Memory Leak Analysis by Contradiction ⋆

Maksim Orlovich and Radu Rugina

Computer Science Department
Cornell University
Ithaca, NY 14853
{maksim, rugina}@cs.cornell.edu

Abstract. We present a novel leak detection algorithm. To prove the absence of a memory leak, the algorithm assumes its presence and runs a backward heap analysis to disprove this assumption. We have implemented this approach in a memory leak analysis tool and used it to analyze several routines that manipulate linked lists and trees. Because of the reverse nature of the algorithm, the analysis can locally reason about the absence of memory leaks. We have also used the tool as a scalable, but unsound leak detector for C programs. The tool has found several bugs in larger programs from the SPEC2000 suite.

1 Introduction

Low-level programming languages such as C and C++ provide manual memory management and require explicit deallocation of program structures by programmers. As a result, memory leaks represent a standard cause of errors in such languages. Memory leaks are difficult to identify, as the only symptom is a slow increase in memory consumption. For long-running applications, this eventually causes to the system running out of memory. In recent years, a number of static memory leak analysis and verification techniques have been developed [1–5].

This paper presents a new static memory leak detection analysis that reasons about the absence of errors by disproving their presence. To show that an assignment in the program is leak-free, the analysis assumes the opposite – that the assignment causes the program to lose the last reference to a heap cell, called the *error cell*. Then, the algorithm performs a backward dataflow analysis, trying to contradict the feasibility of the error cell. If each backward path leads to a contradiction, then the original assumption was wrong. Otherwise, if there is a backward path that validates the feasibility of the error cell, the analysis reports a program trace that leads to the error. The memory leak analysis by contradiction approach has several appealing properties:

- It can be used to reason about the presence or absence of leaks in incomplete programs. It can determine that program fragments are leak-free regardless of their input heap, or identify the inputs under which the program fragment may leak;
- It can be used in interactive tools in a demand-driven fashion, where programmers query particular statements in the program, asking about the possibility of memory leaks at those points.

To the best of our knowledge, existing approaches to memory leak analysis and detection cannot give such guarantees, or be used in such a way.

We have implemented the proposed algorithm in a prototype analysis tool and used it to analyze memory leaks in standard library functions that manipulate recursive heap structures. The tool can show that most of these programs do not leak memory, regardless of their inputs, or leak only for malformed inputs. We have also used the tool to find errors in larger programs from the SPEC2000 benchmarks suite. In this setting, the tool analyzes each program assignment, and uses a cutoff to limit the amount of backward exploration per assignment. The tool has found several memory leaks in these programs.

The rest of the paper is organized as follows. Section 2 presents the scope of this paper. Section 3 illustrates the main idea with a simple example. We show the reverse heap analysis algorithm in Section 4. Experimental results are presented in Section 5. Finally, we discuss related work in Section 6, and conclude in Section 7.

2   Leak Classification and Analysis Scope

The notion of a memory leak is closely related to the notion of lifetime of heap cells: a cell is being leaked if the program or the run-time system doesn't reclaim its memory when the lifetime has ended. However, there are different ways of defining lifetime. These can be classified into three main categories, based on the following heap cell properties:

1. *Referencing*: the lifetime of a cell ends when it there are no references to that cell (excluding references from dead cells). This notion is used, for instance, by reference-counting garbage collectors.
2. *Reachability*: the lifetime of a cell ends when it is no longer reachable from program variables. This notion is used by tracing and copying garbage collectors.
3. *Liveness*: the lifetime of a cell ends after the last access to that cell. This is the most precise notion of lifetime.

The above three classes are increasingly stronger notions of lifetime; they correspond to three classes of leaks. A system that targets a given class of leaks cannot make guarantees about leaks in the stronger classes. For instance, reference counting approaches do not detect unreachable heap cycles; and reachability-based approaches do not detect cells that are still reachable, but no longer needed (sometimes referred to as "Java memory leaks"). The scope of this work is the analysis of referencing leaks. As some of the existing static analyses [6, 4], our technique does not detect unreachable cycles.

The analysis is sound for the imperative language described in Section 4 (essentially a subset of C without pointer arithmetic, unions, and casts), provided that the underlying points-to information is sound. We use our implemented tool in two settings:

- *Verification.* We use the tool to verify the absence of referencing leaks for algorithms written in our language.
- *Bug-finding.* We also use our implementation as a scalable error-detection tool that targets all of the C language. In this setting, the tool makes several unsound assumptions. Despite of being unsound, the tool is useful at identifying potential errors, with a relatively low number of false warnings.

```
1   typedef struct list {
2       int data;
3       struct list *next;
4   } List;
5
6   List *reverse(List *x) {
7       List *y, *t;
8       y = NULL;
9       while (x != NULL) {
10          t = x->next;
11          x->next = y;
12          y = x;
13          x = t;
14      }
15      return y;
16  }
```

Fig. 1. Example program.

3   Example

Figure 1 shows a function that performs in-place list reversal. The function takes a linked list `x` as argument, and returns the reversed list pointed to by `y`. The function is written using C syntax, but assumes type-safety.

Suppose we want to prove that the assignment `y = x` at line 12 cannot cause a memory leak. For this, assume by absurd that a leak occurs at this point. That is, assume that the execution of `y = x` causes the program to loose the last reference to a valid allocated cell at this point, the error cell. We describe this cell using a dataflow fact $\{y\}$ indicating that `y` is the sole reference to the cell in question. Starting with this fact, we analyze program statements backward, looking for a contradiction. In this case, the contradiction shows up right away, at the predecessor statement `x->next = y` at line 11. No program state before line 11 can make it possible to have a heap cell referenced only by `y` at the next program point. This is because:

- either `y` did not reference the cell in question before line 11, in which case it won't reference it after the assignment;
- or `y` did reference the cell in question, in which case there will be two distinct references to the cell after the assignment, one from `y` and one from `x->next`. These represent different memory locations because of the type-safety assumption. Hence, `y` is not the only reference to the cell after line 11;

Each of the two cases yields a contradiction. Hence, the initial assumption is invalid and the assignment `y = x` cannot cause a memory leak.

Similar analyses can be performed for each of the assignments in this program. The analysis of assignments `x->next = y` at line 11, and `x = t` at line 13 is the same as above. The analysis of `t = x->next` at line 10 requires exploring multiple backward paths, one that wraps around the loop and yields a contradiction at `x = t`, and one that

goes out of the loop and yields a contradiction at the procedure entry point where `t`'s scope begins. Finally, the analysis of `y = NULL` at line 8 is contradicted right away, as the scope of `y` begins at that point.

Additionally, the analysis must check that local variables do not leak memory when they go out of scope at the end of the procedure. The analysis models the return statement `return y` as a sequence: "`ret = y; y = NULL; t = NULL;`", where `ret` is a special return variable. Assignment `y = NULL` is contradicted by `ret = y;` and `t = NULL` is contradicted by `x = t` at line 13, and `y = NULL` at line 8. The assignment to `ret` needs not be checked.

Hence, the analysis concludes that `reverse` is leak-free. Not only the analysis has quickly contradicted each assignment, mostly using one single backward step per assignment, but the analysis has actually determined that `reverse` doesn't leak memory regardless of its input heap.

In contrast, forward heap analyses (such as [4] or [7]) cannot determine this fact because they would need to exhaustively enumerate all possible heaps at the entry of `reverse`. Without knowledge about all variables and program structures that might point into the list passed to reverse, enumerating all heaps is not feasible.

4   Backward Memory Leak Analysis

We present the memory leak analysis using a core imperative language consisting of statements and expressions with the following syntax:

$$\textit{Statements}\ \ s \in St \quad s ::= *e_0 \leftarrow e_1 \mid *e \leftarrow \texttt{malloc} \mid \texttt{free}(e) \mid \texttt{cond}(e_0 \equiv e_1)$$
$$\texttt{return}\ e \mid \texttt{enter} \mid *e_0 \leftarrow p(e_1, ..., e_k)$$

$$\textit{Expressions}\ \ e \in E \quad e ::= n \mid a \mid *e \mid e.f \mid e_0 \oplus e_1$$

where $n \in \mathbb{Z}$ ranges over numeric constants (`NULL` being represented as constant 0), $a \in A$ ranges over symbolic addresses, $f \in F$ over structure fields, $p \in P$ over procedures, $\oplus$ over arithmetic operators, and $\equiv$ over the comparison operators $=$ and $\neq$. The special statement `enter` describes the beginning of scope for local variables. The entry node of each procedure is the `enter` statement, and the exit node is the `return` statement. The condition statement $\texttt{cond}(e_0 \equiv e_1)$ ensures that the program execution proceeds only if the condition succeeds. Condition statements can be used to model if statements and while statements as non-deterministic branches in the control-flow graph, followed by the appropriate `cond` statements on each of the branches.

Expressions are represented using a small set of primitives. Symbolic addresses a are the addresses of variables (globals, locals, and parameters). We denote by a_x the symbolic address of variable x. The syntax for expressions doesn't contain variables because variables can be expressed as dereferences of their symbolic addresses. For instance, a_x models the C expression `&x`; $*a_x$ models C expression `x`; $(*a_x).f$ models `&(x->f)`; and $*((*a_x).f)$ models `x->f`. With this representation, each memory read is explicit in the form of a dereference $*e$; and each assignment has the form $*e \leftarrow e'$, making the memory write $*e$ explicit. The set $\text{Mem}(e)$ denotes the subexpressions of e

that represent memory locations. This set is defined recursively as follows:

$$\mathsf{Mem}(n) = \mathsf{Mem}(a) = \varnothing \qquad \mathsf{Mem}(e.f) = \mathsf{Mem}(e)$$
$$\mathsf{Mem}(*e) = \{*e\} \cup \mathsf{Mem}(e) \qquad \mathsf{Mem}(e_0 \oplus e_1) = \mathsf{Mem}(e_0) \cup \mathsf{Mem}(e_1)$$

To simplify the rest of the presentation, we define expression contexts $\mathcal{E}$ as expressions with holes $[\cdot]$:

$$\mathcal{E} ::= \mathcal{E}.f \mid *\mathcal{E} \mid \mathcal{E} \oplus e \mid e \oplus \mathcal{E} \mid [\cdot]$$

If $\mathcal{E}$ is an expression context and e is an expression, then $\mathcal{E}[e]$ is the expression obtained by filling the hole of the context $\mathcal{E}$ with expression e.

The core language is a subset of C that supports heap structures and pointers to variables or to heap cells. The execution of the program disallows casts between numeric constants and pointers. All structures have the same set of non-overlapping fields, so that an update to a structure field does not affect the values of other fields. Essentially, unions and unsafe pointer casts are not allowed. The language semantics are defined in Appendix A using evaluation relations for expressions and statements. The relation $(e, \sigma) \to v$ indicates that the evaluation of expression e in store σ yields value v; and the relation $(s, \sigma) \to \sigma'$ indicates that the evaluation of statement s in store σ yields a new store σ'.

4.1 Aliasing and Disjointness

To resolve pointer aliasing, the leak analysis assumes that an underlying analysis provides: 1) a partitioning of the memory into regions (i.e. different regions model disjoint sets of memory locations); and 2) points-to information between regions. We assume a flow-insensitive points-to interface consisting of a set Rgn of regions, and a function $\mathsf{pt}(e)$ that returns the set of regions that expression e may point into.

Flow-insensitive points-to analyses such as [8, 9] can be used to provide the region partitioning and the points-to information, but they require the availability of the entire program. For incomplete programs, the following approaches can be used: 1) type-based points-to information for type-safe languages, with one region for each type and points-to relations according to type declarations; and 2) address-taken points-to information, with one region for each memory location whose address has not been stored in the program, and one region for everything else.

The analysis uses the points-to information to resolve alias queries. An expression e is *disjoint* from a region set *rs*, written $e \,\#\, rs$, if updates in any of the regions in *rs* do not affect value of e. The analysis answers such queries using the points-to information:

$$e \,\#\, rs \quad \text{iff} \quad \forall (*e') \in \mathsf{Mem}(e) \,.\, \mathsf{pt}(e') \cap rs = \varnothing$$

For expression contexts, $\mathcal{E}[e] \,\#\, rs$ means that all of the sub-expressions of $\mathcal{E}[e]$ other than e are disjoint from regions in *rs*: $\mathcal{E}[e] \,\#\, rs$ if and only if $\forall (*e') \in \mathsf{Mem}(\mathcal{E}[e]) - \{e\} \,.\, \mathsf{pt}(e') \cap rs = \varnothing$.

$$ImplicitMiss(e, (S, H, M)) = (e = n) \lor (e = a) \lor (e = \mathcal{E}[*n]) \lor (e = \mathcal{E}[n.f]) \lor$$
$$(e = *e' \land (S \cap \mathsf{pt}(e') = \varnothing))$$

$$Miss(e, (S, H, M)) = e \in M \lor ImplicitMiss(e, (S, H, M))$$

$$Infeasible(S,H,M) = \exists e \in H . Miss(e, (S, H, M))$$

$$Cleanup(S, H, M) = (S, H, M'), \quad \text{where:}$$
$$M' = \{e \mid e \in M \land \neg ImplicitMiss(e, (S, H, M))\}$$

Fig. 2. Helper functions used by the analysis.

4.2 Error Cell Abstraction

The analysis starts from the potential error point, a program assignment, assuming that the assignment has overwritten the last reference to the error cell. The analysis models this cell using a triple of the form (S, H, M), where:

- $S \subseteq \mathsf{Rgn}$ is the conservative set of regions that might hold pointers to the error cell;
- H is a set of expressions that point to the error cell; and
- M is a set of expressions that do not reference the cell.

We refer to H as the hit set, and to M as the miss set, similarly to [4]. The partial ordering for this abstraction is such that $(S_1, H_1, M_1) \sqsubseteq (S_2, H_2, M_2)$ if and only if $S_1 \subseteq S_2$, $H_1 \supseteq H_2$, and $M_1 \supseteq M_2$. The join operation is defined accordingly: $(S_1, H_1, M_1) \sqcup (S_2, H_2, M_2) = (S_1 \cup S_2, H_1 \cap H_2, M_1 \cap M_2)$. Two elements of the dataflow lattice have special meaning: the top element $\top$ indicates that the error cell is always feasible; and the bottom element $\bot$ indicates a contradiction, i.e., that the error cell is not feasible.

Figure 2 shows several helper functions that the analysis uses to reasons about dataflow triples. The function *ImplicitMiss* helps the analysis identify new miss expressions, to which we refer as implicit miss expressions. These include:

- *Numeric constants n.* Addresses manufactured from numeric constants cannot reference the cell, because casting integers into pointers is not allowed;
- *Symbolic addresses a.* Leaks can occur only for heap cells, which cannot be referenced by symbolic addresses;
- *Invalid expressions $\mathcal{E}[*n]$ and $\mathcal{E}[n.f]$.* These include null pointer dereferences and null field accesses. Evaluating such expressions would cause run-time errors, so they are not valid references to the error cell;
- *Lvalue expressions that represent regions outside of the region set S.* If $S \cap \mathsf{pt}(e) = \varnothing$, then $*e$ is an implicit miss expression.

The function *Infeasible* identifies infeasible dataflow facts: a dataflow fact d is infeasible if there is an expression that hits and misses the error cell. Such facts represent impossible states of the error cell; they are equivalent to the bottom value $\bot$, and correspond to contradictions in our framework.

To keep the dataflow facts as small as possible and avoid redundancy in the abstraction, the analysis uses a function *Cleanup* to remove implicit miss expressions from dataflow triples. This function always yields an abstraction higher up in our lattice (i.e., more conservative dataflow information). Hence, using *Cleanup* after applying a transfer function does not affect the correctness of the analysis.

4.3 Intra-Procedural Analysis

For each abstraction (S, H, M) that describes the error cell after a statement, the analysis computes an abstraction (S', H', M') that describes the known facts about the error cell before the statement. Hence, the analysis computes an over-approximation of the state before each statement (as opposed to weakest preconditions, which are under-approximations). We refer to the state after the statement as the post-state, and the state before the statement as the pre-state.

The overall analysis uses a worklist algorithm to perform the backward dataflow computation. The analysis is initiated at assignment and a few other program points, as discussed later in Section 4.5. Then, the information is propagated backward. When the analysis reaches a contradiction ($\bot$) on all backward paths, the error is disproved. When the abstraction of the error cell is $\top$, or when the analysis reaches the entry point of a program fragment, the analysis reports a potential violation. The analysis of each statement is presented below.

Analysis assignments: $*e_0 \leftarrow e_1$. Given a dataflow triple (S, H, M) that describes the post-state of the error cell, the analysis computes a new dataflow fact that describes the pre-state of the cell. The transfer function for assignments is:

$$[\![*e_0 \leftarrow e_1]\!](S, H, M) \;=\; \begin{cases} \bot & \text{if } \textit{Infeasible}(S', H', M') \\ \textit{Cleanup}(S', H', M') & \text{otherwise} \end{cases}$$

$$\text{where} \quad \begin{aligned} H', M' &\text{ are derived using the rules in Figure 3} \\ S' &= S \cup \mathsf{pt}(e_0) \end{aligned}$$

The region set S always grows since the post-state gives no information about the old value of the written location. The analysis must conservatively assume that $*e_0$ might reference the cell in the pre-state (we discuss how to improve this in Section 4.8).

To keep the analysis rules succinct, we write e^+ and e^- for $e \in H$ and $e \in M$ (hit/miss expressions in the post-state); and ^+e and ^-e for $e \in H'$ and $e \in M'$ (hit/miss expressions in the pre-state). We also write e^{--} to denote that $\textit{Miss}(e, (S, H, M))$. The set w is the set of regions potentially written by the assignment: $w = \mathsf{pt}(e_0)$. Hence, an expression has the same value before and after the statement if it is disjoint from w.

The inference rules in Figure 3 are used to derive hit and miss expressions in the pre-state. If the premises hold in the post-state, then the conclusion holds in the pre-state. Each rule is implemented by iterating over expressions in H and M, matching them against the rightmost expression in the premise. If the rule applies (i.e., all other premises hold), then the expression in the conclusion is added to H' or M'. Our implementation also checks for contradictions as new expressions are generated in the pre-state, returning $\bot$ as soon as the first contradiction occurs.

$$\frac{e \mathrel{\#} w \quad e^{sgn}}{{}^{sgn}e} \quad [\textsc{Filter}1] \qquad\qquad \frac{e_1 = *e \quad e \mathrel{\#} w \quad e_1^{sgn}}{{}^{sgn}e_1} \quad [\textsc{Filter}4]$$

$$\frac{e_1 \mathrel{\#} w \quad e_1^{--} \quad e \mathrel{\#} w \quad (*e)^{+}}{{}^{+}(*e)} \quad [\textsc{Filter}2] \qquad \frac{\mathcal{E}[*e_0] \mathrel{\#} w \quad \mathcal{E}[*e_0]^{sgn}}{{}^{sgn}\mathcal{E}[e_1]} \quad [\textsc{Subst}1]$$

$$\frac{e_1 \mathrel{\#} w \quad e_1^{+} \quad e \mathrel{\#} w \quad (*e)^{-}}{{}^{-}(*e)} \quad [\textsc{Filter}3] \qquad \frac{e_0 \mathrel{\#} w \quad (*e_0)^{--}}{{}^{-}e_1} \quad [\textsc{Subst}2]$$

Fig. 3. Analysis rules for an assignment $*e_0 \leftarrow e_1$. The sign *sgn* is either $+$ or $-$. The set $w = \mathsf{pt}(e_0)$ is the conservative set of written regions.

The first four rules filter out existing expressions from H and M if the assignment might invalidate them. Clearly, each expression disjoint from w will maintain its hit or miss status (rule [$\textsc{Filter}1$]). The other filtering rules are less obvious. They attempt to preserve expressions that fail this simple test. Consider rule [$\textsc{Filter}2$]. If expression e_1 is disjoint from w and e_1 misses in the post-state, then it also misses it in the pre-state. Hence, the assignment writes a value that doesn't reference the cell. Therefore, each expression $*e$ that hits the cell in the post-state must necessarily be a location different than the one written by the assignment, provided that its address e is not affected by the assignment. Therefore, if all these premises are met, $*e$ has the same value in the pre-state, so it will hit the error cell before the statement. Rule [$\textsc{Filter}3$] is symmetric. Rule [$\textsc{Filter}4$] indicates that the RHS expression e_1 can be preserved if its address is not changed by the assignment. The reason is that, if e_1 happens to be written, it is updated with its old value.

Rule [$\textsc{Subst}1$] derives new hit and miss expressions in the pre-state by substitution. If $e_0 \mathrel{\#} w$, then $*e_0$ in the post-state has the same value as e_1 in the pre-state. Hence, if an expression $e = \mathcal{E}[*e_0]$ hits (misses) in the post-state, we can substitute e_1 for $*e_0$ to derive an expression $\mathcal{E}[e_1]$ that hits (misses) in the pre-state. For this to be safe, the expression $\mathcal{E}[*e_0]$ must be disjoint from w. The last rule, [$\textsc{Subst}2$] is similar to substitution, but for implicit misses and for a simple context $\mathcal{E} = [\cdot]$.

Example. Consider a triple $(\{r_y\}, \{*a_y\}, \{\})$ describing the error cell in the post-state. Here, a_y is the symbolic address of variable y, and r_y is the region that contains y. Let $x \leftarrow y$ be the assignment to analyze, represented in our formulation as $*a_x \leftarrow *a_y$. Assume that variables x and y belong to different regions, so all necessary disjointness conditions are met. By rule [$\textsc{Subst}2$] we get $^{-}(*a_y)$, because $*a_x$ is an implicit miss in the post-state. By rule [$\textsc{Filter}1$], $^{+}(*a_y)$ also holds. Hence, a contradiction occurs.

Analysis of allocations: $*e_0 \leftarrow \texttt{malloc}$. The analysis tries to determine if the error cell has been allocated at this site. First, if $(*e_0)^{+}$ and some expression unaliased to $*e_0$ also references the cell, then a contradiction occurs. Second, if there is evidence that the error cell has not been allocated at this site, it proceeds past this statement, treating the allocation as a nullification $*e_0 \leftarrow 0$. Note that the nullification automatically causes

a contradiction when a field of $*e_0$ hits the error cell in the post-state. Otherwise, if none of the above conditions are met, the analysis conservatively stops and returns $\top$, signaling that the leak might be feasible and the error cell might be allocated at this site. The transfer function is defined as follows:

$$\llbracket *e_0 \leftarrow \texttt{malloc} \rrbracket(d) = \begin{cases} \bot & \text{if } \textit{UnaliasedHit} \wedge (*e_0 \in H \wedge e_0 \# w) \\ \llbracket *e_0 \leftarrow 0 \rrbracket(d) & \text{if } \textit{UnaliasedHit} \vee (\textit{Miss}(*e_0, d) \wedge e_0 \# w) \\ \top & \text{otherwise} \end{cases}$$

where $d = (S, H, M)$, $w = \mathsf{pt}(e_0)$, and $\textit{UnaliasedHit} = \exists (*e) \in H : \mathsf{pt}(e) \cap w = \varnothing$. Here, $\bot$ indicates a contradiction, and $\top$ indicates a potential leak for a cell allocated at this site.

Analysis of deallocations: $\texttt{free}(e)$. When the analysis reaches a deallocation $\texttt{free}(e)$, a contradiction occurs if e references the error cell. In other words, losing the last reference of a cell that has been freed is not an error. Otherwise, the analysis learns that e misses in the pre-state and keeps the rest of the state unchanged. The algorithm is:

$$\llbracket \texttt{free}(e) \rrbracket(S, H, M) = \begin{cases} \bot & \text{if } e \in H \\ (S, H, M) & \text{if } \textit{Miss}(e, (S, H, M)) \\ (S, H, M \cup \{e\}) & \text{otherwise} \end{cases}$$

Analysis of conditions: $\texttt{cond}(e_0 \equiv e_1)$. For conditions, the analysis knows that the (in)equality has succeeded in the post-state. It uses this information to derive new hit and miss expressions in the pre-state, as indicated by the following rules:

$\texttt{cond}(e_0 = e_1)$				$\texttt{cond}(e_0 \neq e_1)$	
$\dfrac{e_0^+}{^+e_1}$	$\dfrac{e_1^+}{^+e_0}$	$\dfrac{e_0^{--}}{^-e_1}$	$\dfrac{e_1^{--}}{^-e_0}$	$\dfrac{e_0^+}{^-e_1}$	$\dfrac{e_1^+}{^-e_0}$

If H' and M' are the new hit and miss expressions derived using the above rules, $M'' = M \cup M'$, and $H'' = H \cup H'$, then the transfer function for conditions is:

$$\llbracket \texttt{cond}(e_0 \equiv e_1) \rrbracket(S, H, M) = \begin{cases} \bot & \text{if } \textit{Infeasible}(S, H'', M'') \\ \textit{Cleanup}(S, H'', M'') & \text{otherwise} \end{cases}$$

4.4 Inter-Procedural Analysis

The inter-procedural analysis follows the general structure of the worklist inter-procedural analysis algorithm proposed by Sharir and Pnueli [10]. Information is propagated from the points after procedure calls to the corresponding procedure exit points, and from procedure entry points to the corresponding points before the call. The analysis uses backward procedure summaries to cache previous analysis results. The entire inter-procedural worklist algorithm is presented in Appendix B.

The analysis uses two functions, *Map* and *Unmap*, to account for the necessary changes in the analysis information when crossing procedure boundaries, such as assignments of actuals to formals, or to model the scopes of local variables. The analysis uses *Map* when moving from the caller into the callee space, and uses *Unmap* when moving back into the caller space. The mapping process is performed right before $\texttt{return}$, and the unmapping right after $\texttt{enter}$. Each pair is discussed below.
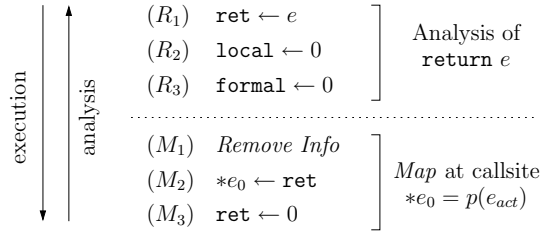
$$
\begin{array}{lll}
(R_1) & \texttt{ret} \leftarrow e \\
(R_2) & \texttt{local} \leftarrow 0 \\
(R_3) & \texttt{formal} \leftarrow 0
\end{array}
\quad
\begin{array}{l}
\text{Analysis of} \\
\texttt{return } e
\end{array}
$$

$$
\begin{array}{lll}
(M_1) & \textit{Remove Info} \\
(M_2) & *e_0 \leftarrow \texttt{ret} \\
(M_3) & \texttt{ret} \leftarrow 0
\end{array}
\quad
\begin{array}{l}
\textit{Map} \text{ at callsite} \\
*e_0 = p(e_{act})
\end{array}
$$

Fig. 4. Analysis for *Map* and $\texttt{return}(e)$.

Analysis for Map and $\texttt{return}(e)$. For simplicity, consider a procedure p with one formal parameter $\texttt{formal}$ and one local variable $\texttt{local}$. Let $\texttt{ret}$ be a special variable that models the return value. Variables $\texttt{local}$, $\texttt{formal}$, and $\texttt{ret}$ are represented in this section using C-style notation instead of the normalized representation for expressions. Consider a call-site that invokes p with an actual argument e_{act}.

The call-site mapping process and the analysis of the return of p are described in Figure 4. The execution proceeds from top to bottom, and the analysis works in the reverse order. We explain the actions in this diagram in the execution order. Each node $\texttt{return}(e)$ is modeled as a sequence that assigns the returned expression to $\texttt{ret}$, and then nullifies all local and formal variables, showing that they go out of scope. The transfer function of $\texttt{return}$ is the composition of the transfer functions of these assignments, in reverse order.

The mapping function *Map* takes place at the point right before a call site $*e_0 \leftarrow p(e_{act})$. The mapping process assigns the return variable $\texttt{ret}$ to expression $*e_0$ and then nullifies the return variable. The mapping process also removes from H and M all the expressions that involve locals or parameters of the caller, keeping only information relevant to the callee. This is shown by the *Remove Info* step right before the analysis moves into the callee's space.

Analysis for Unmap and $\texttt{enter}$. When the analysis reaches the entry of a procedure, it moves back to the caller space, to the point right before the call. The analysis of the $\texttt{enter}$ statement and the unmapping process are described in Figure 5. We use a special operation $\texttt{scope}(s)$ to indicate that symbol s enters its scope. For the analysis, which is reversed, $\texttt{scope}(s)$ indicates that s goes out of scope. The transfer function of $\texttt{scope}(s)$ has two possible outcomes: it yields a contradiction if s occurs in one of the hit expressions; otherwise, it removes all hit and miss expressions that refer to s. The analysis of $\texttt{enter}$ is modeled using a $\texttt{scope}$ operation for each of its locals.

The unmap process accounts for the assignments of actuals to formals and for restoring part of the information that *Map* has filtered out. One complication that arises for recursive functions is that expression e_{act} in the caller might refer to variable $\texttt{formal}$. A direct assignment $\texttt{formal} \leftarrow e_{act}$ would then talk about two different instances of the same variable $\texttt{formal}$, the one of the caller and the one of the callee. This problem can be solved using a $\texttt{shadow}$ variable to perform the assignment in two steps.
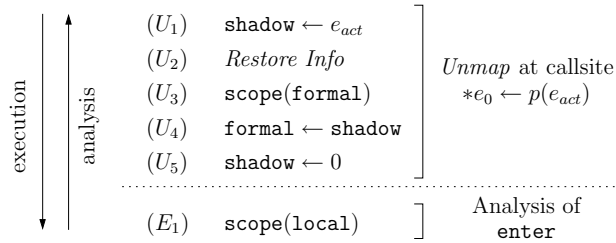
$$
\begin{array}{lll}
(U_1) & \texttt{shadow} \leftarrow e_{act} & \\
(U_2) & \textit{Restore Info} & \textit{Unmap at callsite} \\
(U_3) & \texttt{scope(formal)} & *e_0 \leftarrow p(e_{act}) \\
(U_4) & \texttt{formal} \leftarrow \texttt{shadow} & \\
(U_5) & \texttt{shadow} \leftarrow 0 & \\
\\
(E_1) & \texttt{scope(local)} & \begin{array}{l}\text{Analysis of} \\ \texttt{enter}\end{array}
\end{array}
$$

execution   *analysis*

Fig. 5. Analysis for *Unmap* and `enter`.

In execution order, first assign the actual to the shadow, then move to the callee space and assign the shadow to the formal. In between the two `shadow` assignments, the formal enters its scope (and goes out of scope for the analysis). The analysis also restores expressions that *Map* has filtered out, provided that they cannot be modified by the callee. For instance, local variables whose addresses have not been taken can be safely restored. In general, restoring expressions requires knowledge about the locations being modified by the callee, i.e., MOD information.

4.5 Leak Probing Points

The last piece that completes the analysis is defining the initialization points. The analysis issues one query for each assignment, to determine if the assignment might leak memory. We refer to each leak query as a *leak probe*. For each assignment $*e_0 \leftarrow e_1$ the analysis builds a dataflow triple $(\texttt{pt}(e_0), \{*e_0\}, \{e_1\})$ to describe an error cell referenced by e_0, but not by e_1. In many cases, e_1 is an implicit miss and can be omitted from the miss set. The analysis then asks whether this triple might hold at the point right before the assignment.

In addition to assignments, the analysis issues leak probes at the following points:

1. *Allocations*: each $*e_0 \leftarrow \texttt{malloc}$ is probed as $*e_0 \leftarrow 0$.
2. *Deallocations*: for each $\texttt{free}(e)$, the analysis issues probes that correspond to a sequence of assignments $*(e.f) \leftarrow 0$, for each field f of e. This checks for leaks caused by freeing a cell that holds the last reference to another cell.
3. *Locals and formals*: at return points, it issues a probe for each local variable and formal parameter. These correspond to the nullifications (R_2) and (R_3) in Figure 4. The assignment to `ret` needs not be checked.
4. *Assigned returns*: for each call $*e_0 \leftarrow p(...)$, it issues a probe that corresponds to the assignment (M_2) in Figure 4. Note that this probe will immediately propagate into the callee.
5. *Leaked returns:* Although in our language return values are always assigned, in C function calls are not required to do so. In that case, the assignment (M_2) is missing and the returned value might be leaked. The analysis uses a probe that corresponds to the nullification of `ret` (M_3), to check for leaked returns.

4.6 Formal Framework and Soundness

This section summarizes the soundness result for the transfer functions in our analysis. This result states that the abstract semantics of statements are sound with respect to the concrete execution of the program, provided that the points-to information is sound. The definitions below define the notion of soundness for the points-to information, and describe the abstraction function. Then, the soundness theorem is stated. In these definitions, L denotes the set of all memory addresses, including symbolic addresses $a \in A$ and heap addresses.

Definition 1 (Points-to Soundness). *Let* Rgn *be the finite set of memory regions. A points-to abstraction* $\mathsf{pt} : E \to 2^{\mathsf{Rgn}}$ *is a sound approximation of a concrete store* σ, *with witness mapping* $\mu : L \to \mathsf{Rgn}$ *from locations to regions, written* $\sigma \models_\mu \mathsf{pt}$, *if:* $\forall e \in E \,.\, (e, \sigma) \to l \,\wedge\, l \in L \,\Rightarrow\, \mu(l) \in \mathsf{pt}(e)$.

Definition 2 (Abstraction Function). *Let* σ *be a store,* $l \in L - A$ *a heap location in* σ, $l \in dom(\sigma)$, *and* μ *a mapping from locations to regions. A dataflow fact* d *is a conservative approximation of* l *in* σ *with respect to* μ, *written* $(l, \sigma) \models_\mu d$, *if* $d = \top$, *or* $d = (S, H, M)$ *and:*

1. $\forall e \in H \,.\, (e, \sigma) \to l$
2. $\forall e \in M \,.\, (e, \sigma) \to l \Rightarrow l' \neq l$
3. $\forall l' \in L \,.\, \sigma(l') = l \Rightarrow \mu(l') \in S$

Definition 3 (Dataflow Validity). *Let* s *be a statement,* d *and* d' *two dataflow facts, and* pt *the points-to information. The triple* $\{d\}\ s\ \{d'\}$ *is valid relative to* pt, *written* $\models_{\mathsf{pt}} \{d\}\ s\ \{d'\}$, *if for any pre- and post-stores for which* pt *is sound, and for any location that is approximated by* d' *in the post-store, the location is then approximated by* d *in the pre-store:* $\forall l, \sigma, \sigma', \mu \,.\, (\ \sigma' \models_\mu \mathsf{pt} \,\wedge\, \sigma \models_\mu \mathsf{pt} \,\wedge\, (l, \sigma') \models_\mu d' \,\wedge\, (s, \sigma) \to \sigma'\ ) \,\Rightarrow\, (l, \sigma) \models_\mu d$.

From this definition, it follows that $\{\top\}\ s\ \{d'\}$ is valid for any d'. Also, $\bot$ can never approximate the error cell: $\forall l, \sigma, \mu \,.\, (l, \sigma) \not\models_\mu \bot$. Therefore, $\{\bot\}\ s\ \{d'\}$ is never valid.

Theorem 1 (Transfer Function Soundness). *For any assignment, malloc, free, or condition statement* s, *if* d' *is a dataflow fact after* s, *and* pt *is the points-to information, then the dataflow fact* $d = [\![s]\!](d')$ *that the transfer function computes before* s *is sound relative to the points-to information* pt: $\models_{\mathsf{pt}} \{\ [\![s]\!](d')\ \}\ s\ \{\ d'\ \}$.

Appendix C provides the proof of the above theorem.

4.7 Termination

The leak detection analysis is guaranteed to terminate, because of three reasons. First, region sets are bounded by the finite set Rgn. Second, hit and miss expressions can only shrink during the analysis. Although the set of expressions in unbounded, these sets have finite size when a node is reached for the first time; after that, they only decrease. Third, the transfer functions are monotonic. It is easy to see that larger hit and miss sets will cause the inference rules to derive more facts in the pre-state.

4.8 Extensions

We propose several extensions that improve the precision of the basic analysis:

- *Diminish the region set increase via points-to information.* As mentioned earlier, the region set S grows during the analysis because the state after an assignment doesn't give information about the old value of the location being written. The analysis conservatively assumes that the old value hit the error cell in the pre-state. This is overly conservative, especially in the case of assignments of integers or other non-pointer values. The analysis can use points-to information to avoid this. If $*e_0$ is the LHS of the assignment being probed, and $*e$ is the LHS of the currently analyzed assignment, the analysis can determine that $^-(*e)$ if $*e$ and $*e_0$ are unaliased: $\mathsf{pt}(e_0) \cap \mathsf{pt}(e) = \varnothing$. In this case, $\mathsf{pt}(e_0)$ is not added to S.
- *Enable region set removal via strong updates.* A second improvement is to augment the abstraction so that the analysis also removes regions from S. For this, we tag each region in S with a program expression $*e$, or with a top value. An expression tag shows that the region contains at most one reference to the cell, and that reference, if present, is $*e$. Top indicates imprecision. When the analysis identifies a region r that contains at most one reference $*e$, and the analysis rules imply $^-(*e)$, then it can safely remove r from S in the pre-state.
- *Separated abstraction*: We propose a variation of the analysis where the analysis computes more than one triple (S, H, M) per program point. Two triples are merged only if their subset of regions that contain the hit expressions $S \cap H$ is the same. Otherwise, the triples are maintained separated. Because of less frequent merges, the analysis becomes more precise.

5 Experiments

We have implemented the algorithms presented in this paper in an analysis system developed in our group, CRYSTAL. All of the C constructs are translated into an intermediate representation that is very similar to the normalized representation from Section 4. Therefore, the implementation closely follows the formal presentation; at the same time, it handles the complexity of C. The leak detector uses the extensions discussed in the previous section. The results were collected on a 3Ghz Pentium 4 machine running Red Hat Enterprise Linux 4.

Heap manipulation benchmarks. We have tested our leak analyzer on several small heap manipulation routines. For these experiments, we only consider programs written in the type-safe subset of C from Section 4. The benchmarks include iterative and recursive versions of standard linked list operations (insert, delete, reverse, merge) for singly-linked and doubly-linked lists; two versions of the Deutsch-Schorr-Waite pointer reversal algorithm, for lists and for trees; and AVL tree manipulations. The singly-linked list manipulations are a representative subset of those from [1] and [11]. The doubly-linked list implementations are part of the Gnome's GLib library [1].

---

[1] ftp://ftp.gtk.org/pub/gtk

| Program | Probes | | Trace Length | | Abstractions |
	Total	Warn / Leak	Median	Max	per point
`si-create`	5	0 / 0	2	6	1.0
`si-delete`	11	0 / 0	10	29	1.4
`si-insert`	13	1 / 0	2	130	1.8
`si-reverse`	9	0 / 0	1	6	1.0
`si-rotate`	6	1 / 1	2	10	1.1
`si-merge`	16	0 / 0	12	200	1.7
`sr-append`	11	0 / 0	2	57	2.0
`sr-insert`	11	2 / 0	13	114	2.7
`sr-reverse`	11	0 / 0	1	82	1.6
`sr-rev-leak`	13	1 / 1	1	95	2.8
`di-delete`	9	2 / 2	6	89	2.7
`di-prepend`	7	1 / 1	7	15	1.1
`di-reverse`	7	0 / 0	1	12	1.2
`di-merge`	19	0 / 0	1	24	1.2
`avl-rotate`	6	0 / 0	1	2	1.0
`avl-balance`	6	1 / 0	22	803	1.3
`avl-insert`	31	2 / 0	2	1627	3.9
`dsw-list`	14	0 / 0	2	91	2.1
`dsw-tree`	18	1 / 0	15	1367	8.6

Fig. 6. Experiments on recursive structure manipulations.

We have experimented with each procedure in isolation, or with small groups of procedures when some of them called others. Warnings were reported when the analysis reached the entry of a procedure that is never called. Recursive functions have been wrapped into non-recursive functions. We use a type-based points-to region partitioning and points-to analysis. All of the programs were analyzed in less than one second.

Figure 6 shows analysis statistics and results for the small benchmarks. The meaning of the two-letter prefix is as follows: `s` means "singly" and `d` means "doubly"; `i` means "iterative" and `r` means "recursive". The first group of columns presents the total number of probes, the number of warned probes, and the number of actual errors. The analysis assumes any possible inputs, including malformed inputs. The tool has determined that no leaks occur for about half of the programs. It has found the memory leak in the buggy version of the recursive list reversal program `sr-rev-leak` from [11]. Some procedures leak memory if the inputs are malformed: `si-rotate` leaks when its second argument doesn't point to the last element, as the function expects; and `di-delete` and `di-prepend` leak if the input doesn't satisfy the doubly-linked list invariant. We consider these warnings legitimate. The remaining ones are false positives and are due to imprecision in the analysis.

The last three columns show analysis statistics: the median and maximum length of reverse traces (measured as applications of transfer functions), and the average number of abstraction triples per program point. The trace statistics indicate that for most of the probes contradictions show up quickly, but there are a few probes that require sig-

Benchmark	Size KLOC	Time (sec)	Probes Total	Aband.	Warn	Bugs
`ammp`	13.2	6.95s	1550	123	24	20
`art`	1.2	1.20s	32	16	1	1
`bzip2`	4.6	3.36s	108	62	2	1
`crafty`	19.4	27.71s	1493	1174	0	0
`equake`	1.5	1.50s	55	12	0	0
`gap`	59.4	108.66s	13517	6685	1	0
`gzip`	7.7	5.11s	489	173	3	1
`mcf`	1.9	4.21s	392	64	0	0
`mesa`	50.2	34.50s	5037	956	2	2
`parser`	10.9	19.36s	2859	1021	0	0
`perlbmk`	61.8	340.07s	25151	15801	1	1
`twolf`	19.7	20.29s	2526	1105	0	0
`vortex`	52.6	304.59s	9448	7421	26	0
`vpr`	16.9	15.14s	1216	530	0	0

Fig. 7. Experiments on the SPEC2000 benchmarks.

nificantly more work, especially for complex pointer manipulations. The last column shows that the analysis usually creates few (around 2) abstractions per program point.

Larger benchmarks. We have also experimented with this tool on larger programs from the SPEC200 benchmark suite[2]. To make the tool useful for larger programs, we use several heuristics that cut down the amount of backward exploration: there is a limit on the number of transfer functions per probe (currently 500), and a limit on the size of the region set S (currently 50). When the analysis reaches these limits, it abandons the probe. The analysis also limits the amount of inter-procedural exploration by ignoring callees more than one level deep, but allows tracking the error cell back into the callers. The tool uses type-based points-to information, which is unsound for type-unsafe C programs. Other sources of unsoundness include ignoring library functions other than memory allocators; and the unsound treatment of arrays, where the analysis does not probe assignments of array elements. The analysis cannot reason about array index dependencies, so array element probes would otherwise lead to false warnings.

Figure 7 shows the results. The first column shows the sizes of these programs and the second column the analysis times. The remaining columns show the analysis results: the number of probes explored; the number of probes abandoned because of the cut-off; the number of probes warned; and the number of actual bugs found. Warned probes are those for which the analysis reaches a validating allocation site for the error cell.

Most of the leaks found (e.g., in `ammp` and `perlbmk`) are situations where the application doesn't free memory upon returning on error handlers. Interestingly, this happens even for out-of-memory error handlers: the application allocates several buffers and if allocation fails for one, none of the others are freed. In `art` a function actually forgets to deallocate a local buffer on the normal return path. The main reasons for false

---

[2] We omit `gcc` because all warnings referred to data allocated via `alloca`.

warnings are the imprecision of the type-based region approximation; the use of pointer arithmetic; and the fact that programs such as `vortex` use complex, custom memory management.

Overall, we again find that the length distribution of backward traces is uneven. In most of the cases, the analysis can quickly identify a contradiction, within just a few lines of code. However, a few points generate very long backward traces, along which the analyses loses precision, and therefore becomes unlikely to produce meaningful results. The cutoff helps eliminate such cases from our reports.

6   Related Work

Manevich et al. [12] propose a backward flow analysis with the goal of tracing back null pointer errors, and disprove such errors. Although our analysis is similar in spirit to theirs, the analysis of memory leaks in heap structures is a more challenging problem than that of distinguishing between null and non-null values.

A related line of research has explored demand-driven inter-procedural analyses and frameworks [13–16]. Given a dataflow fact d at a program point p, a demand-driven flow analysis explores backward program paths starting from p with the goal of determining whether d is part of the forward dataflow solution at p. Our backward analysis is, in fact, a demand-driven analysis, although we are not interested in an answer to a forward analysis (since we don't have one), but rather with respect to the program semantics. Furthermore, to the best of our knowledge, the analysis in this paper is the first reverse, demand-driven heap analysis.

Several static leak detection analyses have been recently proposed. Heine and Lam [3] use a notion of pointer ownership to describe those variables responsible for freeing heap cells, and formulate the analysis as an ownership constraint system. In our previous work [4], we used a shape analysis with local reasoning about single heap cells to detect memory leaks and accesses through dangling pointers. Xie and Aiken [5] reduce the problem of memory leak detection to a Boolean satisfiability problem, and then use a SAT-solver to identify potential errors. Their analysis is path- and context-sensitive, but uses unsound techniques to handle recursion and loops. Dor et al. [1] use TVLA, a shape analysis tool based on 3-valued logic, to prove the absence of memory leaks and other memory errors in several list manipulation programs. Their analysis verifies these programs successfully, but is intra-procedural and cannot be applied to recursive and multi-procedure programs. Of these analyses, [3, 4] target referencing leaks; and [1, 5] target reachability leaks. We are not aware of analyses that can detect liveness memory leaks. Compared to our work, the above approaches cannot answer memory leak queries in a demand-driven fashion; and cannot reason about the absence of errors for incomplete programs.

Shape analyses [17, 7, 4] have been proposed with the goal of being able to distinguish, for instance, between cyclic and acyclic heap structures. These are all forward, exhaustive analyses. In contrast, the heap analysis in this paper is a reverse, demand-driven heap analysis.

The leak detection analysis in this paper uses an abstraction similar to the one that we developed in our previous work on shape analysis [4], where the algorithm analyzes

a single heap cell at a time. This is a good match to memory leak detection by contradiction, because the analysis needs to reason about one single cell, the error cell. There two main differences between these analyses. First, the leak analysis in this paper is not aimed at computing shapes or precise reference counts. Therefore, the analysis uses a simpler abstraction (without reference counts), and doesn't require bifurcation. Second, the analysis is backwards. Analyzing the state of a heap cell in the reverse direction is non-trivial and less intuitive than the forward analysis.

Finally, dynamic memory leak detector tools such as Purify [18] or SWAT [19] instrument the program to detect errors at run-time. Dynamic tools will miss errors that do not happen in that run; in particular, they will miss errors that only occur in rarely executed code fragments.

7   Conclusions

We have presented a new approach to memory leak detection where errors are disproved by contradicting their presence. To determine whether a memory leak can occur at a program point, the analysis uses a reverse inter-procedural flow analysis to disprove its negation. We have used this approach to analyze a set of complex list manipulation routines in isolation. We have also used this approach in an error-detection tool and found several memory leaks in the SPEC benchmarks.

References

1. Dor, N., Rodeh, M., Sagiv, M.: Checking cleanness in linked lists. In: Proceedings of the 8th International Static Analysis Symposium, Santa Barbara, CA (2000)
2. Shaham, R., Kolodner, E.K., Sagiv, M.: Automatic removal of array memory leaks in java. In: Proceedings of the 2000 International Conference on Compiler Construction, Berlin, Germany (2000)
3. Heine, D., Lam, M.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: Proceedings of the SIGPLAN '03 Conference on Program Language Design and Implementation, San Diego, CA (2003)
4. Hackett, B., Rugina, R.: Shape analysis with tracked locations. In: Proceedings of the 32th Annual ACM Symposium on the Principles of Programming Languages, Long Beach, CA (2005)
5. Xie, Y., Aiken, A.: Context- and path-sensitive memory leak detection. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal (2005)
6. Heine, D., Lam, M.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: Proceedings of PLDI, San Diego, CA (2003)
7. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002)
8. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, FL (1996)
9. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)

10. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In Muchnick, S., Jones, N., eds.: Program Flow Analysis: Theory and Applications. Prentice Hall Inc (1981)

11. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Proceedings of the 2001 International Conference on Compiler Construction, Genova, Italy (2001)

12. Manevich, R., Sridharan, M., Adams, S., Das, M., Yang, Z.: PSE: Explaining program failures via postmortem static analysis. In: Proceedings of the ACM SIGSOFT '99 Symposium on the Foundations of Software Engineering, Newport Beach, CA (2002)

13. Strom, R., Yellin, D.: Extending typestate checking using conditional liveness analysis. IEEE Transactions on Software Engineering **19**(5) (1993) 478–485

14. Duesterwald, E., Gupta, R., Soffa, M.: Demand-driven computation of interprocedural data flow. In: Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages, San Francisco, CA (1995)

15. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proceedings of the ACM Symposium on the Foundations of Software Engineering, Washington, DC (1995)

16. Sagiv, S., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science **167**(1&2) (1996) 131–170

17. Ghiya, R., Hendren, L.: Is is a tree, a DAG or a cyclic graph? a shape analysis for heap-directed pointers in C. In: Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, FL (1996)

18. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: Proceedings of the 1992 Winter Usenix Conference. (1992)

19. Hauswirth, M., Chilimbi, T.: Low-overhead memory leak detection using adaptive statistical profiling. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA (2004)

A Language Semantics

The language semantics are defined using the following domains:

Numeric constants	$n \in \mathbb{Z}$	Locations	$l \in L = A + (C \times F)$
Symbolic addresses	$a \in A$	Values	$v \in V = \mathbb{Z} + L$
Heap cell addresses	$c \in C$	Stores	$\sigma \in \Sigma = L \rightharpoonup V$
Structure fields	$f \in F$		

The following rules define the evaluation of expressions and statements:

$$\frac{}{(n,\sigma) \to n} \qquad \frac{}{(a,\sigma) \to a} \qquad \frac{(e,\sigma) \to l \quad l \in dom(\sigma)}{(*e,\sigma) \to \sigma(l)} \qquad \frac{(e,\sigma) \to (c,f_1) \quad c \in C}{(e.f,\sigma) \to (c,f)} \qquad \frac{(e_0,\sigma) \to v_0 \quad (e_1,\sigma) \to v_1 \quad v_0,v_1 \in \mathbb{Z} \quad v = v_0 \oplus v_1}{(e_0 \oplus e_1,\sigma) \to v}$$

$$\frac{(e_0,\sigma) \to l \quad (e_1,\sigma) \to v \quad l \in dom(\sigma)}{(*e_0 \leftarrow e_1,\sigma) \to \sigma[l \mapsto v]} \qquad \frac{(e_0,\sigma) \to v_0 \quad (e_1,\sigma) \to v_1 \quad v_0 \equiv v_1}{(\texttt{cond}(e_0 \equiv e_1),\sigma) \to \sigma}$$

$$\frac{(e_0,\sigma) \to l \quad l \in dom(\sigma) \quad c \, fresh \quad \sigma' = \sigma[l \mapsto (c,f_1)] \, \cup \, \{(c,f) \mapsto 0\}_{f \in F}}{(*e_0 \leftarrow \texttt{malloc},\sigma) \to \sigma'} \qquad \frac{(e,\sigma) \to (c,f_1) \quad (c,f_1) \in dom(\sigma) \quad \sigma' = \sigma - \{(c,f) \mapsto {}_-\}_{f \in F}}{(\texttt{free}(e),\sigma) \to \sigma'}$$

B   Demand-Driven Inter-Procedural Analysis Algorithm

Figure 8 shows the entire inter-procedural demand-driven dataflow analysis-by-contradiction algorithm. Each procedure in the program is represented using a control-flow graph whose nodes $n \in N$ are program statements. For each CFG node n, $pred(n)$ represents the predecessors of n in the graph. Node n_e^p is the entry node of procedure p, and node n_x^p is the exit (i.e., return) node. Given a dataflow fact d_0 that describes the error cell and a control-flow graph node n_0, the algorithm returns "Success" if it can determine that a leak cannot occur at point n_0; otherwise, it returns "Potential Leak".

Separable Abstractions.   The algorithm is formulated to work with *separable dataflow abstractions*. A separable abstraction is a map $D = D_i \rightarrow D_s$, where D_i is a finite set called the *index domain*, and $(D_s, \sqcup, \sqsubseteq, \bot, \top)$ is a lattice called the *secondary domain*. Dataflow facts $d \in D_i \rightarrow D_s$ are represented using association lists, and components with bottom secondary values are omitted. For instance, $d = \{(i, s)\}$ stands for $\lambda i'$. if $(i' = i)$ then s else $\bot$. The ordering over D is the pointwise ordering of functions. For a lattice element $d \in D$, each pair $(i, d(i))$ is called a *component* of the abstraction d. Transfer functions are expressed component-wise: $[\![n]\!]^\bullet : (D_i \times D_s) \rightarrow D$. A standard (non-separable) dataflow abstraction is represented as a separable abstraction with one single component. The basic leak analysis uses a non-separable abstraction; and one of the extensions in Section 4.8 uses a separable abstraction to avoid merging dataflow facts and improve the analysis precision.

Worklist Algorithm.   The analysis uses a worklist algorithm. Each worklist element is a single component (not an entire abstraction) and each procedure context c is an index from D_i. There is an additional context *none*, explained below. The set $C = D_i \cup \{none\}$ denotes all possible contexts. The backward propagation stores dataflow facts in the result function $R : N \rightarrow C \rightarrow D$. For a node n and a context c, the value of $R(n)(c) \in D$ is the dataflow fact computed *after* node n in a context c of the enclosing procedure.

   The analysis uses reverse summaries for procedures. The procedure summary for a context c maps a single component with index $i = c$ at procedure exit to a corresponding dataflow fact d at procedure entry. Dataflow facts whose birth-points occur inside of a function are given the special context *none*. Procedure summaries are not stored in a separate data structure; instead, they are implicitly captured in R: the summary or procedure p for a context c is $R(n_e^p)(c)$.

   When the analysis reaches a procedure entry point, it propagates the information to all of the callers that have requested a result for the current context c. For this, the algorithm uses a *call-site map* $S : P \rightarrow C \rightarrow (N \times C)$. When the analysis of a component reaches the entry of p with context c, it propagates the component to the points indicated by $S(p)(c)$. This contains pairs (n', c') of target points and target contexts at those points. The call site map is set up every time a procedure call is encountered (line 16), and used when reaching procedure entries (line 23). For facts with context *none* (born inside procedures), S contains all possible call sites, as indicated by the initialization at lines 6-7.

   The analysis uses two functions, *Map* and *Unmap*, to account for the necessary changes in the analysis information when crossing procedure boundaries. Function *Map* is used when moving from the caller into the callee space; function *Unmap* is used for moving back into the caller space. *Map* and *Unmap* each take two arguments: the dataflow fact d to process, and the call-site n where the mapping or unmapping takes place.

LEAKANALYSISBYCONTRADICTION($d_0 \in D, n_0 \in N$)

```
 1   for each n ∈ N, c ∈ C, i ∈ D_i
 2        R(n)(c)(i) ← ⊥
 3   for each p ∈ P
 4        S(p)(none) ← Callsites(p) × {none}
 5
 6   W ← ∅
 7   INSERT(d_0, none, n_0)
 8
 9   while W ≠ ∅
10        remove (n, c, i) from W
11        s ← R(n)(c)(i)
12        switch n
13          case call p :
14                c' ← i
15                INSERT(Map({ (i, s) }, n), c', n_x^p)
16                S(p)(c') ← S(p)(c') ∪ { (n, c) }
17                if ( R(n_e^p)(c') ≠ ⊥ )
18                   then for each n' ∈ pred(n)
19                           INSERT(Unmap(R(n_e^p)(c'), n), c, n')
20
21          case n_e^p :
22                if (Callsites(p) = ∅)
23                   then return "Potential Leak"
24                for each (n', c') ∈ S(p)(c)
25                   for each n'' ∈ pred(n')
26                        INSERT(Unmap({ (i, s) }, n'), c', n'')
27
28          case default  :
29                d ← ⟦n⟧•(i, s)
30                if (d = ⊤)
31                   then return "Potential Leak"
32                for each n' ∈ pred(n)
33                     INSERT(d, c, n')
34
35   return "Success"(NoLeak)
```

INSERT($d \in D, c \in C, n \in N$)

```
36   for each i ∈ D_i
37        R(n)(c)(i) ← R(n)(c)(i) ⊔ d(i)
38        if R(n)(c)(i) has changed
39           then W ← W ∪ { (n, c, i) }
```

Fig. 8. Inter-Procedural Demand Analysis by Contradiction

C   Soundness Proof

Lemma 1 (Implicit Miss Expressions). *Let $d = (S, H, M)$ be a dataflow fact,* pt *the points-to information, and e an expression such that ImplicitMiss(e) holds. If l, σ, and μ are such that $l \in L - A$, $\sigma \models_\mu$ pt, $(l, \sigma) \models_\mu d$ and $(e, \sigma) \to v$, then $v \neq l$.*

Proof. Let l, σ, and μ with the above properties. By the definition of *ImplicitMiss*, we must consider the following cases.

- Case $e = n$. Then $v = n$, so $v \neq l$.
- Case $e = a$. Then $v = a$, so $v \neq l$ because $l \notin A$.
- Case $e = \mathcal{E}[*n]$. It is not possible to evaluate e, because the sub-expression $*n$ cannot be evaluated using the semantic rules. Hence, the premise $(e, \sigma) \to v$ is false, and the implication is trivially true.
- Case $e = \mathcal{E}[n.f]$. Similar to the case above, since the sub-expression $n.f$ cannot be evaluated using the semantic rules.
- $e = *e' \land S \cap$ pt$(e') = \varnothing$. Assume $v = l$. The evaluation $(e, \sigma) \to v$ implies $(e', \sigma) \to l'$ and $\sigma(l') = v = l$. The abstraction $(l, \sigma) \models_\mu d$ implies (by condition 3) that $\mu(l') \in S$. Therefore, $\mu(l') \notin$ pt(e'), which contradicts the fact that $(e', \sigma) \to l'$ and $\sigma \models_\mu$ pt. Hence $v \neq l$.

$\square$

Lemma 2 (Miss Expressions). *Let $d = (S, H, M)$ be a dataflow fact,* pt *the points-to information, and e an expression such that Miss(e) holds. If l, σ, and μ are such that $l \in L - A$, $\sigma \models_\mu$ pt, $(l, \sigma) \models_\mu d$ and $(e, \sigma) \to v$, then $v \neq l$.*

Proof. If $e \in M$, the result follows immediately from the relation $(l, \sigma) \models_\mu d$. Otherwise, if *ImplicitMiss(e)*, the result follows from Lemma 1 (Implicit Miss Expressions). $\square$

Lemma 3 (Disjointness). *Let e be an expression,* pt *the points-to information, and w a set of regions. Let σ, σ', and μ such that $\sigma \models_\mu$ pt, $\sigma' \models_\mu$ pt, and $\sigma(l) = \sigma'(l)$ for all $l \in dom(\sigma) \cap dom(\sigma')$ such that $\mu(l) \notin w$. If $e\#w$ and $(e, \sigma') \to v$, then $(e, \sigma) \to v$.*

Proof. By induction on the structure of e. Let σ, σ', μ with the above properties. Assume $e\#w$ and take a value v such that $(e, \sigma') \to v$. We want to show that $(e, \sigma) \to v$.

- Case $e = n$ or $e = a$. The evaluation of e does not depend upon the store, so the implication is trivial.
- Case $e = e'.f$. From the evaluation rule for field accesses we get $v = (c, f)$ and $(e', \sigma') \to (c, f_1)$ for some $c \in C$. By the definition of disjointness, $e\#w$ implies $e'\#w$. Using the induction hypothesis for e', we get $(e', \sigma) \to (c, f_1)$. By the evaluation rule for field accesses, $(e'.f, \sigma) \to (c, f)$. Hence, $(e, \sigma) \to v$.
- Case $e = *e'$. From the evaluation rule for dereferences we get $v = \sigma'(l)$ and $(e', \sigma') \to l$ for some $l \in L$. By the definition of disjointness, $e\#w$ implies $e'\#w$. Using the induction hypothesis for e', we get $(e', \sigma) \to l$. Hence, $(e, \sigma) \to \sigma(l)$.
  By the definition of disjointness, $e\#w$ implies pt$(e') \cap w = \varnothing$. Because $\sigma \models_\mu$ pt, the evaluation $(e', \sigma) \to l$ implies that $\mu(l) \in$ pt(e'). Therefore, $\mu(l) \notin w$. By the hypothesis, $\sigma(l) = \sigma'(l)$. Therefore, $\sigma(l) = v$, and $(e, \sigma) \to v$.
- Case $e = e_0 \oplus e_1$. From the evaluation rule for arithmetic operations we get $v = v_0 \oplus v_1$, $(e_0, \sigma') \to v_0$, and $(e_1, \sigma') \to v_1$ for some $v_0, v_1 \in \mathbb{Z}$. By the definition of disjointness, $e\#w$ implies $e_0\#w$ and $e_1\#w$. Using the induction hypothesis for e_0 and e_1, we get $(e_0, \sigma) \to v_0$ and $(e_1, \sigma) \to v_1$. By the evaluation rule for arithmetic expressions, $(e, \sigma) \to v$.

$\square$

Lemma 4 (Assignment Disjointness). *Let e be an expression,* pt *the points-to information, and* $s : *e_0 = e_1$ *and assignment. Let σ, σ', and μ such that $\sigma \models_\mu$* pt, *$\sigma' \models_\mu$* pt, *$(s, \sigma) \rightarrow \sigma'$. If* $e \# \mathsf{pt}(e_0)$ *and* $(e, \sigma') \rightarrow v$ *then* $(e, \sigma) \rightarrow v$.

Proof. Let $w = \mathsf{pt}(e_0)$. We show that $\sigma'(l') = \sigma(l')$ for all l' such that $\mu(l') \notin w$. Let l' such that $\mu(l') \notin w$. Let $l_0 \in L$ such that $(e_0, \sigma) \rightarrow l_0$. Because $\sigma \models_\mu$ pt, we get $\mu(l_0) \in \mathsf{pt}(e_0) = w$. Because $\mu(l') \notin w$, it must be the case that $l' \neq l_0$. By the semantics of assignments, this implies that $\sigma'(l') = \sigma(l')$. The proof now follows from Lemma 3 (Disjointness). $\square$

Lemma 5 (Substitution). *Let $s : *e_0 \leftarrow e_1$ be an assignment, and* pt *the points-to information. Let σ, σ', and μ such that $\sigma \models_\mu$* pt, *$\sigma' \models_\mu$* pt, *and $(s, \sigma) \rightarrow \sigma'$. Let $w = \mathsf{pt}(e_0)$. If $\mathcal{E}[*e_0] \# w$ and $(\mathcal{E}[*e_0], \sigma') \rightarrow v$ then $(\mathcal{E}[e_1], \sigma) \rightarrow v$.*

Proof. By induction on the structure of the expression context $\mathcal{E}$.

- Case $\mathcal{E} = [\cdot]$. We need to prove that if $e_0 \# w$ and $(*e_0, \sigma') \rightarrow v$ then $(e_1, \sigma) \rightarrow v$. By the semantics of dereferences, $(e_0, \sigma') \rightarrow l$ and $\sigma'(l) = v$. Because $e_0 \# w$, by Lemma 4 (Assignment Disjointness), $(e_0, \sigma) \rightarrow l$. By the semantics of assignments, $\sigma' = \sigma[l \mapsto v']$, where v' is the value of e_1 in the pre-store: $(e_1, \sigma) \rightarrow v'$. Since $\sigma'(l) = v$ and $\sigma'(l) = v'$, it means that $v = v'$. Therefore, $(e_1, \sigma) \rightarrow v$.

- Case $\mathcal{E} = \mathcal{E}'.f$. By hypothesis, $\mathcal{E}[*e_0] \# w$ and $(\mathcal{E}[*e_0], \sigma') \rightarrow v$. Therefore, $(\mathcal{E}'[*e_0], \sigma') \rightarrow (c, f_1)$ and $v = (c, f)$ for some $c \in C$. By definition of disjointness for contexts, $\mathcal{E}[*e_0] \# w$ implies $\mathcal{E}'[*e_0] \# w$. By induction hypothesis, $(\mathcal{E}'[e_1], \sigma) \rightarrow (c, f_1)$. By the semantics of field expressions, $(\mathcal{E}[e_1], \sigma') \rightarrow v$.

- Case $\mathcal{E} = \mathcal{E}' \oplus e$. By hypothesis, $\mathcal{E}[*e_0] \# w$ and $(\mathcal{E}[*e_0], \sigma') \rightarrow v$. Therefore, $(\mathcal{E}'[*e_0], \sigma') \rightarrow v_0$, $(e, \sigma') \rightarrow v_1$ and $v = v_0 \oplus v_1$. By definition of disjointness for contexts, $\mathcal{E}[*e_0] \# w$ implies $\mathcal{E}'[*e_0] \# w$ and $e \# w$. By induction hypothesis, $(\mathcal{E}'[e_1], \sigma) \rightarrow v_0$. Also, because $e \# w$, by Lemma 4 (Assignment Disjointness), $(e, \sigma) \rightarrow v_1$. By the semantics of arithmetic expressions, $(\mathcal{E}[e_1], \sigma) \rightarrow v$.

- Case $\mathcal{E} = e \oplus \mathcal{E}'$. This case is similar to the case above.

- Case $\mathcal{E} = *\mathcal{E}'$. By hypothesis, $\mathcal{E}[*e_0] \# w$ and $(\mathcal{E}[*e_0], \sigma') \rightarrow v$. Here, $\mathcal{E}[*e_0] = *\mathcal{E}'[*e_0]$. Therefore, $(\mathcal{E}'[*e_0], \sigma') \rightarrow l$ and $\sigma'(l) = v$ for some $l \in L$. By definition of disjointness for contexts, $\mathcal{E}[*e_0] \# w$ implies $\mathcal{E}'[*e_0] \# w$. By induction hypothesis, $(\mathcal{E}'[e_1], \sigma) \rightarrow l$. By the semantics of dereferences, $(\mathcal{E}[e_1], \sigma) \rightarrow \sigma(l)$. To prove that $(\mathcal{E}[e_1], \sigma) \rightarrow v$, we need to show that $\sigma(l) = \sigma'(l)$.
  Assume by absurd that $\sigma(l) \neq \sigma'(l)$. Then, location l has been written by the assignment, so $(e_0, \sigma) \rightarrow l$. Because $\sigma \models_\mu$ pt, it means that $\mu(l) \in \mathsf{pt}(e_0) = w$. On the other other hand, $(\mathcal{E}'[*e_0], \sigma') \rightarrow l$, so because $\sigma' \models_\mu$ pt, it means that $\mu(l) \in \mathsf{pt}(\mathcal{E}'[*e_0])$. Hence $\mathsf{pt}(\mathcal{E}'[*e_0]) \cap w \neq \varnothing$, which contradicts the fact that $\mathcal{E}'[*e_0] \# w$. Hence, the assumption was incorrect. This completes the proof for this case.

$\square$

Lemma 6 (Inference Rules). *Let $d = (S, H, M)$ be a dataflow fact,* pt *the points-to information, and $s : *e_0 \leftarrow e_1$ an assignment. Let H' and M' be the hit and miss sets computed using the inference rules from Figure 3. If l, σ, σ', and μ are such that $l \in L - A$, $\sigma' \models_\mu$* pt, *$\sigma \models_\mu$* pt, *$(l, \sigma') \models_\mu d$, and $(s, \sigma) \rightarrow \sigma'$, then:*

- $\forall e \in H' \, . \, (e, \sigma) \to l$.
- $\forall e \in M' \, . \, (e, \sigma) \to v$ *and* $v \neq l$.

Proof. For each of the rules in Figure 3, we show that the expressions generated in the pre-state satisfy the above two conditions. Let l, σ, σ', and μ satsifying the conditions above. Let $w = \mathsf{pt}(e_0)$.

- Rule [FILTER1]. Let e such that $e \# w$ and $(e, \sigma') \to v$. If $e \in H$, then $v = l$; and if $e \in M$, then $v \neq l$. From Lemma 4 (Assignment Disjointness), it follows that $(e, \sigma) \to v$.

- Rule [FILTER2]. Let e such that $e \# w$ and $*e \in H$. Also assume that $Miss(e_1)$ and $e_1 \# w$. We want to show that $(*e, \sigma) \to l$.
  Because $*e \in H$, we have $(*e, \sigma') \to l$. Hence, $(e, \sigma') \to l'$ and $\sigma'(l') = l$ for some $l' \in L$. From Lemma 4 (Assignment Disjointness), we get that $(e, \sigma) \to l'$.
  Assume by absurd that $\sigma(l') \neq l$. Since $\sigma'(l') = l$, it means that $l' = l_0$, where l_0 is the location being updated: $(e_0, \sigma) \to l_0$. Furthermore, the value written into $\sigma'(l_0)$ is the value of e_1 in the pre-store: $(e_1, \sigma) \to l$. Now, since $e_1 \# w$, by Lemma 4 (Assignment Disjointness) we have that $(e_1, \sigma') \to l$. Because $Miss(e_1)$, Lemma 2 (Miss Expression) implies $l \neq l$, a contradiction.
  Hence, $\sigma(l') = l$. Therefore $(*e, \sigma) \to l$.

- Rule [FILTER3]. The proof is similar to the case above.

- Rule [FILTER4]. Assume that $e_1 = *e$, $e \# w$, and $(e_1, \sigma') \to v$. Hence, if $e_1 \in H$, then $v = l$; and if $e_1 \in M$, then $v \neq l$. We must show that $(e_1, \sigma) \to v$.
  By the semantics of assignments, $(e, \sigma') \to l'$ and $\sigma'(l') = v$, for some $l' \in L$. Because $e \# w$, using Lemma 4 (Assignment Disjointness), we have $(e, \sigma) \to l'$.
  Let l_0 such that $(e_0, \sigma) \to l_0$. There are two subcases. Subcase 1: $l_0 \neq l'$, i.e., the value of location l' has not been updated. Then, $\sigma(l') = v$. Therefore, $(*e, \sigma) \to v$ or $(e_1, \sigma) \to v$. Subcase 2: $l_0 = l'$, i.e., the value of location l' has been updated. The written value is v (because $\sigma'(l') = v$), and this value must be the value of e_1 in the pre-state. Hence $(e_1, \sigma) \to v$.
  In both subcases we get $(e_1, \sigma) \to v$.

- Rule [SUBST1]. Let $e = \mathcal{E}[*e_0]$ such that $e \# w$ and $(e, \sigma') \to v$. If $e \in H$, then $v = l$; and if $e \in M$, then $v \neq l$. From Lemma 5 (Substitution), $(\mathcal{E}[e_1], \sigma) \to v$.

- Rule [SUBST2]. Assume that $e_0 \# w$ and $Miss(*e_0)$, (S, H, M). By Lemma 2 (Miss Expressions), $(*e_0, \sigma') \to v$ and $v \neq l$. By Lemma 5 (Substitution) for $\mathcal{E} = [\cdot]$, $(e_1, \sigma) \to v \neq l$.

$\square$

Lemma 7 (Infeasibility). *Let s be an assignment, d and d' dataflow facts, and pt is the points-to information. If $\models_{\mathsf{pt}} \{d\} \, s \, \{d'\}$ and Infeasible(d), then $\models_{\mathsf{pt}} \{\bot\} \, s \, \{d'\}$.*

Proof. Let σ, σ', μ such that: 1) $\sigma' \models_\mu \mathsf{pt}$, 2) $\sigma \models_\mu \mathsf{pt}$, 3) $(l, \sigma') \models_\mu d'$, and 4) $(s, \sigma) \to \sigma'$. Because $\models_{\mathsf{pt}} \{d\} \, s \, \{d'\}$, it means that $(l, \sigma) \models_\mu d$. Because *Infeasible(d)*, it means that there are no values of σ, σ', and μ such that conditions 1-4 hold. Hence, any implication based on these assumptions is vacuously true; using any dataflow fact in the pre-state is sound. In particular, using $\bot$ is sound: $\models_{\mathsf{pt}} \{\bot\} \, s \, \{d'\}$.

$\square$

Lemma 8 (Cleanup). *Let s be an assignment, d and d' dataflow facts, and* pt *is the points-to information. If* $\models_{\mathsf{pt}} \{d\}\ s\ \{d'\}$, *then* $\models_{\mathsf{pt}} \{Cleanup(d)\}\ s\ \{d'\}$.

Proof. The result holds because $Cleanup(d) \sqsubseteq d$, and because of safety of over-approximations: $\forall l, \sigma, \mu, d_0, d_1 \ . \ (l, \sigma) \models_{\mu} d_0 \ \wedge \ d_0 \sqsubseteq d_1 \ \Rightarrow \ (l, \sigma) \models_{\mu} d_0$. These facts follow immediately from the definition of *Cleanup* and from the definition of the abstraction function.

$\square$

Lemma 9 (Assignment Transfer Function). *If $s : *e_0 \leftarrow e_1$ is an assignment, d is the dataflow information after s, and* pt *is the points-to information, then* $\models_{\mathsf{pt}} \{[\![s]\!](d)\}\ s\ \{d\}$.

Proof. Let $d = (S, H, M)$. Let $S' = S \cup \mathsf{pt}(e_0)$, and let H' and M' be the hit and miss sets computed using the inference rules in Figure 3.

We want to prove that $\models_{\mathsf{pt}} \{(S', H', M')\}\ s\ \{d\}$. Let l, σ, σ', μ such that $\sigma' \models_{\mu} \mathsf{pt}$, $\sigma \models_{\mu} \mathsf{pt}$, $(l, \sigma') \models_{\mu} d$, and $(s, \sigma) \to \sigma'$. We must show that $(l, \sigma') \models_{\mu} (S', H', M')$. More precisely, we must show conditions 1-3 from the definition of the abstraction function.

Conditions 1-2 require that the hit and miss sets H' and M' computed using the inference rules are sound approximations of l. By Lemma 7 (Inference Rules), conditions 1 and 2 hold.

We show that condition 3 holds for S'. Let $l' \in L$ such that $\sigma(l') = l$. We must show that $\mu(l') \in S'$. There are two cases. Case 1: $\sigma(l') = \sigma'(l')$. Hence, $\sigma'(l') = l$. By the hypothesis, $(l, \sigma') \models_{\mu} (S, H, M)$, so $\mu(l') \in S$. Since $S \subseteq S'$, $\mu(l') \in S'$. Case 2: $\sigma(l') \neq \sigma'(l')$. Then location l' has been updated by the assignment. Hence, l' is the value of e_0 in the pre-state: $(e_0, \sigma) \to l'$. Because $\sigma \models_{\mu} \mathsf{pt}$, it means that $\mu(l') \in \mathsf{pt}(e_0)$. Hence, $\mu(l') \in S'$. In both cases, $\mu(l') \in S'$, so condition 3 is met.

Therefore $(l, \sigma') \models_{\mu} (S', H', M')$. Hence: $\models_{\mathsf{pt}} \{(S', H', M')\}\ s\ \{d\}$. The final result of the transfer function for assignments is then sound because of Lemma 7 (Infesibility) and Lemma 8 (Cleanup).

$\square$

Lemma 10 (Malloc Transfer Function). *If $s : *e_0 \leftarrow$* malloc *is an allocation statement, d is a dataflow information after s, and* pt *is the points-to information, then* $\models_{\mathsf{pt}} \{[\![s]\!](d)\}\ s\ \{d\}$.

Proof. Let $d = (S, H, M)$. Let l, σ, σ', μ such that 1) $\sigma' \models_{\mu} \mathsf{pt}$, 2) $\sigma \models_{\mu} \mathsf{pt}$, 3) $(l, \sigma') \models_{\mu} d$, and 4) $(s, \sigma) \to \sigma'$. Let $w = \mathsf{pt}(e_0)$. Following the transfer function for assignments, we analyze the following cases:

- Case 1: $(*e_0) \in H$, $e_0 \# w$, and there exists e such that $(*e) \in H$, $\mathsf{pt}(e) \cap w = \varnothing$. Because $(*e_0) \in H$, it means that $(e_0, \sigma') \to l_0$ and $\sigma'(l_0) = l$, for some $l_0 \in L$. Because $(e) \in H$, it means that, $(e, \sigma') \to l_1$ and $\sigma'(l_1) = l$, for some $l_1 \in L$. Furthermore, $\mu(l_0) \in \mathsf{pt}(e_0)$ and $\mu(l_1) \in \mathsf{pt}(e)$, so $\mathsf{pt}(e) \cap w = \varnothing$ implies $\mu(l_0) \neq \mu(l_1)$. Therefore, $l_0 \neq l_1$.
  Because $(e_0, \sigma') \to l_0$ and $e_0 \# w$, by Lemma 4 (Assignment Disjointness) we infer that $(e_0, \sigma) \to l_0$. Hence, l_0 is the memory location that has been updated by this statement. By the semantics of allocation statements, $\sigma'(l_0) = (c, f_1)$, where $c \in C$ is fresh. Since c is fresh and the assignment updates a single memory location, it means that l_0 is the only location where σ' has value (c, f_1). Since $\sigma'(l_1) = \sigma'(l_0)$ and $l_0 \neq l_1$, we get a contradiction. Hence, there are no l, σ, σ', μ such that assumptions 1-4 hold in this case. Any implication based on these assumptions is vacuously true. Therefore, can safely use $\bot$ in the pre-state: $\models_{\mathsf{pt}} \{\bot\}\ s\ \{d\}$.

- Case 2.1: *Miss*$(*e_0)$ and $e_0 \# w$. Then $(e_0, \sigma') \to l_0$ and $\sigma'(l_0) \neq l$, for some $l_0 \in L$. Because $(e_0, \sigma') \to l_0$ and $e_0 \# w$, by Lemma 4 (Assignment Disjointness) we infer that $(e_0, \sigma) \to l_0$. Therefore, l_0 is the location being updated: $\sigma'(l_0) = (c, f_1)$, where $c \in C$ is fresh. Hence, the newly allocated cell is not the error cell: $l \neq (c, f_1)$.

Consider another store σ'' such that $(*e_0 \leftarrow 0, \sigma) \rightarrow \sigma''$. In other words, σ'' is the result of performing a nullification instead of an allocation. Stores σ'' and σ' are the same for all locations, except at l_0, where σ'' has value 0; and at (c, f_1), where σ'' is not defined. The error cell l is present in both stores: $l \in dom(\sigma') \cap dom(\sigma'')$.

According to the semantics of allocation statements, the fields of the new cell are all nullified. Hence, expressions that evaluate to addresses other than the new cell will proceed identically in the two stores: $(e, \sigma') \rightarrow l' \neq (c, f)$ iff $(e, \sigma'') \rightarrow l'$.

Therefore, all points-to relations in σ'' are also present in σ', and since $\sigma' \models_\mu \mathsf{pt}$, it means that $\sigma'' \models_\mu \mathsf{pt}$. Also, since the error cell is not the new cell $l \neq (c, f_1)$, all expressions that hit (or miss) the cell in σ' will also hit (or miss) in σ''. Finally, the locations that hold references to the error cell are the same in σ' and σ'' because $l \neq (c, f_1)$. Therefore, $(l, \sigma'') \models_\mu (S, H, M)$.

Now because $\sigma'' \models_\mu \mathsf{pt}$ and $(l, \sigma'') \models_\mu (S, H, M)$, by the soundness of assignment statement we get $(l, \sigma) \vdash_\mu [\![*e_0 \leftarrow 0]\!](S, H, M)$. Hence, $\models_{\mathsf{pt}} \{[\![*e_0 \leftarrow 0]\!](S, H, M)\} \ s \ \{d\}$.

- Case 2.2: There exists e such that $(*e) \in H$ and $\mathsf{pt}(e) \cap w = \varnothing$. Then $(e, \sigma') \rightarrow l'$ and $\sigma'(l') = l$, for some $l' \in L$. By the soundness of pt, we have $\mu(l') \in \mathsf{pt}(e)$.

  Let l_0 such that $(e_0, \sigma) \rightarrow l_0$. Then $\mu(l_0) \in \mathsf{pt}(e_0)$. Because $\mathsf{pt}(e) \cap w = \varnothing$, we get $l_0 \neq l'$. But l_0 is the location being written by the statement. According to the semantics of allocations, location l_0 is updated with the address of the allocated cell: $\sigma'(l_0) = (c, f_1)$; and l_0 is the only location that holds the value in σ'. Therefore, because $l' \neq l_0$, we get $\sigma'(l') \neq \sigma'(l_0)$. But $\sigma'(l') = l$ and $\sigma'(l_0) = (c, f_1)$. So the error cell is not the newly allocated cell: $l \neq (c, f_1)$. The rest of the proof is then the same as in the case above.

- Otherwise, the result is $\top$. This is always a safe approximation: $\models_{\mathsf{pt}} \{\top\} \ s \ \{d\}$.

$\square$

Lemma 11 (Free Transfer Function). *If* $s : \mathtt{free}(e)$ *is a deallocation statement, d is the dataflow fact after s, and pt is the points-to information, then* $\models_{\mathsf{pt}} \{[\![s]\!](d)\} \ s \ \{d\}$.

Proof. Let $d = (S, H, M)$. Let l, σ, σ', μ such that 1) $\sigma' \models_\mu \mathsf{pt}$, 2) $\sigma \models_\mu \mathsf{pt}$, 3) $(l, \sigma') \models_\mu d$, and 4) $(s, \sigma) \rightarrow \sigma'$. We have three cases:

- Case 1: $e \in H$. Then $(e, \sigma') \rightarrow l$. Since deallocations just remove a cell from the heap, but leave all other locations unchanged, it means that the evaluation of e yields the same value in the pre-store: $(e, \sigma) \rightarrow l$. By the semantics of free $l = (c, f_1)$ where $c \in C$ is the deallocated cell. Hence, e is a dangling reference in the post-store: $l \notin dom(\sigma')$. As a result, assumption $(l, \sigma') \models_\mu d$ is invalid, and any pre-state is sound: $\models_{\mathsf{pt}} \{\bot\} \ s \ \{d\}$.

- Case 2: *Miss*(e, d). According to the semantics of deallocations $dom(\sigma') \subseteq dom(\sigma)$ and all memory locations in $dom(\sigma')$ have the same values in σ as they have in σ'. Hence, d is also a sound abstraction before the deallocation: $\models_{\mathsf{pt}} \{d\} \ s \ \{d\}$.

- Case 3: Otherwise, the same argument as in Case 2 applies here. In addition, $e \in M'$. Otherwise, if $(e, \sigma) \rightarrow l$, then $l = (c, f_1)$ where $c \in C$ is the freed cell. This would contradict the assumption that $(l, \sigma') \models_\mu d$, since $(c, f_1) \notin dom(\sigma')$. Hence, $e \in M'$ and $\models_{\mathsf{pt}} \{(S, H, M \cup \{e\})\} \ s \ \{d\}$.

$\square$

Lemma 12 (Condition Transfer Function). *If* $s : \mathtt{cond}(e_0 \equiv e_1)$ *is a condition statement,* d' *is the dataflow fact after* s*, and* $\mathtt{pt}$ *is the points-to information, then* $\models_{\mathtt{pt}} \{[\![s]\!](d')\}\ s\ \{d'\}$.

Proof. The semantics of condition statements shows that conditions don't change the state of the store. Hence, d is automatically a sound abstraction before s. We just need to prove that the newly added expressions in H' (or M') indeed hit (or miss) the error cell. This is straightforward, since evaluation of expressions in the pre-store and in the post-store proceed identically.

Let l, σ, μ such that $\sigma' \models_\mu \mathtt{pt}$, $\sigma \models_\mu \mathtt{pt}$, $(l, \sigma') \models_\mu d$, and $(s, \sigma) \to \sigma'$. By the semantics of conditions, $\sigma' = \sigma$.

Let statement s be $\mathtt{cond}(e_0 = e_1)$. Let $e_0 \in M$. Then $(e_0, \sigma) \to l$. By the semantics of conditions, e_0 and e_1 must evaluate to the same value, so $(e_1, \sigma) \to l$, therefore it is safe to add e_1 to the miss set M'. This proves the correctness of the first rule. The proof for all other rules is similar. $\square$

Theorem 2 (Transfer Function Soundness). *If* s *is a statement,* d' *is a dataflow fact after this statement, and* $\mathtt{pt}$ *is the points-to information, then the dataflow fact* $d = [\![s]\!](d')$ *that the transfer function computes before* s *is sound relative to the points-to information* $\mathtt{pt}$*:*

$$\models_{\mathtt{pt}} \{\ [\![s]\!](d')\ \}\ s\ \{\ d'\ \}$$

Proof. The proof is provided by Lemma 9 (Assignment Transfer Function), Lemma 10 (Malloc Transfer Function), Lemma 11 (Free Transfer Function), and Lemma 12 (Condition Transfer Function). $\square$