Fluid Updates: Beyond Strong vs. Weak Updates ⋆

Isil Dillig ⋆⋆ Thomas Dillig ⋆⋆⋆ Alex Aiken
{isil, tdillig, aiken}@cs.stanford.edu

Department of Computer Science, Stanford University

Abstract. We describe a *symbolic heap abstraction* that unifies reasoning about arrays, pointers, and scalars, and we define a *fluid update* operation on this symbolic heap that relaxes the dichotomy between strong and weak updates. Our technique is fully automatic, does not suffer from the kind of state-space explosion problem partition-based approaches are prone to, and can naturally express properties that hold for non-contiguous array elements. We demonstrate the effectiveness of this technique by evaluating it on challenging array benchmarks and by automatically verifying buffer accesses and dereferences in five Unix Coreutils applications with no annotations or false alarms.

1 Introduction

In existing work on pointer and shape analysis, there is a fundamental distinction between two kinds of updates to memory locations: *weak updates* and *strong updates* [1–4]. A strong update overwrites the old content of an abstract memory location l with a new value, whereas a weak update adds new values to the existing set of values associated with l. Whenever safe, it is preferable to apply strong updates to achieve better precision.

Applying strong updates to abstract location l requires that l correspond to exactly one concrete location. This requirement poses a difficulty for applying strong updates to (potentially) unbounded data structures, such as arrays and lists, since the number of elements may be unknown at analysis time. Many techniques combine all elements of an unbounded data structure into a single *summary location* and only allow weak updates [2, 5, 6]. More sophisticated techniques, such as analyses based on 3-valued logic [3], first isolate individual elements of an unbounded data structure via a *focus* operation to apply a strong update, and the isolated element is folded back into the summary location via a dual *blur* operation to avoid creating an unbounded number of locations. While such an approach allows precise reasoning about unbounded data structures, finding the right focus and blur strategies can be challenging and hard to automate [3].

In this paper, we propose a way of relaxing the dichotomy between applying weak vs. strong updates to a particular kind of unbounded data structure, arrays,

⋆ This work was supported by grants from NSF (CNS-050955 and CCF-0430378) with additional support from DARPA.
⋆⋆ Supported by a Stanford Graduate Fellowship
⋆⋆⋆ Supported by a Siebel Fellowship

by introducing *fluid updates*. Fluid updates can always be safely applied regardless of whether a given abstract memory location represents a single concrete location or an array. Three key ideas underpin fluid updates:

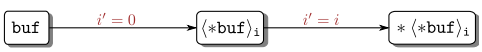
1. Arrays are modeled as abstract locations qualified by *index variables*; constraints on index variables specify which concrete elements are referred to by a points-to edge.
2. In general, we may not know the exact subset of concrete elements updated by a statement. To deal with this uncertainty, each points-to edge is qualified by a pair of constraints $\langle \phi_{NC}, \phi_{SC} \rangle$, called *bracketing constraints*, over- and underapproximating the subset of concrete elements selected by this edge.
3. To apply a fluid update, we compute a bracketing constraint $\langle \phi_{NC}, \phi_{SC} \rangle$ representing over- and underapproximations for the set of concrete elements updated by a statement. A fluid update preserves all existing points-to edges under the negation of the update condition, i.e., $\neg\langle \phi_{NC}, \phi_{SC} \rangle = \langle \neg\phi_{SC}, \neg\phi_{NC} \rangle$, while applying the update under $\langle \phi_{NC}, \phi_{SC} \rangle$.

An important property of bracketing constraints is that the intersection of a bracketing constraint B and its negation $\neg B$ is not necessarily empty (see Section 2.1). For array elements in the intersection, both the new value is added and the old values are retained—i.e., a weak update is performed. Because fluid updates rely on negation, having both over- and underapproximations (or equivalently, necessary and sufficient conditions) is crucial for the correctness of our approach.

If the concrete elements updated by a statement s are known exactly, i.e., ϕ_{NC} and ϕ_{SC} are the same, the fluid update represents a strong update to some set of elements in the array. On the other hand, if nothing is known about the update condition, i.e., $\langle \phi_{NC}, \phi_{SC} \rangle = \langle \mathit{true}, \mathit{false} \rangle$, the fluid update is equivalent to a weak update to all elements in the array. Otherwise, if only partial information is available about the concrete elements modified by s, the fluid update encodes this partial information soundly and precisely. Consider the following example:

```
void send_packets(struct packet** buf, int c, int size) {
  assert(2*c <= size);
  for(int j=0; j< 2*c; j+=2)
    if(transmit_packet(buf[j]) == SUCCESS) { free(buf[j]); buf[j] = NULL; }
}
```

The function `send_packets` takes an array `buf` of `packet`$*$'s, an integer `c` representing the number of high-priority packets to be sent, and an integer `size`, denoting the number of elements in `buf`. All even indices in `buf` correspond to high-priority packets whereas all odd indices are low-priority.[1] This function submits one high-priority packet at a time; if the transfer is successful (which may depend on network traffic), it sets the corresponding element in `buf` to `NULL` to indicate the packet has been processed.



[1] The distinction between even and odd-numbered elements in a network buffer arises in many real network applications, for example in packet scheduling [7] and p2p video streaming [8] .

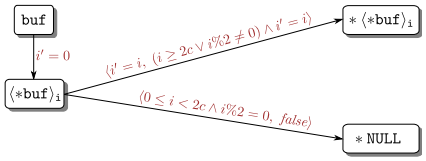


Fig. 1. The points-to graph at the end of function `send_packets`

The figure above shows the *symbolic heap abstraction* at the entry of `send_packets`. Here, nodes represent abstract locations named by *access paths* [9], and edges denote points-to relations. Because either the source or target of an edge may be a set, we write constraints on edges to indicate which elements of the source point to which elements of the target. In the figure, the dereference of `buf` is an array, hence, it is qualified by an *index variable* i; the location named $\langle *\texttt{buf}\rangle_{\texttt{i}}$ represents all elements of array $*\texttt{buf}$. By convention, primed index variables on an edge qualify the edge's target, and unprimed index variables qualify the source. If the over- and underapproximations on an edge are the same, we write a single constraint instead of a pair. In this graph, the edge from `buf` to $\langle *\texttt{buf}\rangle_{\texttt{i}}$ is qualified by $i' = 0$ because `buf` points to the first element of array $\langle *\texttt{buf}\rangle_{\texttt{i}}$. The constraint $i = i'$ on the edge from $\langle *\texttt{buf}\rangle_{\texttt{i}}$ to $*\langle *\texttt{buf}\rangle_{\texttt{i}}$ indicates that the i'th element of array $*\texttt{buf}$ points to some corresponding target called $*\langle *\texttt{buf}\rangle_{\texttt{i}}$.

The concrete elements modified by the statement $\texttt{buf}[\texttt{j}] = \texttt{NULL}$ cannot be specified exactly at analysis time since the success of `transmit_packet` depends on an environment choice (i.e., network state). The loop may, but does not have to, set all even elements between 0 and $2c$ to `NULL`. Hence, the best overapproximation of the indices of $*\texttt{buf}$ modified by this statement is $0 \leq i < 2c \ \wedge \ i\%2 = 0$. On the other hand, the best underapproximation of the set of indices updated in the loop is the empty set (indicated by the constraint *false*) since no element is guaranteed to be updated by the statement $\texttt{buf}[\texttt{j}] = \texttt{NULL}$.

Figure 1 shows the symbolic heap abstraction at the end of `send_packets`. Since the set of concrete elements that may be updated by $\texttt{buf}[\texttt{j}] = \texttt{NULL}$ is given by $\langle 0 \leq i < 2c \wedge i\%2 = 0,\ \mathit{false}\rangle$, the fluid update adds an edge from $\langle *\texttt{buf}\rangle_{\texttt{i}}$ to $*\texttt{NULL}$ under this bracketing constraint. The existing edge from $\langle *\texttt{buf}\rangle_{\texttt{i}}$ to $*\langle *\texttt{buf}\rangle_{\texttt{i}}$ is preserved under $\neg\langle 0 \leq i < 2c \wedge i\%2 = 0,\ \mathit{false}\rangle$. Now, the complement (negation) of an overapproximation is an underapproximation of the complement; similarly the complement of an underapproximation is an overapproximation of the complement. Thus, assuming $i \geq 0$, this is equivalent to $\langle \mathit{true},\ i \geq 2c \vee i\%2 \neq 0\rangle$. Since the initial constraint on the edge stipulates $i = i'$, the edge constraint after the fluid update becomes $\langle i = i',\ (i \geq 2c \vee i\%2 \neq 0) \wedge i = i'\rangle$. The new edge condition correctly and precisely states that any element of $*\texttt{buf}$ may still point to its original target when the function exits, but only those elements whose index satisfies the constraint $i \geq 2c$ or $i\%2 \neq 0$ *must* point to their original target. As this example illustrates, fluid updates have the following characteristics:

– Fluid updates do not require concretizing individual elements of an array to perform updates, making operations such as focus and blur unnecessary.

– Fluid updates never construct explicit partitions of an array, making this approach less vulnerable to the kind of state space explosion problem that partition-based approaches, such as [3], are prone to.
– Fluid updates preserve partial information despite imprecision and uncertainty. In the above example, although the result of `transmit_packet` is unknown, the analysis can still determine that no odd packet is set to `NULL`.
– Fluid updates separate the problem of determining *which* concrete elements are updated from *how* the update is performed. Fluid updates are oblivious to the precision of the over- and underapproximations, and retain the best possible information with respect to these approximations. In the above example, a less precise overapproximation, such as $0 \leq i < 2c$, would not affect the way updates are performed.

This paper is organized as follows: Section 2 defines a simple language and introduces basic concepts. Section 3 formalizes the symbolic heap abstraction, Section 4 presents the basic pointer and value analysis based on fluid updates, and Section 5 discusses treatment of loops. Section 6 discusses a prototype implementation, Section 7 presents our experimental results, and Section 8 surveys related work. To summarize, this paper makes the following key contributions:

– We introduce *fluid updates* as a viable alternative to the dichotomy between weak vs. strong updates, and we describe an expressive memory analysis based on *symbolic heap abstraction* that unifies reasoning about arrays, pointers, and scalars. (We do not, however, address recursive pointer-based data structures in this paper.)
– We propose *bracketing constraints* to allow a sound negation operation when performing updates in the presence of imprecision and uncertainty.
– We demonstrate our technique is precise and efficient for reasoning about values and points-to targets of array elements. Furthermore, our technique is fully automatic, requiring no annotations or user-provided predicates.
– We show the effectiveness of our approach by verifying the safety of buffer accesses and dereferences fully automatically in five Unix Coreutils applications that manipulate arrays and pointers in intricate ways.

2 Language and Preliminaries

We first define a small imperative language in which we formalize our technique:

$$
\begin{array}{ll}
\textit{Program } P & := F^{+} \\
\textit{Function } F & := \textit{define } f(v_1, \ldots, v_n) = S \\
\textit{Statement } S & := S_1; S_2 \mid v_1 = v_2 \mid v_1 = c \mid v_1 = \textit{alloc}(v_2) \mid v_1 = v_2[v_3] \mid v_2[v_3] = v_1 \\
& \quad \mid v_1 = v_2 \oplus v_3 \mid v_1 = v_2 \text{ intop } v_3 \mid v_1 = v_2 \text{ predop } v_3 \mid \\
& \quad \textit{if } v \neq 0 \textit{ then } S_1 \textit{ else } S_2 \mid \textit{while } v \neq 0 \textit{ do } S \textit{ end}
\end{array}
$$

In this grammar, v is a variable, and c is an integer constant. Types are defined by the grammar $\tau := \text{int} \mid \text{pointer}(\text{array}(\tau))$. Load ($v_1 = v_2[v_3]$) and store ($v_2[v_3] = v_1$) statements are defined on pointers v_2 and integers v_3, and we assume programs are well-typed. $v[i]$ first dereferences v and then selects the i'th element of the array pointed to by v. Pointer arithmetic $v_1 = v_2 \oplus v_3$ makes v_1 point to offset

v_3 in the array pointed to by v_2. Integer operations (intop) include $+, -$, and $\times$. Predicate operators (predop) are $=, \neq$ and $<$, and predicates evaluate to 0 (false) or 1 (true). The *alloc*(v_2) statement allocates an array with v_2 elements.

Appendix A gives an operational semantics of this language. In the concrete semantics, a concrete location l_c is a pair (s, i) where s is a start address for a block of memory and i is an offset from s. An environment E maps program variables to concrete locations, and a store S maps locations to other locations or integer values. Due to space limitations, we omit function calls from our formal discussion; Section 6 discusses how we treat function calls in the implementation.

2.1 Constraint Language

The constraints used in the analysis are defined by:

$$\begin{array}{ll} \textit{Term } T & := c \mid v \mid T_1 \text{ intop } T_2 \mid \textit{select}(T_1, T_2) \mid \textit{deref}(T) \\ \textit{Literal } L & := \textit{true} \mid \textit{false} \mid T_1 \text{ predop } T_2 \mid T\%c = 0 \\ \textit{Atom } A & := L \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \\ \textit{Constraint } C & := \langle A_{NC}, A_{SC} \rangle \end{array}$$

Terms are constants, variables, arithmetic terms, and the uninterpreted function terms *select*(T_1, T_2), and *deref*(T). Terms are used to represent scalars, pointers, and arrays; the uninterpreted function term *select*(T_1, T_2) represents the result of selecting element at index T_2 of array T_1, and the term *deref*(T) represents the result of dereferencing T.

Literals are *true*, *false*, comparisons $(=, \neq, <)$ between two terms, and divisibility checks on terms. Atomic constraints A are arbitrary boolean combinations of literals. Satisfiability and validity of atomic constraints are decided over the combined theory of uninterpreted functions and linear integer arithmetic extended with divisibility (mod) predicates. Bracketing constraints C are pairs of atomic constraints of the form $\langle A_{NC}, A_{SC} \rangle$ representing necessary and sufficient conditions for some fact. A bracketing constraint is well-formed if and only if $A_{SC} \Rightarrow A_{NC}$. We write $\lceil \phi \rceil$ to denote the necessary condition of a bracketing constraint ϕ and $\lfloor \phi \rfloor$ to denote the sufficient condition of ϕ.

Example 1. Consider an edge from location $\langle *\texttt{a} \rangle_{\texttt{i}}$ to $*\texttt{NULL}$ qualified by $\langle 0 \leq i < \textit{size}, 0 \leq i < \textit{size} \rangle$.This constraint expresses that *all* elements of the array with indices between 0 and *size* are `NULL`. Since it is sufficient that i is between 0 and *size* for $\langle *\texttt{a} \rangle_{\texttt{i}}$ to point to $*\texttt{NULL}$, it follows that all elements in this range are `NULL`. On the other hand, if the constraint on the edge is $\langle 0 \leq i < \textit{size}, \textit{false} \rangle$, any element in the array may be `NULL`, but no element must be `NULL`.

Boolean operators $\neg, \wedge$, and $\vee$ on bracketing constraints are defined as:

$$\begin{array}{c} \neg \langle A_{NC}, A_{SC} \rangle = \langle \neg A_{SC}, \neg A_{NC} \rangle \\ \langle A_{NC1}, A_{SC1} \rangle \star \langle A_{NC2}, A_{SC2} \rangle = \langle A_{NC1} \star A_{NC2}, A_{SC1} \star A_{SC2} \rangle \quad (\star \in \{\wedge, \vee\}) \end{array}$$

Since the negation of the overapproximation for some set S is an underapproximation for the complement of S, necessary and sufficient conditions are swapped under negation. The following lemma is easy to show:

Lemma 1. *Bracketing constraints preserve the well-formedness property* $A_{\mathrm{SC}} \Rightarrow A_{\mathrm{NC}}$ *under boolean operations.*

Satisfiability and validity are defined in the following natural way:

$$SAT(\langle A_{NC}, A_{SC}\rangle) \equiv SAT(A_{NC}) \qquad VALID(\langle A_{NC}, A_{SC}\rangle) \equiv VALID(A_{SC})$$

Lemma 2. *Bracketing constraints do not obey the law of the excluded middle and non-contradiction, but they satisfy the following weaker properties:*

$$\mathrm{VALID}(\lceil \langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \vee \neg\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \rceil) \quad \mathrm{UNSAT}(\lfloor \langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \wedge \neg\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \rfloor)$$

Proof. $\lceil \langle A_{NC}, A_{SC}\rangle \vee \neg\langle A_{NC}, A_{SC}\rangle \rceil$ is $(A_{NC} \vee \neg A_{SC}) \Leftrightarrow (A_{SC} \Rightarrow A_{NC}) \Leftrightarrow \mathit{true}$, where the last equivalence follows from well-formedness. Similarly, $\lfloor \langle A_{NC}, A_{SC}\rangle \wedge \neg\langle A_{NC}, A_{SC}\rangle \rfloor$ is $(A_{SC} \wedge \neg A_{NC}) \Leftrightarrow \mathit{false}$, where the last step follows from the well-formedness property.

3 Symbolic Heap Abstraction

Abstract locations are named by *access paths* [9] and defined by the grammar:

$$\mathit{Access\ Path}\ \pi := \mathfrak{L}_v \mid \mathit{alloc}_{id} \mid \langle\pi\rangle_i \mid *\pi \mid c \mid \pi_1 \ \mathrm{intop}\ \pi_2 \mid \top$$

Here, $\mathfrak{L}_v$ denotes the abstract location corresponding to variable v, and alloc_{id} denotes locations allocated at program point *id*. Any array location is represented by an access path $\langle\pi\rangle_i$, where π represents the array and i is an *index variable* ranging over the indices of π (similar to [22]). The location $*\pi$ represents the dereference of π. The access path c denotes constants, $\pi_1 \ \mathrm{intop}\ \pi_2$ represents the result of performing intop on π_1 and π_2, and $\top$ denotes any possible value.

A *memory access path*, denoted π_{mem}, is any access path that does not involve c, $\pi_1 \ \mathrm{intop}\ \pi_2$, and $\top$. We differentiate memory access paths because only locations that are identified by memory access paths may be written to; other kinds of access paths are only used for encoding values of scalars.

Given a concrete store S and an environment E mapping program variables to concrete locations (see Appendix A), a function γ maps abstract memory locations to a set of concrete locations $(s_1, i_1) \ldots (s_k, i_k)$:

$$\begin{array}{ll}
\gamma(E, S, \mathfrak{L}_v) & = \{E(v)\} \\
\gamma(E, S, \mathit{alloc}_{id}) & = \{(l, 0) \mid l \textit{ is the result of allocation at program point id }\} \\
\gamma(E, S, \langle\pi\rangle_i) & = \{(l, \mathit{index}_j) \mid (l, \mathit{index}_j) \in S \ \wedge\ (l, 0) \in \gamma(E, S, \pi))\} \\
\gamma(E, S, *\pi) & = \bigcup_{l_i \in \gamma(E,S,\pi)} S(l_i)
\end{array}$$

Since we will concretize abstract memory locations under a certain assumption about their index variables, we define another function γ_c, similar to γ but qualified by constraint ϕ. The only interesting modification is for $\langle\pi\rangle_i$:

$$\gamma_c(E, S, \langle\pi\rangle_i, \phi) = \{(l, \mathit{index}_j) \mid (l, \mathit{index}_j) \in S \wedge (l, 0) \in \gamma_c(E, S, \pi, \phi) \wedge \mathrm{SAT}(\phi[\mathit{index}_j/i])\}$$

As is standard in points-to graphs, we enforce that for any two memory access paths, either $\pi_{mem} = \pi'_{mem}$ or $\gamma(E, S, \pi_{mem}) \cap \gamma(E, S, \pi'_{mem}) = \emptyset$.

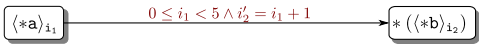
A *symbolic heap abstraction* is a directed graph where nodes denote abstract locations identified by access paths and edges qualified by bracketing constraints denote points-to relations. Since we want to uniformly encode points-to and value information, we extend the notion of points-to relations to scalars. For example, if an integer `a` has value 3, the symbolic heap abstraction contains a "points-to" edge from `a`'s location to some location named *3, thereby encoding that the value of `a` is 3. Hence, the symbolic heap encodes the value of each scalar.

Formally, a symbolic heap abstraction is defined by

$$\Gamma : \pi_{mem} \rightarrow 2^{(\pi,\phi)}$$

mapping a source location to a set of (target location, constraint) pairs. The edge constraint ϕ may constrain program variables to encode the condition under which this points-to relation holds. More interestingly, ϕ may also qualify the source and the target location's index variables, thereby specifying which elements of the source may (and must) point to which elements of the target.

The combination of indexed locations and edge constraints parametric over these index variables makes the symbolic heap abstraction both very expressive but also non-trivial to interpret. In particular, if the source location is an array, we might want to determine the points-to targets of a specific element (or some of the elements) in this array. However, the symbolic heap abstraction does not directly provide this information since edge constraints are parametric over the source and the target's index variables. Consider the following points-to relation:



Suppose we want to know which location(s) the fourth element of array $\langle *a \rangle_{i_1}$ points to. Intuitively, we can determine the target of the fourth element of $\langle *a \rangle_{i_1}$ by substituting the index variable i_1 by value 3 in the edge constraint $0 \leq i_1 < 5 \wedge i'_2 = i_1 + 1$. This would yield $i'_2 = 4$, indicating that the fourth element of $\langle *a \rangle_i$ points to the target of the fifth element of $\langle *b \rangle_{i_2}$.

While a simple substitution allows us to determine the target of a specific array element as in the above example, in general, we need to determine the targets of those array elements whose indices satisfy a certain constraint. Since this constraint may not limit the index variable to a single value, determining points-to targets from an indexed symbolic heap abstraction requires existential quantifier elimination in general. In the above example, we can determine the possible targets of elements of $\langle *a \rangle_{i_1}$ whose indices are in the range $[0, 3]$ (i.e., satisfy the constraint $0 \leq i_1 \leq 3$) by eliminating i_1 from the following formula:

$$\exists i_1 . (0 \leq i_1 \leq 3 \wedge (0 \leq i_1 < 5 \wedge i'_2 = i_1 + 1))$$

This yields $1 \leq i'_2 \leq 4$, indicating that the target's index must lie in the range $[1, 4]$. To formalize this intuition, we define an operation $\phi_1 \downarrow_I \phi_2$, which yields the result of restricting constraint ϕ_1 to only those values of the index variables I that are consistent with ϕ_2.

Definition 1 ($\phi_1 \downarrow_I \phi_2$) Let ϕ_1 be a constraint qualifying a points-to edge and let ϕ_2 be a constraint restricting the values of index variables I. Then,

$$\phi_1 \downarrow_I \phi_2 \equiv \mathit{Eliminate}(\exists I.\ \phi_1 \wedge \phi_2)$$

where the function *Eliminate* performs existential quantifier elimination.

The quantifier elimination performed in this definition is exact because index variables qualifying the source or the target never appear in uninterpreted functions in a valid symbolic heap abstraction; thus the elimination can be performed using [10].

4 Pointer and Value Analysis Using Fluid Updates

In this section, we give deductive rules describing the basic pointer and value analysis using fluid updates. An invariant mapping $\Sigma : \mathit{Var} \rightarrow \pi_{mem}$ maps program variables to abstract locations, and the environment Γ defining the symbolic heap abstraction maps memory access paths to a set of (access path, constraint) pairs. Judgments $\Sigma \vdash \mathtt{a} : \mathfrak{L}_a$ indicate that variable $\mathtt{a}$ has abstract location $\mathfrak{L}_a$, and judgments $\Gamma \vdash_j \pi_s : \langle \pi_{t_j}, \phi_j \rangle$ state that $\langle \pi_{t_j}, \phi_j \rangle \in \Gamma(\pi_s)$. Note that there may be many $\langle \pi_{t_j}, \phi_j \rangle$ pairs in $\Gamma(\pi_s)$, and this form of judgment is used in the rules to refer to each of them without needing to use sets.

We first explain some notation used in Figure 2. The function $U(\phi)$ replaces the primed index variables in constraint ϕ with their unprimed counterparts, e.g., $U(i'_1 = 2)$ is $(i_1 = 2)$; this is necessary when traversing the points-to graph because the target location of an incoming edge becomes the source of the outgoing edge from this location. We use the notation $\Gamma \wedge \phi$ as shorthand for:

$$\Gamma'(\pi) = \{\langle \pi_l, \phi_l \wedge \phi \rangle \mid \langle \pi_l, \phi_l \rangle \in \Gamma(\pi)\}$$

A union operation $\Gamma = \Gamma' \cup \Gamma''$ on symbolic heap abstractions is defined as:

$$\langle \pi', \phi' \vee \phi'' \rangle \in \Gamma(\pi) \ \Leftrightarrow\ \langle \pi', \phi' \rangle \in \Gamma'(\pi) \ \wedge\ \langle \pi', \phi'' \rangle \in \Gamma''(\pi).$$

We write $\mathfrak{I}(\pi)$ to denote the set of all index variables used in π, and we say "i is index of π" if i is the outermost index variable in π.

The basic rules of the pointer and value analysis using fluid updates are presented in Figure 2. We focus mainly on the inference rules involving arrays, since these rules either directly perform fluid updates (Array Store) or rely on the constraint and index-based representation that is key for fluid updates.

We start by explaining the Array Load rule. In this inference rule, each π_{2_j} represents one possible points-to target of v_2 under constraint ϕ_{2_j}. Because π_{2_j} is an array, the constraint ϕ_{2_j} qualifies π_{2_j}'s index variables. Each π_{3_k} represents one possible (scalar) value of v_3. Since we want to access the element at offset v_3 of v_2's target, we select the element at offset v_3 by substituting i' with $i' - \pi_{3_k}$ in the constraint ϕ_{2_j}, which effectively increments the value of i' by π_{3_k}. Now, we need to determine the targets of those elements of π_{2_j} whose indices are consistent with $\phi'_{2_{jk}}$; hence, we compute $\phi_{t_{jl}} \downarrow_{\mathfrak{I}(\pi_{2_j})} \phi'_{2_{jk}}$ (recall Section 3) for each target $\pi_{t_{jl}}$ of π_{2_j}. The following example illustrates this rule.

Assign

$$\frac{\Sigma \vdash v_1 : \mathfrak{L}_{v_1},\ v_2 : \mathfrak{L}_{v_2} \quad \Gamma' = \Gamma[\mathfrak{L}_{v_1} \leftarrow \Gamma(\mathfrak{L}_{v_2})]}{\Sigma, \Gamma \vdash\ v_1 = v_2\ :\ \Gamma'}$$

Array Load

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : \mathfrak{L}_{v_1},\ v_2 : \mathfrak{L}_{v_2}, v_3 : \mathfrak{L}_{v_3} \\ \Gamma \vdash_j \mathfrak{L}_{v_2} : \langle \pi_{2_j}, \phi_{2_j} \rangle \ \ (i \text{ index of } \pi_{2_j}) \\ \Gamma \vdash_k \mathfrak{L}_{v_3} : \langle *\pi_{3_k}, \phi_{3_k} \rangle \\ \Gamma \vdash_l \pi_{2_j} : \langle \pi_{t_{jl}}, \phi_{t_{jl}} \rangle \\ \phi'_{2_{jk}} = U(\phi_{2_j}[i' - \pi_{3_k}/i']) \\ \phi'_{t_{jkl}} = \phi_{t_{jl}} \downarrow_{\mathfrak{I}(\pi_{2_j})} \phi'_{2_{jk}} \\ \Gamma' = \Gamma[\mathfrak{L}_{v_1} \leftarrow (\bigcup_{jkl} \langle \pi_{t_{jl}}, \phi'_{t_{jkl}} \wedge \phi_{3_k} \rangle)]\end{array}}{\Sigma, \Gamma \vdash\ v_1 = v_2[v_3]\ :\ \Gamma'}$$

Pointer Arithmetic

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : \mathfrak{L}_{v_1}, v_2 : \mathfrak{L}_{v_2}, v_3 : \mathfrak{L}_{v_3} \\ \Gamma \vdash_j \mathfrak{L}_{v_2} : \langle \pi_{2_j}, \phi_{2_j} \rangle \\ \Gamma \vdash_k \mathfrak{L}_{v_3} : \langle *\pi_{3_k}, \phi_{3_k} \rangle \\ \phi'_{2_{jk}} = \phi_{2_j}[(i' - \pi_{3_k})/i'] \ \ (i \text{ index of } \pi_{2_j}) \\ \Gamma' = \Gamma[\mathfrak{L}_{v_1} \leftarrow (\bigcup_{jk} \langle \pi_{2_j}, \phi'_{2_{jk}} \wedge \phi_{3_k} \rangle)]\end{array}}{\Sigma, \Gamma \vdash v_1 = v_2 \oplus v_3 : \Gamma'}$$

If Statement

$$\frac{\begin{array}{l}\Sigma \vdash v : \mathfrak{L}_v \\ \Gamma \vdash \mathfrak{L}_v : \{\langle *\mathtt{1}, \phi_{true} \rangle, \langle *\mathtt{0}, \phi_{false} \rangle\} \\ \Sigma, \Gamma \vdash S_1 : \Gamma' \\ \Sigma, \Gamma \vdash S_2 : \Gamma'' \\ \Gamma_T = \Gamma' \wedge \phi_{true} \\ \Gamma_F = \Gamma'' \wedge \phi_{false}\end{array}}{\Sigma, \Gamma \vdash \mathit{if}\ v \neq 0\ \mathit{then}\ S_1\ \mathit{else}\ S_2 : \Gamma_T \cup \Gamma_F}$$

Alloc

$$\frac{\Sigma \vdash v_1 : \mathfrak{L}_{v_1} \quad \Gamma' = \Gamma[\mathfrak{L}_{v_1} \leftarrow \langle alloc_{id} \rangle_i] \wedge i' = 0 \ \ (i \text{ fresh})}{\Sigma, \Gamma \vdash\ v_1 = alloc(v_2)\ :\ \Gamma'}$$

Array Store (Fluid Update)

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : \mathfrak{L}_{v_1},\ v_2 : \mathfrak{L}_{v_2}, v_3 : \mathfrak{L}_{v_3} \\ \Gamma \vdash_j \mathfrak{L}_{v_1} : \langle \pi_{1_j}, \phi_{1_j} \rangle \\ \Gamma \vdash \mathfrak{L}_{v_2} : \{\langle \pi_{2_1}, \phi_{2_1} \rangle \ldots \langle \pi_{2_n}, \phi_{2_n} \rangle\}\ (i_k \text{ index of } \pi_{2_k}) \\ \Gamma \vdash_l \mathfrak{L}_{v_3} : \langle *\pi_{3_l}, \phi_{3_l} \rangle \\ \Gamma' = \begin{cases} \pi \leftarrow \Gamma(\pi)\ \mathit{if}\ \pi \notin \{\pi_{2_1}, \ldots \pi_{2_n}\} \\ \pi \leftarrow \{\langle \pi'_k, \phi'_k \wedge \neg \bigvee_{kl} (U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}) \rangle \\ \qquad \mid \langle \pi'_k, \phi'_k \rangle \in \Gamma(\pi_{2_k})\}\ \mathit{if}\ \pi = \pi_{2_k} \in \{\pi_{2_1}, \ldots \pi_{2_n}\} \end{cases} \\ \Gamma'' = \begin{cases} \pi_{2_1} \leftarrow (\bigcup_{jl} \langle \pi_{1_j}, U(\phi_{2_1}[i'_1 - \pi_{3_l}/i'_1]) \wedge \phi_{3_l} \wedge \phi_{1_j} \rangle) \\ \ldots \\ \pi_{2_n} \leftarrow (\bigcup_{jl} \langle \pi_{1_j}, U(\phi_{2_n}[i'_n - \pi_{3_l}/i'_n]) \wedge \phi_{3_l} \wedge \phi_{1_j} \rangle) \end{cases}\end{array}}{\Sigma, \Gamma \vdash\ v_2[v_3] = v_1\ :\ \Gamma' \cup \Gamma''}$$

Predop

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : \mathfrak{L}_{v_1}, v_2 : \mathfrak{L}_{v_2}, v_3 : \mathfrak{L}_{v_3} \\ \Gamma \vdash_j \mathfrak{L}_{v_2} : \langle *\pi_{2_j}, \phi_{2_k} \rangle \ \ \text{(rename all index variables to fresh } \boldsymbol{f_2}) \\ \Gamma \vdash_k \mathfrak{L}_{v_3} : \langle *\pi_{3_k}, \phi_{3_k} \rangle \ \ \text{(rename all index variables to fresh } \boldsymbol{f_3}) \\ \phi_{jk} = (\overline{\pi_{2_j}} \text{ predop } \overline{\pi_{3_k}}) \wedge \phi_{2_j} \wedge \phi_{3_k} \\ \phi_{jk}^{true} = \mathit{Eliminate}(\exists \boldsymbol{f_2}, \boldsymbol{f_3}.\ \phi_{jk}) \\ \Gamma' = \Gamma[\mathfrak{L}_{v_1} \leftarrow (\bigcup_{jk} \langle *\mathtt{1}, \phi_{jk}^{true} \rangle \cup \langle *\mathtt{0}, \neg \phi_{jk}^{true} \rangle)]\end{array}}{\Sigma, \Gamma \vdash v_1 = v_2 \text{ predop } \ v_3 : \Gamma'}$$

While Loop

$$\frac{\begin{array}{l}\Gamma_P = \mathit{Parametrize}(\Gamma) \\ \Sigma \vdash v : \mathfrak{L}_v \\ \Gamma_P \vdash \mathfrak{L}_v : \{\langle *\mathtt{1}, \phi_{true} \rangle, \langle *\mathtt{0}, \phi_{false} \rangle\} \\ \Sigma, \Gamma_P \vdash S : \Gamma'' \quad \Gamma''' = \Gamma'' \wedge \phi_{true} \\ \Delta = \Gamma''' - \Gamma_P \quad \Delta_n = \mathit{fix}(\Delta) \\ \Delta_{gen} = \mathit{Generalize}(\Delta_n) \\ \Gamma_{final} = \Gamma \circ \Delta_{gen} \ \text{(Generalized Fluid Update)}\end{array}}{\Sigma, \Gamma \vdash \mathit{while}\ v \neq 0\ \mathit{do}\ S\ \mathit{end}\ : \Gamma_{final}}$$

Fig. 2. Rules describing the basic analysis

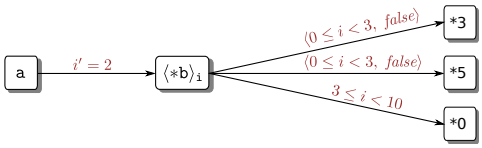


Fig. 3. Here, `a` points to the third element of an array of size 10, whose first three elements have the value 3 or 5, and the remaining elements are 0.

Example 2. Consider performing `t = a[1]` on the symbolic heap abstraction shown in Figure 3. Here, $\mathfrak{L}_{v_2}$ is the memory location labeled `a`, the only target π_{2_j} of $\mathfrak{L}_{v_2}$ is $\langle *\mathtt{b} \rangle_{\mathtt{i}}$, and the only π_{3_k} is `1`. The constraint $\phi'_{2_{jk}}$ is $U((i' = 2)[i'/i' - 1])$, which is $i = 3$. Thus, we need to determine the target(s) of the fourth element in array $\langle *\mathtt{b} \rangle_{\mathtt{i}}$. There are three targets $\pi_{t_{jl}}$ of $\langle *\mathtt{b} \rangle_{\mathtt{i}}$: $*\mathtt{3}, *\mathtt{5}, *\mathtt{0}$; hence, we compute $\phi'_{t_{jkl}}$ once for each $\pi_{t_{jkl}}$. The only satisfiable edge under constraint $i = 3$ is the edge to $*\mathtt{0}$ and we compute

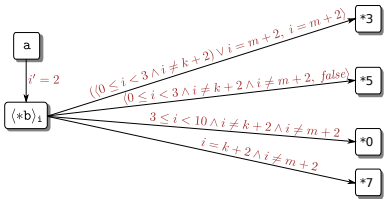


Fig. 4. Graph after processing the statements in Example 3

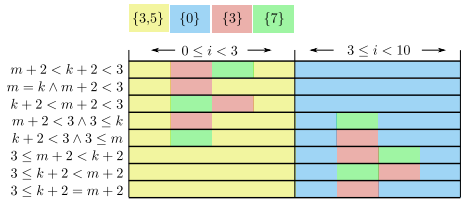


Fig. 5. Colored rectangles illustrates the partitions in Example 3; equations on the left describe the ordering between variables.

Eliminate($\exists i.\ 3 \le i < 10 \wedge i = 3$), which is *true*. Thus, the value of `t` is guaranteed to be 0 after this statement.

The Array Store rule performs a fluid update on an abstract memory location associated with an array. In this rule, each $\pi_{2_k} \in \{\pi_{2_1} \ldots \pi_{2_n}\}$ represents an array location, a subset of whose elements may be written to as a result of this store. Γ' represents the symbolic heap abstraction after removing the points-to edges from array elements that are written to by this store while preserving all other edges, and Γ'' represents all edges added by this store. Hence, Γ' and Γ'' are unioned to obtain the symbolic heap abstraction after the store. Note that Γ' preserves the existing targets of any access path $\pi \notin \{\pi_{2_1} \ldots \pi_{2_n}\}$. The points-to targets of those elements of $\pi_{2_1}, \ldots \pi_{2_n}$ that are not affected by this store are also preserved in Γ' while elements that are written to by the store are killed in Γ'. This is because elements that are updated by the store *must* satisfy $U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}$ for some k, l such that the edge to π'_k is effectively killed for those elements updated by the store. On the other hand, elements that are *not* affected by the store are guaranteed not to satisfy $U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}$ for any k, l, i.e., $\neg \bigvee_{kl}(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}) = \mathit{false}$, and the existing edge to π'_k is therefore preserved. Note that negation is only used in the Fluid Update rule; the soundness of negation, and therefore the correctness of fluid updates, relies on using bracketing constraints.

Example 3. Consider the effect of the following store instructions

```
a[k] = 7;  a[m] = 3;
```

on Figure 3. Suppose `k` and `m` are symbolic, i.e., their values are unknown. When processing the statement `a[k]` $= 7$, the only location stored into, i.e., π_{2_k}, is $\langle$`*b`$\rangle_{\mathtt{i}}$. The only π_{3_l} is `k` under *true*, and the only π_{1_j} is `*7` under *true*. The elements of $\langle$`*b`$\rangle_{\mathtt{i}}$ updated by the store are determined from $U((i' = 2)[i' - \mathtt{k}/i']) = (i = \mathtt{k} + 2)$. Thus, a new edge is added from $\langle$`*b`$\rangle_{\mathtt{i}}$ to `*7` under $i = \mathtt{k} + 2$ but all outgoing edges from $\langle$`*b`$\rangle_{\mathtt{i}}$ are preserved under the constraint $i \neq \mathtt{k} + 2$. Thus, after this statement, the edge from $\langle$`*b`$\rangle_{\mathtt{i}}$ to `*3` and `*5` are qualified by the constraint $\langle 0 \le i < 3 \wedge i \neq k + 2,\ \mathit{false}\rangle$, and the edge to `*0` is qualified by $3 \le i < 10 \wedge i \neq k + 2$. The instruction `a[m]` $=$ `3` is processed similarly; Figure 4 shows the resulting symbolic heap abstraction after these store instructions. Note that if $k = m$, the graph correctly reflects `a[k]` must be 3. This is because if $k = m$, the constraint

on the edge from $\langle *\mathtt{b} \rangle_{\mathtt{i}}$ to $*\mathtt{7}$ is unsatisfiable. Since the only other feasible edge under the constraint $i = k + 2$ is the one to $*\mathtt{3}$, $k = m$ implies $\mathtt{a}[\mathtt{k}]$ must be 3.

As Example 3 illustrates, fluid updates do not construct explicit partitions of the heap when different symbolic values are used to store into an array. Instead, all "partitions" are implicitly encoded in the constraints, and while the constraint solver may eventually need to analyze all of the cases, in many cases it will not because a query is more easily shown satisfiable or unsatisfiable for other reasons. As a comparison, in Example 3, approaches that eagerly construct explicit partitions may be forced to enumerate all partitions created due to stores using symbolic indices. Figure 5 shows that eight different heap configurations arise after performing the updates in Example 3. In fact, only one more store using a symbolic index could create over 50 different heap configurations.

In the Pointer Arithmetic rule, the index variable i' is replaced by $i' - \pi_{3_k}$ in the index constraint ϕ_{2_j}, effectively incrementing the value of i' by v_3. We also discuss the Predop rule, since some complications arise when array elements are used in predicates. In this rule, we make use of an operation $\overline{\pi}$ which converts an access path to a term in the constraint language:

$$\overline{\pi_R} = \pi_R \quad \text{if } \pi_R \in \{c, \mathfrak{L}_v, \mathit{alloc}_{id}\} \qquad \overline{*\pi} = \mathit{deref}(\overline{\pi})$$
$$\overline{\langle \pi \rangle_i} = \mathit{select}(\overline{\pi}, i) \qquad \overline{\pi_1 \text{ intop } \pi_2} = \overline{\pi_1} \text{ intop } \overline{\pi_2}$$

In this rule, notice that index variables used in the targets of $\mathfrak{L}_{v_2}$ and $\mathfrak{L}_{v_3}$ are first renamed to fresh variables $\boldsymbol{f_2}$ and $\boldsymbol{f_3}$ to avoid naming conflicts and are then existentially quantified and eliminated similar to computing $\phi_1 \downarrow_I \phi_2$. The renaming of index variables is necessary since naming conflicts arise when $\langle *\pi_{2_j}, \phi_{2_j} \rangle$ and $\langle *\pi_{3_k}, \phi_{3_k} \rangle$ refer to different elements of the same array. [2]

In the If Statement rule, observe that the constraint under which $v \neq 0$ evaluates to true (resp. false) is conjoined with all the edge constraints in Γ' (resp. Γ''); hence, the analysis is path-sensitive. We defer discussion of the While Loop rule until Section 5.

4.1 Soundness of the Memory Abstraction

We now state the soundness theorem for our memory abstraction. For a concrete store S, we use the notation $S(l_s, l_t) = \mathit{true}$ if $S(l_s) = l_t$ and $S(l_s, l_t) = \mathit{false}$ otherwise. Similarly, we write $\Gamma(\pi_s, \pi_t) = \phi$ to denote that the bracketing constraint associated with the edge from π_s to π_t is ϕ, and ϕ is *false* if there is no edge between π_s and π_t. Recall that $\mathfrak{I}(\pi)$ denotes the set of index variables in π, and we write $\sigma_{\mathfrak{I}(\pi)}$ to denote some concrete assignment to the index variables in $\mathfrak{I}(\pi)$; $\sigma'_{\mathfrak{I}(\pi)}$ is an assignment to $\mathfrak{I}(\pi)$ with all index variables primed. The notation $\sigma(\phi)$ applies substitution σ to ϕ. Finally, we use a function $\mathit{eval}^{\star}(\phi, E, S)$ for $\star \in \{+, -\}$ which evaluates the truth value of the necessary and sufficient conditions of constraint ϕ for some concrete environment E and concrete store S; this function is precisely defined in Appendix B.

[2] Quantifier elimination performed here may not be exact; but since we use bracketing constraints, we compute quantifier-free over- and underapproximations. For instance, [11] presents a technique for computing covers of existentially quantified formulas in combined theories involving uninterpreted functions. Another alternative is to allow quantification in our constraint language.

Definition 2 (Agreement) We say a concrete environment and concrete store (E, S) *agrees with* abstract environment and abstract store (Σ, Γ) (written $(E, S) \sim (\Sigma, \Gamma)$) if and only if the following conditions hold:

1. E and Σ have the same domain
2. If $S(l_s, l_t) = b$ and $\Gamma(\pi_s, \pi_t) = \langle \phi^+, \phi^- \rangle$, then for all substitutions $\sigma_{\mathfrak{I}(\pi_s)}, \sigma'_{\mathfrak{I}(\pi_t)}$ such that $l_s \in \gamma_c(E, S, \pi_s, \sigma_{\mathfrak{I}(\pi_s)})$ and $l_t \in \gamma_c(E, S, \pi_t, \sigma'_{\mathfrak{I}(\pi_t)})$, we have:

$$eval^-(\sigma'(\sigma(\phi^-)), E, S) \Rightarrow b \Rightarrow eval^+(\sigma'(\sigma(\phi^+)), E, S)$$

Theorem 1. (Soundness) Let P be any program. If $(E, S) \sim (\Sigma, \Gamma)$, then

$$E, S \vdash P : S' \Rightarrow (\Sigma, \Gamma \vdash P : \Gamma' \ \wedge \ (E, S') \sim (\Sigma, \Gamma'))$$

We sketch the proof of Theorem 1 in Appendix C.

5 Fluid Updates in Loops

In loop-free code, a store modifies one array element, but stores inside a loop often update many elements. In this section, we describe a technique to over- and underapproximate the set of concrete elements updated in loops. The main idea of our approach is to analyze the loop body and perform a fixed-point computation parametric over an *iteration counter*. Once a fixed-point is reached, we use quantifier elimination to infer elements that may and must be modified by the loop. [3]

5.1 Parametrizing the Symbolic Heap Abstraction

When analyzing loops, our analysis first identifies the set of scalars modified by the loop; we call such values *loop-dependent* scalars. We then infer equalities relating each loop-dependent scalar to the unique iteration counter k for that loop. The iteration counter k is assumed to be initialized to 0 at loop entry and is incremented by one along the back edge of the loop. We say that a loop-dependent value i is *linear* with respect to the loop if $i - i_0 = c * k$ for some constant $c \neq 0$. We compute a set of equalities relating loop-dependent scalars to the iteration counter using standard linear invariant generation techniques [12, 13]. At loop entry, we use these linear equalities to modify Γ as follows:

- Let π be a linear loop-dependent scalar with the linear relation $\pi = \pi_0 + c*k$, and let $\langle *\pi_t, c_t \rangle \in \Gamma(\pi)$. Then, replace π_t by $\pi_t + c * k$.
- Let π be a loop-dependent value not linear in k. Then, $\Gamma(\pi) \leftarrow \{\langle \top, \mathit{true} \rangle\}$.

Thus, all loop-dependent scalars are expressed in terms of their value at iteration k or $\top$; analysis of the loop body proceeds as described in Section 4.

Example 4. Consider the `send_packets` function from Section 1. Here, we infer the equality $j = j_0 + 2k$, and Γ initially contains an edge from j to $*(j_0 + 2k)$.

[3] In this section, we assume no pointer arithmetic occurs in loops; our implementation, however, does not make this restriction.

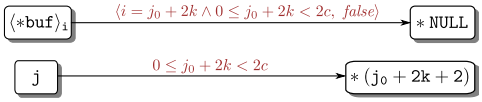


Fig. 6. The effect set after analyzing the loop body once in function `send_packets`

5.2 Fixed-Point Computation

Next, we perform a fixed-point computation (parametric on k) over the loop's net effect on the symbolic heap abstraction. This is necessary because there may be loop carried dependencies through heap reads and writes. We define the net effect of the loop on the symbolic heap abstraction during some iteration k as the *effect set*:

Definition 3 (Effect Set Δ) Let Γ' be a symbolic heap obtained by performing fluid updates on Γ. Let $\Delta = \Gamma' - \Gamma$ be the set of edges such that if ϕ qualifies edge e in Γ and ϕ' qualifies e in Γ', then Δ includes e under constraint $\phi' \wedge \neg\phi$ (where $\phi = \mathit{false}$ if $e \notin \Gamma$). We call Δ the *effect set* of Γ' with respect to Γ.

Example 5. Figure 6 shows the effect set of the loop in `send_packets` after analyzing its body once. (Edges with *false* constraints are not shown.) Note that the constraints qualifying edges in this figure are parametric over k.

We define $\Gamma \circ \Delta$ as the generalized fluid update that applies Δ to Γ:

Definition 4 ($\Gamma \circ \Delta$) Let π be a location in Γ and let S_π denote the edges in Δ whose source is π. Let $\delta(S_\pi)$ be the disjunction of constraints qualifying edges in S_π, and let I be the set of index variables used in the target locations in S_π but not the source. Let $\mathit{Update}(\pi) = \mathit{Eliminate}(\exists I.\delta(S_\pi))$. Then, for each $\pi \in \Gamma$:

$$(\Gamma \circ \Delta)[\pi] = (\Gamma(\pi) \wedge \neg \mathit{Update}(\pi)) \cup S_\pi$$

The above definition is a straightforward generalization of the fluid update operation given in the Store rule of Figure 2. Instead of processing a single store, it reflects the overall effect on Γ of a set of updates defined by Δ. The fixed-point computation is performed on Δ. We denote an edge from location π_s to π_t qualified by constraint ϕ as $\langle \pi_s, \pi_t \rangle \backslash \phi$. Since we compute a least fixed point, $\langle \pi_s, \pi_t \rangle \backslash \langle \mathit{false}, \mathit{true} \rangle \in \bot$ for all legal combinations (i.e., obeying type restrictions) of all $\langle \pi_s, \pi_t \rangle$ pairs. Note that the edge constraints in $\bot$ are the inconsistent bounds $\langle \mathit{false}, \mathit{true} \rangle$ representing the strongest over- and underapproximations. We define a $\sqcup$ and $\sqsubseteq$ on effect sets as follows:

$$\langle \pi_s, \pi_t \rangle \backslash \langle (\phi_{nc1} \vee \phi_{nc2}), (\phi_{sc1} \wedge \phi_{sc2}) \rangle \in \Delta_1 \sqcup \Delta_2 \iff (\langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc1}, \phi_{sc1} \rangle \in \Delta_1 \wedge \langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc2}, \phi_{sc2} \rangle \in \Delta_2)$$

$$\Delta_1 \sqsubseteq \Delta_2 \iff ((\phi_{nc1} \Rightarrow \phi_{nc2} \wedge \phi_{sc2} \Rightarrow \phi_{sc1}) \forall \langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc1}, \phi_{sc1} \rangle \in \Delta_1 \wedge \forall \langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc2}, \phi_{sc2} \rangle \in \Delta_2)$$

Let Γ_0 be the initial symbolic heap abstraction before the loop. We compute Γ^n_{entry} representing the symbolic heap on entry to the n'th iteration of the loop:

$$\Gamma^n_{entry} = \begin{cases} \Gamma_0 & \text{if} \quad n = 1 \\ \Gamma_0 \circ (\Delta_{n-1}[k-1/k]) & \text{if} \quad n > 1 \end{cases}$$

Γ^n_{exit} is obtained by analyzing the body of the loop using Γ^n_{entry} at the entry point of the loop. In the definition of Γ^n_{entry}, the substitution $[k-1/k]$ normalizes the effect set with respect to the iteration counter so that values of loop-dependent scalars always remain in terms of their value at iteration k. We define Δ_n representing the total effect of the loop in n iterations as follows:

$$\Delta_n = \begin{cases} \bot & \text{if } n = 0 \\ (\Gamma^n_{exit} - \Gamma^n_{entry}) \sqcup \Delta_{n-1} & \text{if } n > 0 \end{cases}$$

First, observe that $\Delta_{n-1} \sqsubseteq \Delta_n$ by construction (monotonicity). Second, observe the analysis cannot create an infinite number of abstract locations because (i) arrays are represented as indexed locations, (ii) pointers can be dereferenced only as many times as their types permit, (iii) all allocations are named by their allocation site, and (iv) scalars are represented in terms of their linear relation to k. However, our constraint domain does not have finite ascending chains, hence, we define a widening operator on bracketing constraints (although widening was never required in our experiments). Let $\boldsymbol{\beta}$ denote the unshared literals between any constraint ϕ_1 and ϕ_2. Then, we widen bracketing constraints as follows:

$$\phi_1 \bigtriangledown \phi_2 = \begin{array}{l} \langle(\lceil\phi_1\rceil \vee \lceil\phi_2\rceil)[\mathit{true}/\boldsymbol{\beta}] \vee (\lceil\phi_1\rceil \vee \lceil\phi_2\rceil)[\mathit{false}/\boldsymbol{\beta}], \\ (\lfloor\phi_1\rfloor \wedge \lfloor\phi_2\rfloor)[\mathit{true}/\boldsymbol{\beta}] \wedge (\lfloor\phi_1\rfloor \wedge \lfloor\phi_2\rfloor)[\mathit{false}/\boldsymbol{\beta}]\rangle \end{array}$$

Example 6. The effect set obtained in Example 5 does not change in the second iteration; therefore the fixed-point computation terminates after two iterations.

5.3 Generalization

In this section, we describe how to *generalize* the final effect set after a fixed-point is reached. This last step allows the analysis to extrapolate from the elements modified in the k'th iteration to the set of elements modified across all iterations and is based on existential quantifier elimination.

Definition 5 (Generalizable Location) We say a location identified by π is *generalizable* in a loop if (i) π is an array, (ii) if π_i is used as an index in a store to π, then π_i must be a linear function of the iteration counter, and (iii) if two distinct indices π_i and π_j may be used to store into π, then either only π_i, or only π_j (or neither) is used to index π across all iterations.

Intuitively, if a location π is generalizable, then all writes to π at different iterations of the loop must refer to distinct concrete elements. Clearly, if π is not an array, different iterations of the loop cannot refer to distinct concrete elements. If an index used to store into π is not a linear function of k, then the loop may update the same concrete element in different iterations. Furthermore, if two values that do not have the same relation with respect to k are used to store into π, then they may update the same element in different iterations.

In order to generalize the effect set, we make use of a variable N unique for each loop that represents the number of times the loop body executes. If the value of N can be determined precisely, we use this exact value instead of introducing N. For instance, if a loop increments i by 1 until $i \geq \mathit{size}$, then it is easy to determine that $N = \mathit{size} - i_0$, assuming the loop executes at least once. [4] Finally, we generalize the effect set as follows:

- If an edge qualified by ϕ has a generalizable source whose target does not mention k, the generalized constraint is $\phi' = \mathit{Eliminate}(\exists k.\ (\phi \wedge 0 \leq k < N))$.
- If an edge qualified by ϕ does not have a generalizable source, the generalized constraint is $\phi' = \mathit{Eliminate}\langle \exists k.\ \phi\ \wedge\ 0 \leq k < N,\ \ \forall k.\ 0 \leq k < N \Rightarrow \phi \rangle$ [5].
- If π is a loop-dependent scalar, then $\Delta[\pi] \leftarrow \Delta[\pi][N/k]$.

We now briefly explain these generalization rules. If the source of an edge is generalizable, for each iteration of the loop, there exists a corresponding concrete element of the array that is updated during this iteration; thus, k is existentially quantified in both the over- and underapproximation. The constraint after the existential quantifier elimination specifies the set of concrete elements updated by the loop. If the source is not generalizable, it is unsafe to existentially quantify k in the underapproximation since the same concrete element may be overwritten in future iterations. One way to obtain an underapproximation is to universally quantify k because if the update happens in all iterations, then the update must happen after the loop terminates. According to the last rule, loop-dependent scalar values are assigned to their value on termination. Once the effect set is generalized, we apply it to Γ_0 to obtain the final symbolic heap abstraction after the loop.

Example 7. Consider the effect set given in Figure 6. In the `send_packets` function, $\langle *\texttt{buf} \rangle_{\texttt{i}}$ is generalizable since `j` is linear in k and no other value is used to index $\langle *\texttt{buf} \rangle_{\texttt{i}}$. Furthermore, if the loop executes, it executes exactly c times; thus $N = c$. To generalize the edge from $\langle *\texttt{buf} \rangle_{\texttt{i}}$ to $*\texttt{NULL}$, we perform quantifier elimination on $\langle \exists k.i = j_0 + 2k \wedge 0 \leq j_0 + 2k < 2c \wedge 0 \leq k < c,\ \mathit{false} \rangle$, which yields $\langle j_0 \leq i \wedge i < j_0 + 2c \wedge (i - j_0)\%2 = 0,\ \mathit{false} \rangle$. Since j_0 is 0 at loop entry, after applying the generalized effect set to Γ_0 , we obtain the graph from Figure 1.

6 Implementation and Extensions

We have implemented the ideas presented in this paper in the Compass program verification framework for analyzing C programs. For solving constraints, Compass utilizes a custom SMT solver called Mistral [14], which also provides support for simplifying constraints. Compass does not assume type safety and

[4] Even though it is often not possible to determine the exact value of N, our analysis utilizes the constraint $(\forall k.0 \leq k < N \Rightarrow \neg\phi_{term}(k))\ \wedge\ \phi_{term}(N)$ stating that the termination condition ϕ_{term} does not hold on iterations before N but holds at the N'th iteration. Our analysis takes this "background axiom" into account when determining satisfiability and validity.

[5] We can eliminate a universally quantified variable k from $\forall k.\phi$ by eliminating existentially quantified k in the formula $\neg\exists k.\neg\phi$.

handles casts soundly using a technique based on physical subtyping [15]. Compass supports most features of the C language, including structs, unions, multi-dimensional arrays, dynamic memory allocation, and pointer arithmetic. To check buffer overruns, Compass also tracks buffer and allocation sizes. For inter-procedural analysis, Compass performs a path- and context-sensitive summary-based analysis. Loop bodies are analyzed in isolation before the function or loop in which they are defined; thus techniques from Section 5 extend to nested loops.

While the language defined in Section 2 only allows loops with a single exit point, techniques described in this paper can be extended to loops with multiple break points by introducing different iteration counters for each backedge, similar to the technique used in [16] for complexity analysis.

Compass allows checking arbitrary assertions using a `static_assert`(...) primitive, which can be either manually or automatically inserted (e.g., for memory safety properties). The `static_assert` primitive also allows for checking quantified properties, such as "all elements of arrays `a` and `b` are equal" by writing:

```
static_assert(buffer_size(b) == buffer_size(a));
for(i=0; i<buffer_size(a); i++) static_assert(a[i] == b[i]);
```

7 Experiments

7.1 Case Study on Example Benchmarks

To demonstrate the expressiveness of our technique, we evaluate it on 28 challenging array benchmarks available at `http://www.stanford.edu/~tdillig/array.tar.gz` and shown in Figure 7. The functions `init` and `init_noncost` initialize all elements of an array to a constant and an iteration-dependent value respectively. `init_partial` initializes part of the array, and `init_even` initializes even positions. `2D_array_init` initializes a 2-dimensional array using a nested loop. The programs labeled `_buggy` exhibit subtle bugs, such as off-by-one errors. Various versions of `copy` copy all, some, or odd elements of an array to another array. `reverse` reverses elements, while `swap` (shown in Figure 8) swaps the contents of two arrays. `double_swap` invokes `swap` twice and checks that both arrays are back in their initial state. `strcpy`, `strlen`, and `memcpy` implement the functionality of the standard C library functions and assert their correctness. `find` (resp. `find_first_nonnull`) looks for a specified (resp. non-null) element and returns its index (or -1 if element is not found). `append` appends the contents of one array to another, and `merge_interleave` interleaves odd and even-numbered elements of two arrays into a result array. The function `alloc_fixed_size` initializes all elements of a double array to a freshly allocated array of fixed size, and then checks that buffer accesses to the element arrays are safe. The function `alloc_nonfixed_size` initializes elements of the double array `a` to freshly allocated arrays of different size, encoded by the elements of another array `b` and checks that accessing indices $[0, \texttt{b}[\texttt{i}-1]]$ of array $\texttt{a}[\texttt{i}]$ is safe. Compass can automatically verify the full functional correctness of all of the correct programs without any annotations and reports all errors present in buggy programs. To check functional correctness, we add static assertions as described in Section 6 and as shown in Figure 8.

Program	Time	Memory	#Sat queries	Solve time
init	0.01s	< 1 MB	172	0s
init_nonconst	0.02s	< 1 MB	184	0.01s
init_partial	0.01s	< 1MB	166	0.01s
init_partial_buggy	0.02s	< 1 MB	168	0s
init_even	0.04s	< 1 MB	146	0.04s
init_even_buggy	0.04s	< 1 MB	166	0.03s
2D_array_init	0.04s	< 1 MB	311	0.04s
copy	0.01s	< 1 MB	209	0.01s
copy_partial	0.01s	< 1 MB	220	0.01s
copy_odd	0.04s	< 1 MB	243	0.02s
copy_odd_buggy	0.05s	< 1 MB	246	0.05s
reverse	0.03s	< 1 MB	273	0.01s
reverse_buggy	0.04s	< 1 MB	281	0.02s
swap	0.12s	2 MB	590	0.11s
swap_buggy	0.11s	2 MB	557	0.06s
double_swap	0.16s	2 MB	601	0.1s
strcpy	0.07s	< 1 MB	355	0.04s
strlen	0.02s	< 1 MB	165	0.01s
strlen_buggy	0.01s	< 1 MB	89	0.01s
memcpy	0.04s	< 1 MB	225	0.04s
find	0.02s	< 1 MB	119	0.02s
find_first_nonnull	0.02s	< 1 MB	183	0.02s
append	0.02s	< 1 MB	183	0.01s
merge_interleave	0.09s	< 1 MB	296	0.07s
merge_interleave _buggy	0.11s	< 1 MB	305	0.09s
alloc_fixed_size	0.02s	< 1 MB	176	0.02s
alloc_fixed_size_buggy	0.02s	< 1 MB	172	0.02s
alloc_nonfixed_size	0.03s	< 1 MB	214	0.02

Fig. 7. Case Study

```
void swap(int* a, int* b, int size) {
  for(int i=0; i<size; i++) {
    int t = a[i]; a[i] = b[i]; b[i] = t; }
}
void check_swap(int size, int* a, int* b) {
  int* a_copy = malloc(sizeof(int)*size);
  int* b_copy = malloc(sizeof(int)*size);
  for(int i=0; i<size; i++)  a_copy[i] = a[i];
  for(int i=0; i<size; i++)  b_copy[i] = b[i];
  swap(a, b, size);
  for(i=0; i<size; i++) {
    static_assert(a[i] == b_copy[i]);
    static_assert(b[i] == a_copy[i]);
  }
  free(a_copy); free(b_copy);
}
```

Fig. 8. Swap Function from Figure 7. The static assertions check that all elements of `a` and `b` are indeed swapped after the call to the `swap` function. Compass verifies these assertions automatically in 0.12 seconds.

Figure 7 reports for each program the total running time, memory usage (including the constraint solver), number of queries to the SMT solver, and constraint solving time. All experiments were performed on a 2.66 GHz Xeon workstation. We believe these experiments demonstrate that Compass reasons precisely and efficiently about array contents despite being fully automatic. As a comparison, while Compass takes 0.01 seconds to verify the full correctness of `copy`, the approach described in [4] reports a running time of 338.1 seconds, and the counterexample-guided abstraction refinement based approach described in [17] takes 3.65 seconds. Furthermore, our technique is naturally able to verify the correctness of programs that manipulate non-contiguous array elements (e.g., `copy_odd`), as well as programs that require reasoning about arrays inside other arrays (e.g., `alloc_nonfixed_size`). Figure 7 also shows that the analysis is memory efficient since none of the programs require more than 2 MB. We believe this to be the case because fluid updates do not create explicit partitions.

Observe that the choice of benchmarks in Figure 7 sheds light on both what our technique is good at and what it is *not* meant for. In particular, notice these benchmarks do not include sorting routines. While sorting is an interesting problem for invariant generation techniques, the focus of this work is improving static analysis of updates to aggregate data structures, such as arrays, through fluid updates. As shown in Section 5, fluid updates can be combined with invariant generation techniques to analyze loops, but we do not claim that this particular invariant generation approach is the best possible. We leave as future work the combination of fluid updates and more powerful invariant generation techniques.

7.2 Checking Memory Safety on Unix Coreutils Applications

To evaluate the usefulness of our technique on real programs, we also check for memory safety errors on five Unix Coreutils applications [20] that manipulate arrays and pointers in complex ways. In particular, we verify the safety

Program	Lines	Total Time	Memory	#Sat queries	Solve Time
hostname	304	0.13s	5 MB	1533	0.12s
chroot	371	0.13s	3 MB	1821	0.10s
rmdir	483	1.05s	12 MB	3461	1.02s
su	1047	1.86s	32 MB	6088	1.69s
mv	1151	0.70s	21 MB	7427	0.68s
Total	3356	3.87s	73 MB	20330	3.61

Fig. 9. Experimental results on Unix Coreutils applications.

of buffer accesses (both buffer overruns and underruns) and pointer dereferences. However, since Compass treats integers as mathematical integers, the soundness of the buffer analysis assumes lack of integer overflow errors, which can be verified by a separate analysis. In the experiments, Compass reports zero false positives, only requiring two annotations describing inputs to `main`: `assume(buffer_size(argv) == argc)` and `assume(argv != NULL)`). Compass is even able to discharge some arbitrary assertions inserted by the original programmers. Some of the buffer accesses that Compass can discharge rely on complex dependencies that are difficult even for experienced programmers to track; see Appendix D for an interesting example.

The chosen benchmarks are challenging for static analysis tools for multiple reasons: First, these applications heavily use arrays and string buffers, making them difficult for techniques that do not track array contents. Second, they heavily rely on path conditions and correlations between scalars used to index buffers. Finally, the behavior of these applications depends on environment choice, such as user input. Our technique is powerful enough to deal with these challenges because it is capable of reasoning about array elements, is path-sensitive, and uses bracketing constraints to capture uncertainty. To give the reader some idea about the importance of these components, 85.4% of the assertions fail if array contents are smashed and 98.2% fail if path-sensitivity is disabled.

As Figure 9 illustrates, Compass is able to analyze all applications in under 2 seconds, and the maximum memory used both for the program verification and constraint solving combined is less than 35 MB. We believe these running times and memory requirements demonstrate that the current state of Compass is useful and practical for verifying memory safety in real modest-sized C applications manipulating arrays, pointers, and scalars in complex ways.

8 Related Work

Reasoning about unbounded data structures has a long history. Jones et al. first propose *summary nodes* to finitely represent lists in LISP [21], and [1] extends this work to languages with updates and introduces strong and weak updates. Representation of access paths qualified by indices is first introduced in Deutsch [22], which uses a combination of *symbolic access paths* and numeric abstract domains to represent may-alias pairs for recursive data structures. This technique does not address arrays, and since it does not reason about updates, negation is not a consideration. Deutsch's technique does not allow disjunctive constraints, is not path-sensitive, and does not address underapproximations.

The most basic technique for reasoning about array contents is *array smashing*, which represents all elements with one summary node and only allows weak updates [2]. Gopan et al. propose a 3-valued logic based framework to discover

relationships about values of array elements [4]. This technique isolates individual elements to perform strong updates and places elements that share a common property into a partition (usually a contiguous range), and relevant partitions are heuristically inferred. In contrast, our approach does not need to distinguish between strong and weak updates or concretize individual elements; it can also naturally express invariants about non-contiguous array elements. Furthermore, our approach obviates the need for explicit partitioning, and effectively delays decisions about partitions until constraint solving. While many factors contribute to the overall performance of program analysis systems, we believe our tool's significantly better performance over [4] is largely due to avoiding the construction of explicit partitions. Jhala and McMillan propose a technique similar to [4] for reasoning about arrays, but their technique is based on counterexample guided abstraction refinement and interpolation [17]. This approach also only reasons about contiguous ranges and constructs explicit partitions. Furthermore, the predicates used in the abstraction belong to a finite language to guarantee convergence.

Many techniques have been proposed for generating invariants about elements of unbounded data structures [18, 19, 23–26]. Some of these techniques can reason about complex data invariants, such as sortedness, which is orthogonal to the ability to perform fluid updates. Unlike these approaches whose goal is to discover complex invariants about array elements, our goal is to design an expressive pointer and value analysis that unifies reasoning about pointers, scalars, and arrays. However, we believe these techniques can be gainfully combined.

Concepts similar to the *iteration counter* from Section 5 have been previously proposed. For example, Gulwani et al. [16] use an iteration counter for performing complexity analysis. The invariant generation technique described in [19] also uses a combination of an iteration counter combined with quantifier elimination.

Our technique uses bracketing constraints to represent both over- and underapproximations to naturally handle imprecision and uncertainty. Furthermore, bracketing constraints allow for a sound negation operation in the presence of approximations. The idea of over- and underapproximations has been proposed previously in the context of abstract interpretation by Schmidt [27]; however, the techniques presented there are not concerned with negation. In this paper, we share the goal of gracefully handling imprecision when analyzing unbounded data structures with [28], which presents a compositional shape analysis based on separation logic. In contrast to [28] which focuses exclusively on recursive pointer data structures, such as linked lists, this paper focuses on arrays. We believe our approach can be extended to at least some useful recursive data structures, such as lists, and we leave this extension as future work.

References

1. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI, NY, USA, ACM (1990) 296–310
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software (2002)

3. Reps, T.W., Sagiv, S., Wilhelm, R.: Static program analysis via 3-valued logic. In: CAV. Volume 3114 of Lecture Notes in Comp. Sc., Springer (2004) 15–30
4. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: POPL, NY, USA, ACM (2005) 338–350
5. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the saturn project. In: PASTE, NY, USA, ACM (2007) 43–48
6. Ball, T., Rajamani, S.: The slam project: debugging system software via static analysis. In: POPL, NY, USA (2002) 1–3
7. Lee, S., Cho, D.: Packet-scheduling algorithm based on priority of separate buffers for unicast and multicast services. Electronics Letters **39**(2) (Jan 2003) 259–260
8. Nguyen, K., Nguyen, T., Cheung, S.: P2p streaming with hierarchical network coding (July 2007)
9. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. SIGPLAN Not. **27**(7) (1992) 235–248
10. Cooper, D.: Theorem proving in arithmetic without multiplication. Machine Intelligence **7** (1972) 91–100
11. Gulwani, S., Musuvathi, M.: Cover algorithms. In: ESOP. (2008) 193–207
12. Karr, M.: Affine relationships among variables of a program. A.I. (1976) 133–151
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, NY, USA, ACM (1978) 84–96
14. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: In CAV, Springer (2009)
15. Chandra, S., Reps, T.: Physical type checking for c. SIGSOFT **24**(5) (1999) 66–75
16. Gulwani, S., Mehra, K., Chilimbi, T.: SPEED: precise and efficient static estimation of program computational complexity. In: POPL. (2009) 127–139
17. Jhala, R., Mcmillan, K.L.: Array abstractions from proofs. In: CAV. (2007)
18. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI, NY, USA, ACM (2008) 339–348
19. Kovacs, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: FASE 2009, Springer (2009) 470–485
20. http://www.gnu.org/software/coreutils/: Unix coreutils
21. Jones, N., Muchnick, S.: Flow analysis and optimization of LISP-like structures. In: POPL, ACM NY, USA (1979) 244–256
22. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: PLDI, ACM NY, USA (1994) 230–241
23. Allamigeon, X.: Non-disjunctive numerical domain for array predicate abstraction. In: ESOP. (2008) 163–177
24. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM New York, NY, USA (2008) 235–246
25. Seghir, M., Podelski, A., Wies, T.: Abstraction Refinement for Quantified Array Assertions. In: SAS, Springer-Verlag (2009) 3
26. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, ACM NY, USA (2002) 191–202
27. Schmidt, D.A.: A calculus of logical relations for over- and underapproximating static analyses. Sci. Comput. Program. **64**(1) (2007) 29–53
28. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, ACM NY, USA (2009) 289–300
29. Cousot, P.: Verification by abstract interpretation. Lecture notes in computer science (2003) 243–268

Appendix A

Here, we present an operational semantics of the language from Section 2. A concrete location l_c is a pair (s, i) where s is a start address for a block of memory and i is an offset from s. For scalars, we use the notation $(v, \cdot)$ to indicate that the value stored in l_c is v and that the offset is not relevant. The environment E: $\texttt{Var} \rightarrow l_c$ maps variables to concrete locations, and the store S: $l_c \rightarrow l_c$ maps locations to other locations. The notation $S' = S[l \leftarrow \mathfrak{e}]$ denotes that store S' is identical to store S except that it maps location l to $\mathfrak{e}$. The function $newloc(S, c)$ returns the start address of freshly allocated memory containing c cells such that no cell overlaps existing memory cells.

Sequence

$$\frac{E, S \vdash s_1 : S' \quad E, S' \vdash s_2 : S''}{E, S \vdash s_1; s_2 : S''}$$

Variable Assign

$$\frac{E \vdash v_1 : l_1, v_2 : l_2 \quad S \vdash l_2 : \mathfrak{e}}{E, S \vdash v_1 = v_2 : S[l_1 \leftarrow \mathfrak{e}]}$$

Constant Assign

$$\frac{E \vdash v : l \quad S' = S[l \leftarrow (c, \cdot)]}{E, S \vdash v = c : S'}$$

Alloc

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2 \\ S \vdash l_2 : (c, 0) \\ s = newloc(S, c) \\ S' = S[(s, 0) \leftarrow 0, ..., (s, c-1) \leftarrow 0] \\ S'' = S'[l_1 \leftarrow (s, 0)] \end{array}}{E, S \vdash v_1 = alloc(v_2) : S''}$$

Array Load

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_3 : (c, \cdot) \\ S \vdash l_2 : (s, i) \\ S \vdash (s, i + c) : \mathfrak{e} \\ S' = S[l_1 \leftarrow \mathfrak{e}] \end{array}}{E, S \vdash v_1 = v_2[v_3] : S'}$$

Array Store

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_3 : (c, \cdot) \\ S \vdash l_2 : (s, i) \\ S \vdash l_1 : \mathfrak{e} \\ S' = S[(s, i + c) \leftarrow \mathfrak{e}] \end{array}}{E, S \vdash v_2[v_3] = v_1 : S'}$$

Pointer plus

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_2 : (s, i) \\ S \vdash l_3 : (c, \cdot) \\ S' = S[l_1 \leftarrow (s, i + c)] \end{array}}{E, S \vdash v_1 = v_2 \oplus v_3 : S'}$$

Intop, Predop

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_2 : (c, \cdot) \\ S \vdash l_3 : (c', \cdot) \\ S' = S[l_1 \leftarrow (c \text{ op } c', \cdot)] \end{array}}{E, S \vdash v_1 = v_2 \text{ op } v_3 : S'}$$

If-True

$$\frac{\begin{array}{l} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : true \\ E, S \vdash s_1 : S' \end{array}}{E, S \vdash if\, v \neq 0 \ then\ s_1 else\ s_2 : S'}$$

If-False

$$\frac{\begin{array}{l} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : false \\ E, S \vdash s_2 : S' \end{array}}{E, S \vdash if\, v \neq 0 \ then\ s_1 else\ s_2 : S'}$$

While-True

$$\frac{\begin{array}{l} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : true \\ E, S \vdash s : S' \\ E, S' \vdash while\ v \neq 0\ do\ s\ end : S'' \end{array}}{E, S \vdash while\ v \neq 0\ do\ s\ end : S''}$$

While-False

$$\frac{\begin{array}{l} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : false \end{array}}{E, S \vdash while\ v \neq 0\ do\ s\ end : S}$$

Appendix B

Here, we give a precise definition of the *eval* function used in Section 4.1. First, we define an $eval_t$ function on terms in the constraint language as follows:

$$
\begin{array}{ll}
eval_t(c, E, S) & = \{c\} \\
eval_t(v, E, S) & = \{S(l_i) |\ l_i \in \gamma(E, S, v)\} \\
eval_t(select(deref(t_1, t_2)), E, S) & = \{S(s_1, i_1 + c) \mid (s_1, i_1) \in eval_t(t_1, E, S) \wedge (c, \cdot) \in eval_t(t_2, E, S)\} \\
eval_t(deref(t), E, S) & = \{S(l_i) \mid l_i \in eval_t(t, E, S)\} \\
eval_t(t_1 \text{ intop } t_2, E, S) & = \{v_{1i} + v_{2j} \mid (v_{1i}, \cdot) \in S(l_i) \wedge l_i \in eval_t(t_1, E, S) \\
 & \quad \wedge (v_{2j}, \cdot) \in S(l_j) \wedge l_j \in eval_t(t_2, E, S)\}
\end{array}
$$

Since we only allow pointers to arrays in the language from Section 2, terms involving *select* are guaranteed to be followed by a *deref*. Thus, we give a definition for $eval_t(select(deref(t_1, t_2)))$, but not for $eval_t(select(t_1, t_2))$. We define $val(t)$ for a term t as follows:

$$val(t,\ E,\ S) = \{s_i + off_i | (s_i, off_i) \in eval_t(t, E, S)\}$$

where $off_i = 0$ if $(s_i, \cdot) \in eval_t(t, E, S)$.

Finally, we define the $eval^\star$ ($\star \in \{+, -\}$) function for constraints as follows:

$$
\begin{array}{ll}
eval^\star(b, E, S) & = b, \quad b \in \{true, false\} \\
eval^\star(t_1 \text{ predop } t_2, E, S) & = \begin{cases} \bigvee_{v_i \in val(t_1,E,S), v_j \in val(t_2,E,S)} (v_i \text{ predop } v_j) & if \star = + \\ \bigwedge_{v_i \in val(t_1,E,S), v_j \in val(t_2,E,S)} (v_i \text{ predop } v_j) & if \star = - \end{cases} \\
eval^\star(t\%c, E, S) & = \begin{cases} \bigvee_{v_i \in val(t,E,S)} (v_i\%c) & if \star = + \\ \bigwedge_{v_i \in val(t,E,S)} (v_i\%c) & if \star = - \end{cases} \\
eval^\star(\neg\phi, E, S) & = \begin{cases} \neg eval^-(\phi, E, S) & if \star = + \\ \neg eval^+(\phi, E, S) & if \star = - \end{cases} \\
eval^\star(\phi_1 \wedge \phi_2, S) & = eval^\star(\phi_1, E, S) \wedge eval^\star(\phi_2, E, S) \\
eval^\star(\phi_1 \vee \phi_2, S) & = eval^\star(\phi_1, E, S) \vee eval^\star(\phi_2, E, S)
\end{array}
$$

Appendix C

In this section, we present a proof sketch of Theorem 1. The proof is a standard induction on the inference rules given in Figure 2; here, we only establish the base case for the fluid update rule $v_2[v_3] = v_1$ using a proof by contradiction.

By assumption, $(E, S) \sim (\Sigma, \Gamma)$ before the fluid update, but suppose $(E, S') \not\sim (\Sigma, \Gamma')$ after the fluid update, i.e., there exist two concrete locations $l_s = (s, o)$ and l_t in Σ' and two abstract locations π_s and π_t in Γ' such that $l_s \in \gamma_c(E, S', \pi_s, \sigma_{\mathfrak{I}(\pi_s)})$ and $l_t \in \gamma_c(E, S', \pi_t, \sigma'_{\mathfrak{I}(\pi_t)})$ for some substitutions σ, σ' and one of the following two conditions holds:

1. $S'(l_s, l_t) = true$, but $eval^+(\sigma'(\sigma(\Gamma'(\pi_s, \pi_t))), E, S') = false$, or
2. $S'(l_s, l_t) = false$, but $eval^-(\sigma'(\sigma(\Gamma'(\pi_s, \pi_t))), E, S') = true$.

Since the arguments for conditions (1) and (2) are symmetric, we focus on disproving (1). To disprove (1), we consider two cases:

1. Either the points-to edge from l_s to l_t was added due to this store instruction, or
2. There was an edge from l_s to l_t before this instruction that was not killed by the store.

We first consider (1). By assumption, $(E, S) \sim (\Sigma, \Gamma)$; hence, π_s must correspond to some π_{2_k} in the array store rule. Furthermore, σ must assign i_k to o, otherwise $l_s \notin \gamma_c(E, S', \pi_{2_k}, \sigma_{\mathfrak{I}(\pi_{2_k})})$. Also, π_t must correspond to some π_{1_j} such that $l_t \in \gamma_c(E, S, \pi_{1_j}, \sigma'_{\mathfrak{I}(\pi_{1_j})})$ and $eval^+(\sigma'(\phi_{1_j}), E, S)$ is *true*. Consider any π_{3_l} that represents the value of $v_3 = o'$ in this execution; for such a π_{3_l}, $eval^+(\phi_{3_l}, E, S)$ must be *true*.

Thus, if an edge from l_s to l_t was added by the current store instruction but $eval^+(\sigma'(\sigma(\Gamma'(\pi_s, \pi_t))), E, S')$ is *false*, then by the argument above and the fluid update rule, it must be the case that:

$$eval^+(\sigma'(\sigma((U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l} \wedge \phi_{1_j}))), E, S) = \mathit{false}$$

From above, we already have $eval^+(\sigma'(\phi_{1_j}), E, S) = \mathit{true}$, and $eval^+(\phi_{3_l}, E, S) = \mathit{true}$. Thus,

$$eval^+(\sigma(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k])), E, S) = \mathit{false} \quad (*)$$

must hold for some σ such that $i_k : o$. Consider substitution σ'' that is the same as σ, but all index variables are replaced by their primed counterparts. Clearly, (*) implies:

$$eval^+(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]), E, S) = \mathit{false}$$

Now consider an assignment that is identical to σ'' but it assigns $o - o'$ to i_k instead of o. Since $S \sim \Gamma$:

$$eval^+(\sigma''[i'_k \leftarrow (o - o')](\phi_{2_k}), E, S) = \mathit{true}$$

because ϕ_{2_k} is the constraint under which the dereference of v_2 is π_{2_k} and $v_3 = o'$. However, this contradicts $eval^+(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]), E, S) = \mathit{false}$ since $\pi_{3_l} = o'$.

We now consider (2), i.e., suppose there is an existing edge between l_s and l_t that was not killed by this store, but $\sigma'(\sigma(eval^+(\Gamma(\pi_s, \pi_t))), E, S') = \mathit{false}$. As before, π_s corresponds to some π_{2_k} of the store rule, otherwise, none of the constraints on the outgoing edges of π_s could have been weakened in the abstraction. The fluid update weakens constraints by conjoining $\neg(\bigvee_{l,k}(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}))$ with existing edge constraints. For the edge between π_s and π_t to be killed, we must have

$$eval^-(\sigma(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}), E, S) = \mathit{true}$$

for some l, k. If we construct σ'' from σ as in case (1), clearly:

$$eval^-(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k] \wedge \phi_{3_l}), E, S) = \mathit{true}$$

As in the previous case, σ must assign i_k to o; otherwise $l_s \notin \gamma_c(E, S', \pi_{2_k}, \sigma_{\mathfrak{I}(\pi_{2_k})})$; hence σ'' assigns i'_k to o. Assume the concrete value of v_3 is o'. If this store did not update l_2 and since $S \sim \Gamma$ before the store, $eval^{-}(\sigma''[i'_k \leftarrow (o - o')](\phi_{2_k}), E, S) = \mathit{false}$. Again, this contradicts

$$eval^{-}(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k] \wedge \phi_{3_l}), E, S) = \mathit{true}$$

since π_{3_l} must represent the value of o'.

Appendix D

Here, we discuss an interesting buffer access from the Coreutils `chroot` application that is quite challenging for a human to prove safe. Figure 10 presents a simplified slice of the relevant segment of the `chroot` program. Here, our goal is to prove the safety of two buffer accesses marked `Buffer check 1` and `Buffer check 2` in comments. First, observe that there is no buffer access in `main` if `getopt_long` does not return -1 because `usage()` is an exit function, i.e., a call to `usage` terminates the program. Therefore, only the return points 1, 2, and 4 in `getopt_long` could have been taken at points where a buffer is accessed. Second, observe that the if statement marked `Cond 1` exits if `argc <= optind`. Therefore, if program point `(***)` is reached, only the exit point `Return 4` could have been taken. This is because `Return 1` is taken if `argc < 1`; since `optind` is initialized to 0, `Cond 1` would hold and `(***)` could not be reached. Similarly, return point 2 could also not have been taken if `(***)` is reached because return point 2 implies `argc <= optind`. Furthermore, observe that if `getopt_long` returns at return point 4, `optind` is at least 2. Thus, `Buffer check 1` is safe because `argc` is at least 3 inside the if statement marked `Cond 2`. It is now easy to see why `Buffer check 2` is also safe if `Cond 2` holds. If `Cond 2` does not hold, we still need to prove that `argv` has at least one remaining element since `argv` is incremented by `optind + 1` in the else branch. If the else branch is taken, from `Cond 2`, we have `argc != optind+1`, and from `Cond 1`, we know `argc > optind`. Therefore, `argc` is strictly greater than `optind+1`, and `Buffer check 2` is safe even if the else branch is taken. Compass is able to prove these buffer accesses and many other challenging ones safe fully automatically without any difficulty. As this example demonstrates, the techniques presented in this paper are not just limited to tracking contents of arrays; they are equally powerful at reasoning about scalar and pointer values.

```
int optind = 0;
int getopt_long (int argc, char **argv,...)
{
  if(argc < 1)
     return -1;  /* Return 1 */
  if(optind == 0)
     optind = 1;
  while( skip(argv[optind])
     && optind<argc) optind++;
  if(optind>=argc)
     return -1;  /* Return 2 */
  optind++;
  if(str_prefix(options,
         argv[optind-1])) {
    optarg = argv[optind-1];
    return 0;  /* Return 3 */
  }
  return -1;      /* Return 4 */
}
```

```
int main (int argc, char **argv) {
  if (getopt_long (argc, argv, "+",
      NULL, NULL) != -1) usage (EXIT_FAILURE);

  if (argc <= optind) {  /* Cond 1 */
    error (0, 0, "missing operand");
    usage (EXIT_FAILURE);
  }
  (***)
  if (argc == optind + 1) { /* Cond 2 */
    /* Buffer check 1 */
    static_assert(buffer_size(argv) > 2);
    argv[0] = shell; argv[1] = bad_cast ("-i");
    argv[2] = NULL;
  }
  else argv += optind + 1;
  /* Buffer check 2 */
  static_assert(buffer_size(argv) > 0);
  execvp (argv[0], argv);
}
```

Fig. 10. A challenging buffer access from `chroot`