA to obtain more precise results automatically (denoted as AB$'$ (or NB$'$)) thereby reducing the human effort in using TVLA productively.

7. Implementation and Measurements

We have implemented fully flow- and context-sensitive ABHA with different summarization techniques and classical NBHA using the LTO framework of GCC 4.7.2 using 32 bit Ubuntu 12.04 on a single Intel Core(TM) i7-3770 CPU at 3.40GHz and 8 GiB RAM. ABHA scales up to 20 kLoC and we have performed measurements for all C based SPEC CPU 2006 benchmarks up to 20.6 kLoC that use heap. Rest of the benchmarks in SPEC CPU 2006 are larger, do not use heap, or include Fortran/C++ files. We have also performed measurements for all heap related SV-COMP benchmarks.

7.1 Language Features

NBHA and ABHA handle the advanced features of C in the usual manner: Pointees of function pointers are computed on-the-fly to identify indirect calls. An array is treated as a single variable without distinguishing between its indices. This allows us to ignore pointer arithmetic on arrays. Pointer arithmetic on structure fields will be handled when we strengthen our implementation.

7.2 Efficient Representation and Analysis Algorithms

Other than reachability based localization (see Section 5.2), we have used the following optimizations for efficiency.

- Only the set of access-based nodes are saved at each program point; edges are saved globally. The graph at each program point is a vertex induced graph. Thus, our fixpoint identification and join operation are simple operations on sets of nodes rather than complex operations of graphs. Saving edges globally helps in merging common subgraphs across program points as is also achieved by the use of isomorphism [22] in TVLA which is a complex operation and applied only at join program points.
- Optimized $\mathsf{Normalize}(\cdot)$ (Section 3.3) updates only the portion of the graph affected by a flow function i.e., it operates in a local way like separation logic [5] whereas TVLA operates in a global way by updating predicates of all nodes in a graph [5].
- Points-to information is computed only for live pointers [14].

SPEC	Benchmark details					
	kLoC	Funcs	Blocks	Stmts	Allocs	Vars
lbm	0.9	19	229	326	1	311
mcf	1.6	23	461	457	3	231
libquantum	2.6	76	917	139	7	386
bzip2	5.7	83	2417	1379	5	637
milc	9.5	184	3677	1330	39	1343
sjeng*	10.5	121	5106	372	11	608
hmmer	20.6	254	6730	4880	11	3386

Figure 10: The table shows the number of lines of code, functions, blocks, pointer assignments, allocs, and variables in SPEC CPU 2006 benchmarks.

7.3 Variants Implemented

Our baseline analyzer is NBHA in which all concrete memory locations created by the same allocation site are represented by a unique node. We compare NBHA with variants of ABHA that use allocation sites and different definitions of $\mathsf{Summ}(\cdot)$ (Section 4.1) to summarize access paths in terms of *(i)* finite automata (**abfa**), *(ii)* limiting repeating fields (**abrf**), *(iii)* names of the variables (**abnv**), and *(iv)* k length prefix (**abkp**) for $0 < k \leq 4$. Performing a sound strong update in NBHA is not straightforward. For a fair comparison, we do not perform strong updates in ABHA as well.

We have developed C2TVP that converts Java like statements of C to TVP (Section 6). A more detailed comparison with TVLA awaits a bug fix in handling function calls in TVLA. Our experiments reveal that the interprocedural example available with TVLA suite [21] currently gives an unsound result which has been confirmed by its authors. Since our interest is in real life large programs, this bug is a show stopper for a comparison with TVLA.

7.4 Metric for Precision

Since variables have compile time names, the number of points-to pairs is a fair measure of precision. However, heap locations do not have (natural) compile-time names and are accessed in terms of access paths. Therefore, the precision of a sound points-to analysis for heap memory can be measured in terms of the number of aliased access paths—for a sound analysis, the smaller this number, the more precise is the analysis.

Since different variants summarize access paths differently, a fair comparison of the invariants requires retrieving access paths of the same length from a given summarization—we have chosen to retrieve access paths of length 4 (variable followed by up to 3 field names); longer access paths did not lead to any new observation.

We record the following three measures of aliases: unique number of pairs of aliased access paths across the program (**unique**), total number of pairs of all the aliased access paths at each program point (**total**), and total number of pairs of aliases of dereferenced access paths (**deref**) at each dereferencing statement.

7.5 Empirical Observations

We compare precision and efficiency of ABHA with baseline NBHA on SPEC CPU 2006 benchmarks (Figures 10 and 11) and heap manipulating SV-COMP benchmarks (Figure 12).

Since all variants of ABHA (Section 7.3) use allocation site also for representing heap nodes, they are at least as precise as NBHA. Figure 11 shows the improved precision on SPEC CPU 2006 in terms of the number of **deref** aliases. Benchmarks bzip2 (5.7 kLoC) and hmmer (20.6 kLoC) perform heavy heap manipulating operations and reflect a significant precision improvement of 70% and 99.66%, respectively using **abfa** summarization technique of ABHA. The variant **abfa** gives the most precise results on all benchmarks except on sjeng, on which **abkp** variant with $k = 3$ and $k = 4$ give better precision within comparable time. The variant **abnv** gives comparable or lower precision on all benchmarks as compared to all other access-based

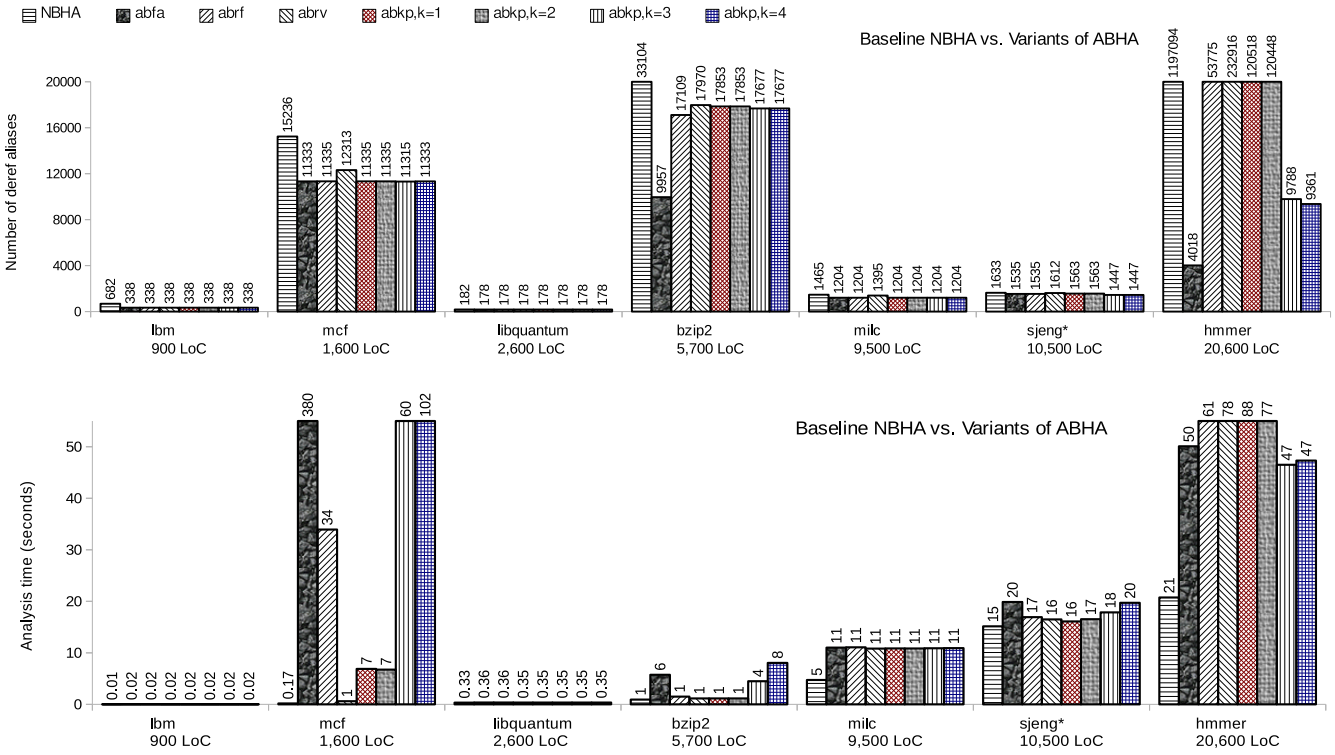


Figure 11: The two bar charts, respectively, depict a comparison of precision (in terms of number of **deref** aliases) and analysis time (in seconds) of baseline NBHA and 7 variants of ABHA on SPEC CPU 2006 benchmarks. Summarization techniques used in ABHA: *(i)* finite automata (**abfa**), *(ii)* limiting repeating fields (**abrf**), *(iii)* names of the variables (**abnv**), and *(iv)* k length prefix (**abkp**) for $0 < k \leq 4$. *In benchmark sjeng, without changing the semantics, we have replaced an array with a calloc because we want to compare heap manipulations rather than array manipulations.

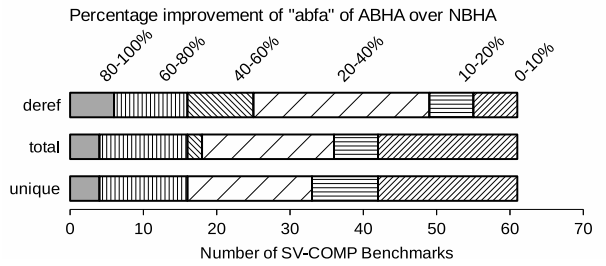


Figure 12: The bar charts depict percentage improvement in precision over NBHA calculated in terms of **unique**, **total**, and **deref** aliases for **abfa** summarization technique in ABHA on 61 SV-COMP benchmarks.

summarization techniques. However, it also takes less time on all benchmarks except hmmer. It is interesting to note that increased precision may not always come at the cost of efficiency. For example, **abfa** variant is both more precise and more efficient as compared to **abrf**, **abnv**, and **abkp** ($k = 1, 2$) variants on hmmer. This is because the memory allocation in hmmer is confined to 11 allocation sites and hence the locations are used in the program through long access paths. Thus, the impact on time of an imprecise summarization of access paths (like **abrf**, **abnv**, and **abkp** ($k = 1, 2$) variants) is worse in hmmer as compared to that in other benchmarks. In Normalize$(\cdot)$ (Section 3.3), the number of nodes affected per program point by each flow function is as large as 13.22 using **abrf**, 50.62 using **abnv**, and only 7.60 using **abfa**. Thus, the analysis time using **abrf** is 61 seconds, **abnv** is 78 seconds, and **abfa** is lesser i.e., 50 seconds.

Figure 12 shows the improved precision using **abfa** variant in percentages for 61 heap-manipulating SV-COMP benchmarks of 27–153 LoC. We have created 6 buckets of percentage improvements (e.g. 0–10%, 10–20%, etc.). The benchmarks are assigned to these buckets based on their percentage improvement. For measures **unique** and **total**, the improvement for majority of benchmarks is less than 20%. However, for the **deref** measure, we record higher improvement in the range of 20%–40%.

Although ABHA is inefficient compared to NBHA (Figure 11), it is certainly far more efficient than other methods such as TVLA (Section 8) that do not scale beyond a thousand lines whereas our implementation scales to 20 kLoC.

8. Related Work

We present the directly relevant literature here; a detailed survey of heap abstractions in static analysis can be found elsewhere [10].

Name-based abstraction is the most commonly used classical technique of bounding heap memory for static analysis [4, 7, 9, 24]. The names resulting from the same allocation site in a procedure when it is called from different calling contexts, can be distinguished using call chains reaching the procedure or by using creation sites of the receiver objects [18, 24, 30]. This requires special machinery. According to us, this distinction becomes unnecessary in an access-based abstraction at program point u because a statement associated with u will have the same effect if two locations allocated similarly are also accessed alike in every execution instance of u.

Access paths have been used in past [17, 23, 31] for variables [8] and for heap locations by randomly choosing one of the access paths reaching the location [17] by summarizing access paths using k-limiting. k-limiting has been found to be inefficient for large values of k [29]. We summarize only an unbounded number of repetitions of a field similar to access graphs [11] and not a bounded

repetition of a field [23]. Sets of access paths have been used to perform must-points-to analysis [31]. Unlike our may-points-to analysis, it is scalable because it involves an intersection of data flow values across control flow paths (the data flow values cannot grow without bound). This creates shorter bounded access paths that do not need summarization.

Empirical measurements of TVLA implementations (using simple instrumentation predicates for trees and singly linked lists [2, 19, 20, 27, 28]) show that they do not scale to real world programs, even after applying optimizations specialized for trees and singly linked lists. These techniques require times varying from a few seconds to up to ten hours on small programs of less than hundred lines of code. Variants of TVLA using core predicates and additional counting quantifiers and even without performing materialization [1] require time in orders of seconds on small handwritten programs. Further the TVLA tool requires human intervention in defining predicates and transfer functions.

Interprocedural implementations of heap abstractions [6, 32] which apply separation logic using TVLA's instrumentation predicates take some seconds to some hours on handwritten programs and small examples of less than thousand lines of code.

The use of higher order logics [3, 15, 25] is highly precise but requires human intervention for deriving non-trivial properties.

In contrast, our tool ABHA scales to programs as large as 20 kLoC without human intervention.

9. Conclusions and Future Work

Starting from a runtime ideal of keeping all concrete locations distinct, we optimize the distinctions that need to be made at compile time without much loss of precision—some loss is inevitable due to summarization. Our heuristic attempts to strike a balance between the two by *(i)* avoiding distinctions that a program cannot make based on the accesses in its statements because these distinctions do not add value to a static analysis, and *(ii)* making those distinctions that may be meaningful for the precision of a static analysis. Hence, our access-based abstraction groups concrete memory locations at program point q *if and only if* they have been allocated at the same site and are accessed alike at q via summarized access paths. The *if* condition combines similarly accessed memory locations thereby reducing the amount of information. The *only if* condition preserves precision by distinguishing between differently accessed locations.

Provably, our tool ABHA is strictly more precise than classical NBHA except in some trivial cases, and practically can improve the precision even up to 99% (in terms of the number of aliases), and scales to 20 kLoC. In terms of efficiency, as expected, the increased precision of ABHA results in being slower than classical NBHA; however, it scales much better than TVLA [29]. Further, we can use ABHA as a front-end to TVLA in order to automatically parametrize it with predicates that capture the program behaviour more accurately. A more detailed comparison with TVLA for real world programs awaits a bug fix (confirmed by its authors) in the current TVLA suite [21].

A. Higher Periodicity of Flow Functions in Access-Based Analysis

Here we illustrate the existence of higher periodicity of functions in access-based analysis. They warrant accumulation of access-based graphs at a program point across iterations for detecting convergence (Section 4.2).

Example 16. The overall function computing the fixed point solution of the program in Figure 13 has a period $k = 2$. The access-based graph Eout_{12} of iteration 1 of the analysis

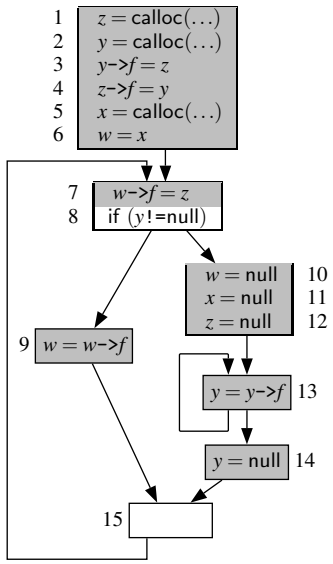
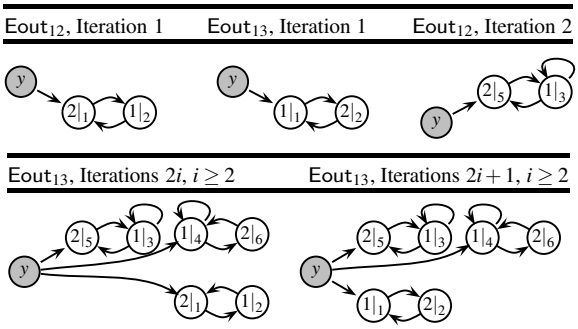


Figure 13: Example to illustrate an overall function for fixed point computation with period $k = 2$ in access-based analysis when values are not accumulated over iterations. We do not analyze any condition statement.



Access-based nodes for site 2		Access-based nodes for site 1	
$2\|_1$	$\langle 2, \{y.(f.f)^*\}\rangle$	$1\|_1$	$\langle 1, \{y.(f.f)^*\}\rangle$
$2\|_2$	$\langle 2, \{y.f.(f.f)^*\}\rangle$	$1\|_2$	$\langle 1, \{y.f.(f.f)^*\}\rangle$
$2\|_5$	$\langle 2, \{y.(f.(f)^*.f)^*\}\rangle$	$1\|_3$	$\langle 1, \{y.f.f^*,\ y.f.(f^*.f.f)^*\}\rangle$
$2\|_6$	$\langle 2, \{y.f^*.f,\ y.f.(f.(f)^*.f)^*\}\rangle$	$1\|_4$	$\langle 1, \{y.f^*,\ y.(f.f.(f)^*)^*\}\rangle$

Figure 14: Access-based graphs for the program in Figure 13 in access-based analysis without accumulating data flow values over iterations.

(Figure 14) reaches the start of the loop on statement 13 because of the complex program structure before the loop (Figure 13). In iteration 2, this graph does not reach out_{12}. Instead, the access-based graph Eout_{12} of iteration 2 of the analysis (Figure 14) reaches the start of the loop of statement 13. Each iteration of the analysis over the loop on statement 13 advances the pointer y by field f in the graph. This changes the access paths and requires creation of new access-based nodes. Node $2|_1$ is created for the access paths obtained by replacing $y.f$ by y in the access paths for node $2|_2$. Similarly node $1|_1$ is created for the access paths obtained by replacing $y.f$ by y in the access paths for node $1|_2$. Node $2|_5$ is created for the access paths obtained by replacing $y.f$ by y in the access paths for node $2|_6$.

As the analysis proceeds, no two successive iterations have identical values. We overcome this problem by accumulating the access-based graphs. When this is done, the larger graph with all eight nodes becomes the fixed point of the analysis. □

Acknowledgments

Supratik Chakraborty gave us valuable insights into TVLA and suggested the use of ABHA as a front-end for TVLA. Alan Mycroft helped us to make our key idea more accessible. The profiling tool developed by Amey Karkare and Prasanna Kumar helped us to improve the efficiency of our implementation dramatically. We thank the anonymous referees for their valuable suggestions for improving the paper.

References

[1] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping tvla: Making parametric shape analysis competitive. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 221–225, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] M. Bozga, R. Iosif, and Y. Lakhnech. *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings*, chapter On Logics of Aliasing, pages 344–360. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[4] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[5] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 240–260, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[8] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[9] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, Nov. 2002.

[10] V. Kanvar and U. P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, June 2016.

[11] U. Khedker, A. Sanyal, and A. Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), Nov. 2007.

[12] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, USA, 1st edition, 2009.

[13] U. P. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 213–228, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] U. P. Khedker, A. Mycroft, and P. S. Rawat. Liveness-based pointer analysis. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 265–282, Deauville, France, 2012. Springer-Verlag.

[15] V. Kuncak, P. Lam, K. Zee, and M. C. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Trans. Softw. Eng.*, 32 (12):988–1005, Dec. 2006.

[16] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[17] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 24–31, New York, NY, USA, 1988. ACM.

[18] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[19] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 547–561, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] A. Loginov, T. Reps, and M. Sagiv. Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 261–279, Berlin, Heidelberg, 2006. Springer-Verlag.

[21] R. Manevich. Tvla: 3-valued logic analysis engine, tvla3+hedec, June 2011. URL `http://www.cs.tau.ac.il/ tvla/tvla-3.tar.gz`.

[22] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. *Partially Disjunctive Heap Abstraction*, pages 265–279. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[23] I. Matosevic and T. S. Abdelrahman. Efficient bottom-up heap analysis for symbolic path-based data access summaries. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 252–263, New York, NY, USA, 2012. ACM.

[24] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. *SIGSOFT Softw. Eng. Notes*, 27(4):1–11, July 2002.

[25] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 221–231, New York, NY, USA, 2001. ACM.

[26] R. Padhye and U. P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM.

[27] J. Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.

[28] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 284–302, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[30] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[31] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Aliasing in object-oriented programming. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Alias Analysis for Object-oriented Programs*, pages 196–232. Springer-Verlag, Berlin, Heidelberg, 2013.

[32] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 25–34, NY, USA, 2004. ACM.