Practical Memory Leak Detection using Guarded Value-Flow Analysis *

Sigmund Cherem Lonnie Princehouse Radu Rugina

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

Abstract

This paper presents a practical inter-procedural analysis algorithm for detecting memory leaks in C programs. Our algorithm tracks the flow of values from allocation points to deallocation points using a sparse representation of the program consisting of a value flow graph that captures def-use relations and value flows via program assignments. Edges in the graph are annotated with guards that describe branch conditions in the program. The memory leak analysis is reduced to a reachability problem over the guarded value flow graph. Our implemented tool has been effective at detecting more than 60 memory leaks in the SPEC2000 benchmarks and in two open-source applications, *bash* and *sshd*, while keeping the false positive rate below 20%. The sparse program representation makes the tool efficient in practice, and allows it to report concise error messages.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Reliability; D.3.4 [*Processors*]: Compilers, Memory management; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, Languages, Reliability, Verification

Keywords Static error detection, memory leaks, memory management, value-flow analysis

1. Introduction

The increasing importance of software reliability has led to a large body of research aimed at identifying violation of various program safety properties, including memory safety properties (such as null pointer dereferences, or heap errors), resource usage properties (e.g., file usage or locking discipline), or security properties (such as the use of tainted data).

Among these, several important properties can be expressed as properties of the following form: on any execution of the program, the value v generated by a program event A must flow into (or be consumed by) exactly one occurrence of another event B. We refer to such properties as *source-sink* properties. Examples include detecting lock usage violations (where A is the lock acquire action, B is the lock release, and the value is the pointer to the lock); or detecting potential memory leaks or double frees (where A is the heap allocation event, B represents heap deallocation, and the value is the pointer to the allocated heap cell).

Current error-detection techniques include both general-purpose tools for checking arbitrary finite-state machine properties [7, 5, 1]; as well as tools that have been used to check specific properties such as memory leaks [11, 21, 14]. The existing approaches largely fall into two categories: dataflow analysis approaches that track the state of program values through the control-flow, computing information at each program point; and flow-insensitive approaches, that track the flow of values using a sparse representation of the program (for instance, using definition-use chains or SSA form), but ignoring the control-flow otherwise. The latter is more efficient due to the sparse representation of value flows, but it is not applicable to checking arbitrary state-machine properties since it cannot reason about events that must eventually take place on all program paths, such as deallocating memory, releasing locks, or closing files. To the best of our knowledge, all of the existing memory leak detection techniques use flow-sensitive approaches.

This paper presents a novel inter-procedural analysis algorithm for checking source-sink properties using a sparse representation of value flows, and applies this algorithm to the detection of heap memory errors, such as memory leaks or double frees. Our analysis identifies value flows from `malloc` sources to `free` sinks through program assignments and use-def chains using a value-flow graph representation of the program. A simple case of a memory leak occurs when a source never reaches a sink in the value-flow graph. Checking the other cases, where a malloc source reaches a free sink on all program paths, and only once on each path, is more challenging. Our approach is to annotate the edges in the value-flow graph with *guards* that represent branch conditions under which the value flow happens, and then use the guard information to reason about sink reachability on all paths.

The following list highlights the main features of our approach to the source-sink problem:

- *Efficiency:* The analysis is efficient due to the sparse program representation using value flows. The flows of data from sources to sinks need not be tracked through all of the intermediate program points, but only through the relevant ones;
- *Analysis refinement:* The analysis first uses simple techniques to identify potential errors. It falls back on the more expensive reasoning about guarded value-flows only when the simple ap-

* This work was supported in part by National Science Foundation grant CCF-0541217 and AFOSR grant FA9550-06-1-0244.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

proaches fail. Guards are computed in a demand-driven fashion, only for the relevant portions of the program.

- *Concise error reports:* The use of sparse value-flows makes it possible to report concise messages to point to a few relevant assignments and path conditions that cause the error to happen.

We have implemented our memory leak detection analysis and applied it to the set of SPEC2000 benchmark programs. Our results show that the analysis presented in this paper is substantially faster than existing leak detectors. Compared to a backward dataflow analysis memory leak detector that we have previously proposed [14] and where we experimented with a similar set of benchmarks, the value-flow analysis presented in this paper runs faster, finds roughly the same number of errors, and yields a lower number of false warnings.

We have also used our method to derive a classification of heap allocation and deallocation patterns. We find this classification useful for understanding how heap allocation is being used, and how difficult is it to reason about deallocation in real programs.

The remainder of the paper is organized as follows. We first present the problem statement in Section 2. Next, Section 3 presents a motivating example. Section 4 presents an overview of our system. We discuss the analysis algorithm in Section 5, present an evaluation in Section 6, discuss related work in Section 7, and conclude in Section 8.

2. Problem Statement and Classification

We define a classification of value-flow problems and place the memory leak and double free problems in this context.

DEFINITION 1. *Consider a program P that consists of three kinds of statements: source statements s that produce fresh values when executed; assignment statements that copy values; and sink statements k that receive values. A* **source-sink[n,m]** *safety-checking problem is the problem of checking that each dynamic value instance produced by a source will eventually flow into at least n and at most m sinks in any execution of the program.*

Example source statements include heap allocation statements, opening files, acquiring locks, or loading untrusted values. Corresponding sink statements include heap deallocations (frees), closing files, releasing locks, or storing values into trusted memory locations.

Given this definition, the memory leak detection problem is a source-sink[1,∞] problem, since it requires that each allocated cell must be eventually freed. On the other hand, the double-free problem is a source-sink[0,1] problem since it requires that each allocated cell must be freed at most once. Checking both properties becomes a source-sink[1,1] problem, that is, each allocated cell must be freed exactly once. We will refer to source-sink[1,1] problems simply as *source-sink* problems.

In contrast, the problem of determining that an untrusted value never flows into a trusted location is a source-sink[0,0] problem. We will refer to such problems as *source-not-sink* problems.

Solving source-sink problems using a sparse representation of value flows (such as def-use chains, or SSA form) is more difficult than solving source-not-sink problems. This is the case because value-flow edges are "may" edges, so value-flow paths indicate that values may flow from sources to sinks. This information is enough for source-not-sink problems, but not for source-sink problems. The latter require more precise information to determine that values *must* eventually flow into sinks.

This paper addresses the source-sink problem in the context of memory leak detection. Our solution is to annotate each value-flow edge with a guards that precisely describe the branch conditions under which the value-flow takes place.

```
 1  int func() {
 2    int *p = malloc();
 3    if (p == NULL)
 4      return -1;
 5
 6    int *q = malloc();
 7    if (q == NULL)
 8      return -1;
 9
10    ... /* use p[i] and q[j] */
11
12    free(p);
13    free(q);
14    return 0;
15  }
```

Figure 1. Example: leaked cell on an early exit path. A deallocation `free(p)` is needed before returning at line 8.

Finite-state machine properties form a more general class of properties, of which source-sink and source-not-sink problems are particular cases. More general finite-state machine problems include checking the absence of unsafe accesses through dangling-pointers, or checking that file operations access opened files. This paper does not address such problems.

In the remainder of the paper we will only discuss the memory leak and double free problems, and will use the terms malloc and source, and free and sink interchangeably.

3. Examples

This section discusses two examples that illustrate the main challenges of analyzing source-sink problems in the context of the memory leak detection problem.

3.1 Example 1: Missing Free on Function Exit Path

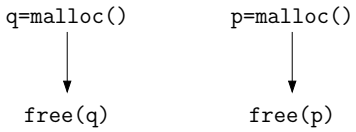
The code fragment from Figure 1 shows a simplified version of several leaks found in *ammp*, one of the SPEC2000 benchmarks. The program manipulates two heap-allocated arrays: array `p` allocated at line 2, and array `q` allocated at line 6. The function uses the arrays locally, and deallocates them at the end of the function, at lines 12 and 13. The omitted code at line 10 accesses array elements `p[i]` and `q[j]`, but doesn't store copies of pointers `p` or `q` in other variables or program structures. After each allocation, the program checks if the call to `malloc()` successfully returns a non-null pointer (lines 3 and 7); if so, the function immediately returns an error code. The program leaks memory at the return at line 8. At that point, the function returns without freeing the successfully allocated array `p`.

3.1.1 Leak Detection via Dataflow Analysis

This leak can be identified using any of the dataflow analyses proposed in [7, 5, 10]. Starting from a (successful) allocation at line 2, a dataflow analysis will analyze all forward program paths from that point, tracking the typestate of `p` (i.e., allocated or deallocated state) up to each of the return points at lines 4, 8, and 14, when `p` goes out of scope. The analysis will identify that the path to line 4 is infeasible when the allocation was successful; that the returning at line 14 is safe because the cell has just been deallocated at line 12; and that an error occurs at line 8 because the cell is not freed on that return path. Note that the dataflow analysis doesn't need to be path-sensitive (i.e., be able to identify correlated branches, as in [5]) to handle this example.

3.1.2 Leak Detection via Guarded Value Flows

Our goal is to identify the memory leak in the above example just by reasoning about value flows in the program. We describe value-flows using a graph whose nodes represent definitions of program variables, and edges describe flows via program assignments. In addition, there is a node for each `free(x)` statement; its predecessors are the definitions of `x`. The value-flow graph for the above example is shown below:



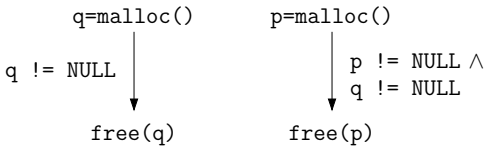
The graph indicates that each of the values `p` and `q` flows into its corresponding free. The other uses of `p` and `q`, such as `p[i]` and `q[j]`, are not included since they do not store copies of these pointers, so they are not relevant.

However, the information in the above value-flow graph is not enough for identifying the error: it shows that `p` and `q` may be freed freed; but it doesn't show that they always freed, on all program paths. In fact, we can write a correct program that would have the same value-flow graph:

```
p = malloc();
if (p == NULL) return -1;
else { ...; free(p); }

q = malloc();
if (q == NULL) return -1;
else { ...; free(q); }
```

The question is how to distinguish between the two cases. Our solution to this problem is to annotate the value-flow edges with *guards* that describe the branch conditions under which the value-flow takes place. For the program from Figure 1, the guarded flow graph is:



The condition `q != NULL` for `q` is redundant, and so is the condition `p != NULL` for `p`, because the analysis is only concerned with the cases where allocations are successful. Excluding these conditions, the analysis determines that the allocation `q = malloc()` always flows into `free(q)`, hence it is safe; but the allocation `p = malloc()` flows into `free(p)` only when `q != NULL`. The analysis therefore concludes that the program leaks in the opposite case, when `q == NULL`, and reports the error:

```
Error: test.c, line 2: allocation leaks memory
Leak path:  q == NULL   test.c, line 7
```

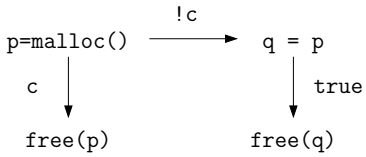
The message concisely summarizes the problem and its cause.

3.2 Example 2: Aliases and Multiple Deallocations

Identifying that source mallocs flow into frees on all program paths using value-flow graphs becomes more difficult when the program creates aliases of the allocated cell. The code fragment below illustrates the issue:

```
int *p = malloc();
if (c) { ...; free(p);}
else   { q = p; ...; free(q);}
```

The above program safely deallocates the allocated heap cell on each of the two program paths. However, deallocation takes place at multiple points, through different value aliases. The guarded value flow graph for this example is as follows (where the edge labeled "`true`" indicates that the value assigned at the definition `q = p` always flows into `free(q)`):



Using the guards on the edges, the analysis performs the following reasoning to show the absence of errors. The analysis inspects each `free` reachable from the allocation and determines that: `free(p)` takes place when condition `c` holds; and `free(q)` takes place when $!c \wedge \texttt{true} = !c$ holds. Using this information, the compiler can check for heap errors as follows:

- *Absence of leaks:* Since $c \vee !c = \texttt{true}$ it means that the two cases cover all possibilities, so the allocated cell is freed on all paths;
- *Absence of double frees:* Since $c \wedge !c = \texttt{false}$ it means that the conditions under which the two frees take place are disjoint, hence the allocated cell is never freed twice.

4. System Overview

Figure 2 gives an overview of our program analysis system. The system is built using Crystal [16], a program analysis infrastructure for C developed in our group. The shaded components have been developed part of this work, while the others are provided by Crystal or have been taken from external sources. The analyzer consists of the following components:

- *Front-end*. The front-end parses the program and builds a control-flow graph representation of the program.
- *Reaching definition analysis*. A standard dataflow analysis computes the set of uses for each definition in the program.
- *Value-flow graph construction.* The results of the reaching definitions analysis are used to build the value flow graph. The guards on the value-flow edges are not computed yet; the guard computation is postponed for a subsequent phase.
- *Region points-to analysis.* The analyzer uses the pointer analysis provided by Crystal to disambiguate indirect memory references via pointers. This is a flow-insensitive unification-based points-to analysis [18]. The analysis is context-insensitive, but field-sensitive. The equivalence classes computed by the analysis provide a partitioning of the memory into disjoint regions. Indirect memory accesses are represented in the value-flow graph using their memory region.
- *Unguarded reachability analysis*. Once the value-flow graph is computed, the analyzer checks each allocation site in turn. The analysis searches the value-flow graph for all of the frees that the allocation may reach. The reachability algorithm is context-sensitive and matches value flows at calls and returns. If no free statements are encountered, the analyzer classifies the current allocation site as one that is never freed. Otherwise, it extracts the sub-graph relevant to the allocation site and proceeds to the guard computation in the next phase.
- *Guarded reachability analysis*. In this phase, the analyzer requests guard information on each of the value-flow edges in the subgraph relevant to the currently analyzed allocation site.

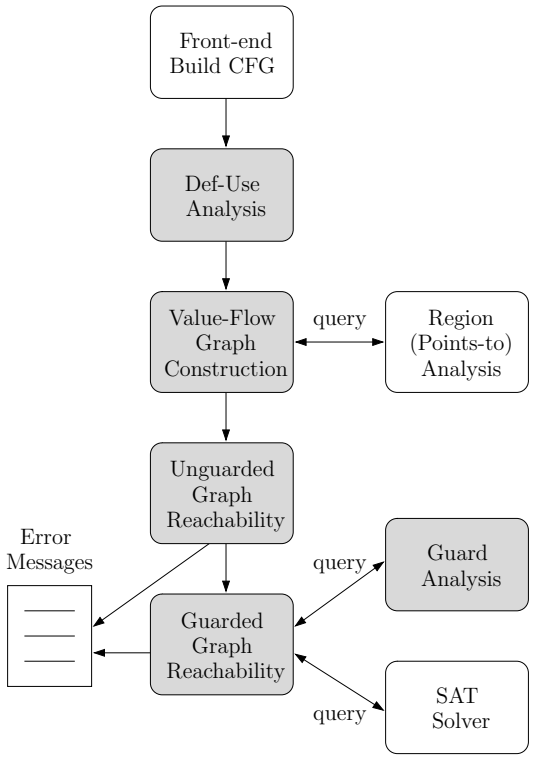


Figure 2. Analysis system overview

These queries are answered by the guard analyzer module. Once value-flow guards are computed, the algorithm derives larger formulas describing the entire path from the malloc to each free. The analyzer then generates two queries, one for memory leaks and another for double frees, as in the example discussed Section 3.2. The resulting formulas are checked for satisfiability using a SAT-solver. If a satisfying assignment of boolean values to guards is reported by the solver, then a violation is detected and reported.

- *Guard analysis*. This component answers queries about guards on value-flow edges. The analysis identifies the program points that correspond to the source s and target t of the edge, and then performs a local analysis of the control-flow between these points. The computed formula describes the program conditions under which the control flows from s to t.
- *SAT Solver*. Satisfiability queries are answered using SAT4J, freely available SAT-solver [3].

The following section discusses the program representation provided by the front-end. Next, it presents the details of our analyses.

4.1 Program Representation

The front-end parses the source program and builds a control-flow graph (CFG). The input program is simplified to a canonical form where each node in the control-flow graph is one of the following:

- *Assignment node:* $e = e'$, where e and e' are program expressions.
- *Call node:* $e = f(e_1, ..., e_k)$, where e, e_1,.., e_k are expressions in canonical form. The left-hand side expression might be missing, so this also models calls to functions that return void, or calls that don't assign the returned value.
- *Return node:* `return` e, where e is a program expression.
- *Switch node:* `switch` e $(c_1 : n_1, ..., c_k : n_k)$, where e is the test condition, c_i are the case constants for each branch, and n_i are the corresponding successor nodes. Control flows to n_i when $e = c_i$. A constant `default` represents the default case. If statements are modeled as `switch` e $(0 : n_f, \texttt{default} : n_t)$.

Allocation sites are calls of the form `x = malloc();` and deallocations are calls of the form `free(x)`. For simplicity, assume that in both cases x is a local variable.

We denote by V the set of local variables, and P the set of function parameters. Variables whose addresses are taken with the address-of operator & are considered as non-local expressions. We use N to represent the set of all CFG nodes. For each node $n \in N$ (switch or otherwise), k_n is the number of successors of n, and $I_n = \{1, .., k_n\}$ is the range of possible successors. We denote by n_i the i-th successor of n, where $i \in I_n$. Assignments and call nodes each have a single successor.

Expressions e don't have side-effects and include variables (locals, parameters, or globals), as well as arbitrary expression trees involving pointer dereferences, field accesses, array accesses, and unary or binary operations.

5. Analysis Algorithm

This section covers the details of the memory leak analysis, discussing the algorithm for each of the shaded components in Figure 2.

5.1 Reaching Definitions

This is the standard reaching definitions analysis implemented as a bitvector dataflow analysis. The analysis tracks only local variables and parameters whose addresses have not been taken. For each node n, the analysis computes a set of variable definitions that reach node n.

5.2 Building the Value-Flow Graph

The value-flow graph (VFG) captures the flow of values through program assignments. The nodes of the value-flow graph include:

- *Variable definitions*: There is a VFG node for each definition of a variable $x \in V$ at a CFG node n. By abuse of notation, we also refer to the resulting VFG node as n. Definitions at allocation sites are marked as source nodes.
- *Frees*: There is a VFG node for each free statement. These are marked as sink nodes.
- *Regions*: There is a VFG node for each memory region that doesn't correspond to a local variable or a parameter. As mentioned earlier, the region partitioning is provided by the unification-based pointer analysis.
- *Call site*: There is a VFG node for each local variable $x \in V$ passed as an argument at a call site. The VFG node of an actual argument x at a call node n is denoted by $[x_@n]$.
- *Parameters*: There is a VFG node for each formal parameter $p \in P$, denoted by $[p]$.
- *Return*: There is a VFG node for each return statement n in a function.

The edges of the value-flow graph are constructed by traversing each CFG node n in the program and performing one of the following actions:

- If n is an assignment $x = y$ where $x, y \in V$, add an edge $n' \rightarrow n$ from each definition n_y of y that reaches n.

Statement (n)	Added edges
$y = x$	$n_x \rightarrow n$
free(x)	$n_x \rightarrow n$
return(x)	$n_x \rightarrow n$
$y = f(..., x, ...)$	$n_x \rightarrow [x_{@}n]$ $[x_{@}n] \rightarrow [p]$ $n^f_{\mathsf{ret}} \rightarrow n$

Where $x, y \in V$, n_x is a definition of x that reaches n, p is the formal parameter of argument x, and n^f_{ret} is a return node of function f.

Figure 3. Value Flow Graph construction rules for propagation of values through program variables.

- If n is of the form $x = e$ where $x \in V$ but $e \notin V$, then query the points-to analysis for the region r of e, add a load edge $n_r \rightarrow n$ from the node n_r of region r.
- If n is of the form $e = x$ where $x \in V$ and $e \notin V$, add a store edge $n \rightarrow n_r$ from n to the region r of e.
- If n is a $\texttt{free}(x)$ statement, add an edge $n_x \rightarrow n$ from definition n_x of x that reaches n.
- If n is a $\texttt{return}(e)$ statement, add edges from the reaching definitions of e (if $e \in V$) or from the region of e (if $e \notin V$) to the return node n.
- If n is a call $e = f(x_1, ..., x_n)$, the analysis adds three kinds of edges. First, it adds an edge $n_{x_i} \rightarrow [x_{i@}n]$ from each reaching definition n_{x_i} of x_i to $[x_{i@}n]$. Second, it adds a call edge $[x_{i@}n] \rightarrow [p_i]$ from the node of each actual argument x_i to the node of its corresponding formal parameter p_i of the callee. Third, the analysis adds return edges to model the assignment of the return value to the left hand side of assignments at call sites. For each return node n_{ret} in the callee, the analysis adds a return edge from $n_{\mathsf{ret}} \rightarrow n$, if $e \in V$. Otherwise, it queries the points-to analysis for the region r of e and adds a store edge $n_{\mathsf{ret}} \rightarrow n_r$. For indirect function calls through function pointers, the analysis repeats the above procedure for each potential callee.

  Call edges and return edges are labeled with call-site information in a standard fashion: for a call site n, call edges are labeled with open parentheses $(_n$ and return edges are labeled with close parentheses $)_n$. Feasible inter-procedural value flows correspond to paths containing properly nested parentheses and context sensitive analyses can be formulated as context-free language (CFL) reachability problems [15].

Figure 3 summarizes the process of building the value-flow graph for the propagation of values through variables and parameters. The edges constructed so far are not annotated with guards. Branch conditions will be computed subsequently, only when needed.

5.3 Unguarded Reachability Detection

Once the value-flow graph is built, the algorithm analyzes each source allocation site *src*. The goal of this stage is to determine whether the allocation reaches any free, and if so, what is the relevant portion of the VFG that connect the malloc to the reaching frees. The algorithm is as follows:

1. Identify the set of nodes F_{src} reachable from the source *src* using a forward traversal over the nodes N of the VFG:

   $$F_{src} = \mathit{CFLForwardReach}(src, N)$$

   The algorithm uses CFL-reachability to match call and return edges and eliminate unfeasible inter-procedural flows of values. Hence the analysis is context-sensitive. The algorithm uses function summaries to cache and reuse matched calls and returns. This is a standard approach and we omit the details.

2. Identify a set of reachable sinks K:

   $$K = \{k \in F_{src} \mid k \text{ is a free node}\}$$

3. Apply one of the following three cases:

   - If there are no reachable sinks, then classify *src* as an allocation that is never freed. The analysis of this allocation site stops here.
   - If some region node n_r is reachable from *src* ($n_r \in F_{src}$) then classify the allocation as one that flows into global scope or aggregate data structures such as arrays. The algorithm will classify this case as too complex, and will not further analyze this allocation site.
   - Otherwise, identify a *relevant slice* R of the VFG, consisting of all nodes on paths from *src* to a reachable sink. This is the set of relevant nodes for the allocation *src* with respect to its allocation state. The slice is computed using a backward traversal from the sinks K back to the source, but going only through the nodes in F_{src} that have been discovered in the forward traversal:

     $$R = \mathit{BackwardReach}(K, F_{src})$$

     The backward traversal doesn't match calls and returns, since invalid inter-procedural flows have already been filtered out in the forward traversal. In this case, the analysis proceeds to the next phase to perform guarded reachability over the slice R of the currently analyzed allocation site *src*.

5.4 Guarded Reachability Detection

The goal of this phase is to perform a deeper analysis over the value-flow graph when the previous step has determined that the current allocation site might be freed on some execution paths. Given an allocation site *src*, a relevant VFG slice R for this allocation, and a non-empty set of reachable sinks $K \subset R$, the analysis determines whether the allocated cell is freed exactly once on each program execution path. As sketched in the examples from Section 3, the analysis uses guards on the value-flow edges to reason about the possibility of heap errors.

The guards that the analysis computes are boolean formulas that include standard logical operators (and, or, not), boolean constants, and boolean variables to represent switch conditions:

$$\begin{array}{lrll} \text{Guards } g & ::= & \mathit{true} \mid \mathit{false} & \text{(constants)} \\ & \mid & g_1 \wedge g_2 \mid g_1 \vee g_2 & \text{(and, or)} \\ & \mid & \overline{g} & \text{(negation)} \\ & \mid & (e = c_i)_n & \text{(switch test)} \end{array}$$

The switch test $(e = c_i)_n$ describes an instance of the test that occurs at switch node n and tests expression e on the branch on which the test $e = c_i$ succeeds. For each switch statement n, a consistency formula C_n models the fact that one and only one of its branches can be taken:

$$C_n = \left[\bigvee_i (e = c_i)_n \right] \wedge \left[\bigwedge_{i \neq j} \overline{(e = c_i)_n \wedge (e = c_j)_n} \right]$$

The guarded reachability algorithm proceeds as follows. First, it computes guards for each edge in the relevant slice R by exploring CFG paths. Next, it aggregates VFG guards into larger formulas. Finally, it performs satisfiability tests to detect errors. We describe each of these steps below.

$$cguard(n \to m) = cg(x, n, m, \varnothing)$$
where n is of the form $x = e$

$$cg(x, n, m, E) = \begin{cases} true & \text{if } pdom(x, n, m, \varnothing) \\ cg(x, n_1, m, E) & \text{if n is not a switch and } \neg defines(n_1, x) \\ \bigvee_{i \in I_{n,x,E}} cond(n, n_i, E) \wedge cg(x, n_i, m, E \cup \{\langle n, n_i \rangle\}) & \text{otherwise} \end{cases}$$
where $I_{n,x,E} = \{i \in I_n \mid \neg defines(n_i, x) \wedge \langle n, n_i \rangle \notin E\}$

$$cond(n, n_i, E) = \begin{cases} (e = c_i)_n & \text{if } n \text{ is } switch\ e\ (..., c_i : n_i, ...) \text{ and } \nexists n_j \,.\, \langle n, n_j \rangle \in E \\ true & \text{otherwise} \end{cases}$$

$$pdom(x, n, m, S) = \begin{cases} true & \text{if } m = n \text{ or } n \in S \\ false & \text{otherwise if } n \text{ is a return node} \\ \bigwedge_{i \in I_n} \neg defines(n_i, x) \wedge pdom(x, n_i, m, S \cup \{n\}) & \text{otherwise} \end{cases}$$

Figure 4. Demand-Driven CFG Guard Computation. The predicate $defines(n, x)$ indicates that node n writes variable x.

CFG guard computation. For each value-flow edge $n \to m$, where n and m are two program points in the same function, the analysis computes a formula describing when the value defined at n flows into its use at node m. This guard is computed by exploring the structure of the control-flow graph (CFG) between points n and m. Intuitively, the computed guard summarizes the portion of the CFG between n and m.

The guard computation algorithm is performed by the function $cguard(n \to m)$ in Figure 4. This function can be efficiently implemented using dynamic programming (i.e., caching). The algorithm uses a helper function $cg(x, n, m, E)$ that maintains the variable x whose value flow is being tracked and the set of traversed edges E. The guard between n and m is the disjunction of the guards over all possible paths, where the guard of each path is the conjunction of conditions on all control-flow edges on that path:

$$\bigvee_{i \in I_{n,x,E}} cond(n, n_i, E) \wedge cg(x, n_i, m, E \cup \{\langle n, n_i \rangle\})$$

The algorithm considers only those paths that allow the value of variable x defined at node n to flow into the use at node m. Hence, paths that redefine x do not contribute to the value flow from n to m (i.e., these paths have a *false* guard). The algorithm in Figure 4 filters out control-flow paths that redefine variable x using the predicate $defines(n, x)$.

In the presence of loops, guards can grow unbounded. To avoid unbounded conjunctions that describe all possible iterations of a loop, the analysis bounds the number of loop iterations to at most one iteration. This is done by maintaining a set E of visited switch edges. The definition of $I_{n,x,E}$ filters out control-flow edges that have been already traversed, ensuring that loops are traversed at most once.

Furthermore, the analysis adds the loop exit condition only when the loop is not executed. The reason for this is that the analysis does not distinguish between different instances of the loop test: entering the loop is described by $(e = c_i)_n$, and exiting the loop after the first iteration would correspond to $\overline{(e = c_i)_n}$. Without distinguishing between the two different instances of this test, their conjunction would make this path infeasible. To avoid this, the analysis does not add the exit condition $\overline{(e = c_i)_n}$ after one loop iteration. This is formalized in the algorithm by the condition $\nexists n_j \,.\, \langle n, n_j \rangle \in E$ in the definition of *cond*.

$$vguard(n, m) = vg(n, m, \varnothing)$$

$$vg(n, m, E) = \begin{cases} true & \text{if } n = m \\ \bigvee_{n \to n' \notin E} vg'(n \to n', m, E) & \text{otherwise} \end{cases}$$

$$vg'(n \to n', m, E) = \begin{cases} vg(n', m, E \cup \{n \to n'\}) & \text{if } n \text{ is call or return node} \\ cguard(n \to n') \wedge vg(n', m, E \cup \{n \to n'\}) & \text{otherwise} \end{cases}$$

Figure 5. VFG Guard Computation.

In addition, the analysis uses post-dominance information to simplify the computation of guards. More precisely, the algorithm computes a post-dominance relation *pdom* and returns a *true* guard for the edge $n \to m$ whenever m post-dominates node n. The post-dominance relation *pdom* is computed on-demand, as shown in Figure 4. The post-dominance computation also excludes program paths that redefine variable x, which contains the value assigned at node n.

VFG guard computation. Next, the analysis computes aggregate guards for the entire value flows from the source *src* to each of the sinks. For this, the algorithm traverses the value-flow graph, accumulating guards along each path and combining guards from different paths.

The algorithm is similar to the computation of guards in the CFG, except that it is performed on the VFG and it doesn't use post-dominators. The value-flows in VFG do not exhaustively cover all program paths, hence using dominators would be unsafe. Recursion is bounded in a similar way as loops: recursive call sites can be invoked at most once. Figure 5 presents the algorithm. The result of this part of the analysis is an aggregate guard G_k for each reachable sink k, describing the path from the source to this particular sink.

Satisfiability testing and error detection. Once aggregate guards are computed for each sink, the analysis performs two tests to identify potential errors. First, it checks for potential leaks from the currently analyzed allocation. For this, it constructs the formula below and tests its satisfiability:

$$F_{\mathsf{L}} = \overline{\bigvee_{k \in K} G_k} \wedge C$$

The formula C combines the consistency formula C_n of all switch statements that appear in the guards. It is used to eliminate cases that contradict the semantics of switch statements (for instance, cases where both branches of a conditional are taken).

If the formula F_{L} is satisfiable, there may be a memory leak. The satisfying assignment A gives truth values for the branches in the formula: a true value for a branch indicates that the branch is taken, and a false value indicates that the branch is not taken. Hence, the satisfying assignment A describes the error path.

Similarly, the analysis checks for double frees by testing the satisfiability of the following formula for each pair of sinks $i, j \in K, i \neq j$:

$$F_{\mathsf{DF}} = (G_i \wedge G_j) \wedge C$$

If the formula F_{DF} is satisfiable, then a double free error has been detected and the satisfying assignment indicates the error path.

GUARDEDREACHABILITY(*source src*, *slice* R, *sinks* K)

```
1  for (each value-flow edge n → m in R)
2      compute the edge guard cguard(n → m)
3
4  for (each sink k ∈ K)
5      compute aggregate guard G_k = vguard(src, k)
6
7  Let S be the switch nodes appearing in all of G_k
8  Build a switch consistency formula C = ⋀_{n∈S} C_n
9
10 if (the formula F_L = ¬⋁_{k∈K} G_k ∧ C is satisfiable)
11   then let A be a satisfying assignment
12        report memory leak for s on path A
13
14 for (all sinks i, j ∈ K s.t. i ≠ j)
15     if (the formula F_DF = G_i ∧ G_j ∧ C is satisfiable)
16       then let A be a satisfying assignment
17            report double free for s on path A
```

Figure 6. Error Detection via Guarded Graph Reachability.

The overall guard reachability and error detection algorithm is summarized in Figure 6.

5.5 Extensions

We present several extensions aimed at improving the basic analysis presented so far by improving its precision, efficiency, or its error reports.

5.6 Allocator Functions

Applications often use *allocator functions* that return fresh heap cells allocated in their bodies. Examples include allocation wrapper functions that extend the standard `malloc()` by testing the returned value against null and aborting the program if allocation has failed; or initializer functions that allocate fresh data structures and return them to their callers. If an allocator function is called multiple times, then the cell created in the allocator body will flow into each of the allocator's call sites. As a result, slices can get larger and error reports can become complex.

To keep the slices and error reports small and easy to understand, we enhance the algorithm to automatically detect allocator functions and to use knowledge about allocators to construct smaller slices. More precisely, for each detected allocator function the analyzer builds: 1) one slice per call site to the allocator, treating the call site as a source node; and 2) one additional slice showing the flow of values inside the allocator itself, treating the allocator return point as a sink node in that slice.

An allocator function is a function that may return a fresh heap cell created in its body. We do not require that allocators always return fresh cells: they might return non-fresh values on some executions, and fresh cells on others. However, the slices starting at their call sites only refer to those executions where they returned a fresh value. Allocator functions can be nested, that is, cells created inside them might be generated by inner allocators.

Detecting allocator functions is performed during the forward traversal of the VFG in the slicing phase, in Section 5.3. If the traversal for a source allocation *src* in a function f reveals a value flow path from src to the return variable n^f_{ret} of function f, and that path consists only of local variables and parameters, then function f is marked as an allocator. All of the call sites of f are marked as allocation sites and the algorithm will subsequently analyze them, building a slice for each of them.

5.6.1 Constant Tests

As discussed in Section 3, tests comparing the allocated value against `NULL` can be replaced with an appropriate constant truth value. For instance, if the currently analyzed allocation is `x = malloc()`, then test conditions of the form `(x == NULL)` can be replaced by *false* because the analysis considers only the cases where the allocation is successful.

This simplification technique can be generalized to tests of the form `(y == e)` where variable `y` definitely points to the allocated cell, but expression `e` is different than the cell. To determine that `y` points to the allocated cell, the analysis checks that the value of `x` flows into all the definitions of `y` reaching the test. To determine that expression `e` does not refer to the cell, the analysis checks that either `e` is not a variable (for instance `NULL`) or it is a variable but the value of `x` never flows into that variable. If these conditions are met, `y` and `e` are different, so the test `(y == e)` is *false*.

5.6.2 Pointer Arithmetic

Programs sometimes manipulate pointers inside (or even outside) allocated memory blocks instead of pointers to the beginning of memory blocks. Even though a freed pointer may correctly point to the beginning of its block, values may flow to the free through pointers in the middle of the block, as in the example below:

```
x = malloc();  y = x + 4;  free(y - 4);
```

To support value flows through pointer arithmetic, the VFG construction algorithm abstracts away offsets within allocated blocks. More precisely, each occurrence of a pointer arithmetic expression $x + e$ or $x - e$ in an assignment, a return expression, an actual function argument, or an argument to `free` is treated just as an occurrence of variable x. Hence, the VFG captures the flow of pointers into allocation blocks, without recording their offsets into their blocks. To keep the error detector simple and practical, we don't check that arithmetic operations from the allocation to the free cancel each other. Freeing a pointer into a block simply counts as freeing the block.

5.6.3 Formula Simplification

The guard computation from Figure 4 can lead to an exponential blowup in the size of the computed formulas when the analysis traverses code fragments that involve many branches. To maintain smaller formulas, the analysis performs on-the-fly simplifications using standard logical identities:

$$f \wedge \mathit{true} = f \qquad f \wedge \mathit{false} = \mathit{false}$$
$$f \vee \mathit{true} = \mathit{true} \qquad f \vee \mathit{false} = f$$
$$(f \wedge g) \vee (f \wedge \overline{g}) = f$$

The last identity eliminates irrelevant tests when both branches (g and $\overline{g}$) of an if statement are taken.

Even after such simplifications, formulas can still become large for programs with complex, unstructured control-flow. To keep the tool practical, the analysis bounds the sizes of the computed formulas by bounding the height of expression trees to a fixed value (30 in our experiments). When a formula exceeds the bound, the analysis stops analyzing the current allocation site and classifies the current allocation as one having unknown guards.

5.7 Discussion: Test Conditions and Unsoundness

In our framework, test conditions $(e = c_i)_n$ are distinguished based on the program point n where they occur. In fact, in our implementation a test in a boolean formula is an outgoing CFG edge emanating from switch statement n. Therefore: 1) tests at program points are considered different and uncorrelated even if they test the same condition; and 2) different dynamic instances of the same test are the same because they occur at the same

program point. The former can lead to imprecision, whereas the latter is a source of unsoundness, i.e., may cause the tool to miss some errors. Additional analyses could be added to handle such issues. For instance, a sound treatment would require an additional analysis to check that all dynamic instances of a given test have the same truth value during the lifetime of the currently analyzed allocated cell. However, in practice we have not encountered cases that would justify such an analysis.

Other sources of unsoundness are due to the guard computation in the presence of loops:

- Bounding loops to execute at most once implies a stronger guard from malloc to free, i.e., one that cover fewer paths to free points than the program does. Interestingly, this means that bounding loop iterations is actually sound for detecting memory leaks. However, it is unsound for detecting double frees.
- Dropping the test condition on exit after the first iteration makes the guard on this program path weaker. Hence, this causes unsoundness for the memory leak problem, but it is sound for the double free problem.

In spite of the unsoundness due loop approximations and dynamic test instances, in practice this never cause our analysis to miss bugs. We have manually checked all cases where the analysis reported that allocated cells were safely freed and all of these reports were correct.

6. Results

We have implemented an heap error detection tool *FastCheck* [1] that implements the analysis algorithm described in this paper, including the extensions discussed in Section 5.5. The analysis was implemented in Crystal [16], a program analysis for C written in Java. Crystal provides a unification-based field-sensitive pointer analysis and uses the points-to information to disambiguate indirect function calls. The control-flow graph representation provided by Crystal is fairly similar to the one presented in Section 4.1. In addition, *FastCheck* uses SAT4J [3], a freely available boolean satisfiability solver written in Java.

All of our experiments were executed on a 3.2GHz Pentium D machine with 3GB of memory running Linux. We collected statistics for programs from the SPEC2000 benchmarks [20] and for two open-source applications, the shell program *bash-3.1* and the ssh daemon *sshd-4.3p2-4*.

6.1 Analysis times

The left portion of Table 1 presents the analysis times for each application. This includes the time for building the value-flow graph and performing the reachability analysis that includes guard computation and satisfiability solving. For most benchmarks, the entire analysis takes less than a second. The most expensive application is *bash*, where the analysis takes 5 seconds. The analysis time is about one order of magnitude smaller than the time spent parsing the source files. For instance, parsing *gcc* takes about 20 seconds, and parsing *bash* takes about 40 seconds.

6.2 Memory Leak Study

A key feature of *FastCheck* is its ability to classify allocation sites into several categories and selectively report these categories the the user. The high-priority category contains likely errors; the medium-priority class contains errors that are possibly benign; whereas the low-priority category contains more complex scenarios (e.g., allocations placed into arrays) where the analysis cannot precisely reason about the safe deallocation of heap cells. In addition, the tool also identifies a set of safe allocations where it has determined that cells are correctly freed.

The categories that *FastCheck* identifies are the following:

1. *Never freed*: a cell allocated at that site is never deallocated. We divide this kind of message in two subcategories:
   (a) *local*: The allocated cell is manipulated only using local variables and parameters. In many cases, cells of this kind do not escape the function's scope, but in some cases they are passed to and returned from functions. The tool distinguishes between two sub-categories:
      i. *local allocations in main*. These allocations are likely to be live throughout the program, so the tool classifies them as possibly benign leaks.
      ii. *local allocations in functions other than main*. These are likely errors and are reported with high priority.
   (b) *not local*: The allocated cell is stored in some structure, array, pointer, or a global. Since such cell may be live through the rest of the program, the tool classifies them as possibly benign leaks.
2. *Freed, precisely known*: the path from the allocation to each deallocation is described by our tool using guards. In particular, the allocated cell cannot have pointer aliases and its value flows only through copy assignments, method calls and return statements. We further divide this class into two sub-categories:
   (a) *always*: the allocated cell is always deallocated. The analysis can derive that the condition to deallocate the cell is either *true* or a simple test, checking if the allocation returned a non-null value. These allocations are safe.
   (b) *conditional*: the allocated cell might leak, and the error is reported with high priority. The analysis derives a guard under which the cell is not deallocated, querying the SAT solver gives a possible scenario where the leak might occur.
3. *Freed, but unknown*: the allocated cell may be freed, but the analysis can't derive a precise condition under which it is deallocated. This can happen when the allocated cell is only copied through local variables, but the size of a guard formula exceeds the fixed bound; or, when the cell is simply not local, i.e., its address is stored in a structure, array or global variable.

The right portion of Table 1 summarizes the results for each application. The "leak messages" section shows the total number of source allocations, and the number of warnings (total, true, and false warnings) issued by the tool in the standard mode of operation, where only the high-priority errors are reported. The number of source allocations include all malloc sites and all calls to allocator functions, as discussed in Section 5.6 (the breakdown between malloc sites and calls to allocators is shown next in Section 6.6). The following portion labeled "messages by category" provides a detailed breakdown of the allocations by category. In the default setting, only the columns shown in bold are reported. Our tools provides additional flags for reporting allocations the other categories.

Error Reporting Methodology. For each allocation site, *FastCheck* provides the following information to users:

- An error message at the console indicating the location of the allocation site in the source file, and the error path as a sequence of branches that lead to the error.
- An html file that highlights the source lines of code in the relevant slice, as well as the branches that lead to the error;

[1] Available at: http://www.cs.cornell.edu/projects/crystal/fastcheck

		Analysis Times (sec)			Leak Messages				Messages by Category						
									Never Freed			*Freed*			
												Known		*Unknown*	
Programs	Size (Kloc)	Building VFG	Reachability Unguarded	Reachability Guarded	Total Sources	Leak Warn.	Leak Bugs	False pos.	not local	local main	local other	always	cond	local	not local
ammp	13.3	0.14	0.01	0.15	37	20	20	0	10	0	**0**	5	**20**	0	2
art	1.3	0.02	0.01	0.01	11	1	1	0	9	0	**1**	1	**0**	0	0
bzip	4.6	0.05	0.01	—	10	0	0	0	5	1	**0**	0	**0**	0	4
crafty	18.9	0.19	0.01	—	12	0	0	0	0	1	**0**	0	**0**	0	11
equake	1.5	0.08	0.01	—	29	0	0	0	29	0	**0**	0	**0**	0	0
gap	59.5	1.75	0.11	0.01	2	0	0	0	1	0	**0**	1	**0**	0	0
gcc	205.8	2.22	0.02	0.09	126	37	35	2	43	0	**25**	17	**12**	9	20
gzip	7.8	0.07	0.01	—	5	0	0	0	2	1	**0**	0	**0**	0	2
mcf	1.9	0.04	0.01	—	3	0	0	0	0	0	**0**	0	**0**	0	3
mesa	49.7	0.39	0.02	0.11	133	2	0	2	6	0	**0**	29	**2**	1	95
parser	10.9	0.16	0.01	—	1	0	0	0	0	0	**0**	0	**0**	0	1
perlbmk	58.2	0.79	3.77	0.09	321	4	1	3	29	0	**1**	27	**3**	0	261
twolf	19.7	0.29	0.07	0.02	185	2	2	0	77	1	**2**	18	**0**	0	87
vortex	52.7	0.75	0.06	—	9	0	0	0	0	0	**0**	0	**0**	0	9
vpr	17.0	0.14	0.11	0.08	157	1	0	1	31	0	**0**	57	**1**	0	68
bash	100.0	1.53	0.50	2.46	276	3	2	1	25	0	**0**	88	**3**	5	155
sshd	48.7	1.11	0.06	0.13	454	3	2	1	67	6	**1**	243	**2**	0	135
Averages	—	55%	27%	18%	—	—	—	14%	19%	1%	2%	27%	2%	1%	48%

Table 1. Analysis times, leak warnings statistics, and leak messages classification. Only cases in bold are reported as high priority errors.

- A graphical representation of the value-flows using the *dot* [9] format, showing the relevant slice for this allocation.

We found the html and *dot* error reports extremely useful in quickly understanding the errors. The slices for all of the warnings were very small: the average size of the relevant slice was two nodes, and the biggest slice in the reported warnings had 11 nodes. As a result, error reports were small and easy to understand. There are two main reasons for relevant slices to be small. First, they only include value flows that lead to frees; all other flows are not included in the slice and are not reported. For instance, passing a pointer to an allocated cell to a function that just accesses the contents of that cell would not be highlighted in the report. In the extreme case, slices for allocations that are never freed consist of one single node, the allocation site. Second, detecting allocator functions and using one slice per allocator call site breaks down larger slices into several smaller and simpler slices.

Reported Warnings. For these benchmarks, our tool has reported 73 memory leak warnings, of which 63 were actual errors, yielding a false positive rate of 14%. Overall, these warnings account for a small fraction (4%) of the total number of allocations. The distribution of warnings and errors across the benchmarks is skewed, with *ammp* and *gcc* accounting for the majority of errors.

Several of the errors referred to allocations that were never freed. Many bugs in *gcc* come from mishandling concatenation of strings; we will discuss in more detail an instance of this bug later in Section 6.3. The tool also pointed out a few cases in *gcc* where memory leaks occur when adding entries to a symbol table. The program creates a symbol string and passes it to a function that performs the insertion. However, the insertion function creates a copy of the string and doesn't deallocate the original string. The string is not deallocated in the caller either and is leaked shortly after the insertion. Applications such as *twolf* and *art* create dynamic arrays that are used locally, but are never deallocated.

Conditional leaks are cases where applications don't deallocate memory when returning from functions on error conditions. In *ammp* the error path is correspond to failed allocations of other cells, similar to the first example from Section 3. Other conditional leaks were found in *perlbmk* and *bash*. We will discuss the example bug in *bash* later in Section 6.3.

The false positives were due to several reasons: explicit reference counting and deallocation when reference counts become zero (two warnings in *mesa*); ignoring extra parameters for functions variable number of arguments (two warnings in *gcc*); not recognizing exit functions such as `execve` (one warning in *perl*); bounding the number of loop iterations(in *sshd* and *vpr*); and not recognizing unfeasible program paths two warning in *perl* and one in *bash*. The example below shows the false positive from *bash*:

```
if (lose == 0) {
   name_vector = malloc ();
   lose |= name_vector == NULL;
}
if (lose) return;
```

The tool reports a possible leak at the return point. Identifying that the path from the malloc to the return is infeasible would require reasoning about the bitwise operation after the allocation.

Other categories. The tool has identified that 27% of the allocations are safely freed. Since the tool is unsound, allocations reported to be safe might leak memory. However, this never happened for our benchmarks: we have manually inspected all of the allocations in this category and determined they were all freed correctly. About 20% of the allocations in the program were never freed. Many of these were cells placed into global variables and were live throughout the application. For this reason, we classified all of the errors in this category as possibly benign.

Allocations in the unknown category represent about half of the allocations in our benchmarks. These allocations include cells that escape to the heap, to global variables, or to aggregate structures such as arrays. Handling such cases would require more sophisticated analyses, such as shape analysis for reasoning about recursive data structures, or array analysis for reasoning about array of pointers to allocated cells. In the future, we envision a refinement-based tool that first uses the lightweight value-flow analysis in this paper to check the simpler cases, and subsequently uses more heavyweight analyses for the allocations in the unknown category.

```
     /* file "c-aux-info.c" */
53:  char* concat (char* s1, char* s2) {
54:      if (!s1) s1 = "";
55:      if (!s2) s2 = "";
56:      int size1 = strlen (s1);
57:      int size2 = strlen (s2);
58:      char* ret_val = malloc (size1 + size2 + 1);
59:      strcpy (ret_val, s1);
60:      strcpy (&ret_val[size1], s2);
61:      return ret_val;
62:  }
...  ...
281: char* gen_formal_list_for_func_def(tree fdecl) {
282:    char* f_list = "";
...     ...
290:    while (fdecl) {
291:        if (...)
292:            f_list = concat(f_list, ",");
293:        f_list = concat(f_list, formal);
...         ...
302:    }
303:    return f_list;
304: }
```

Figure 7. Example bug from *gcc*: the cell pointed by `f_list` is leaked on subsequent calls to `concat`.

6.3 Memory Leak Examples

We discuss two example memory leaks from *gcc* and *bash* in more detail. The code for these examples is slightly reformatted for clarity purposes.

6.3.1 Allocation Never Freed

About 20 of the messages in *gcc* referred to string buffers that were never freed. Our tool reported messages of the form:

```
Error: "c-aux-info.c", line 292:
       allocation never freed:
       f_list = concat(f_list, ", ");
```

Figure 7 presents the actual code for `concat` extracted from *gcc*. The code allocates a new string containing the concatenation of the input strings. Hence, `concat` is an allocator function. The lower portion of Figure 7 shows a function `gen_formal_list_for_func` that generates a string containing the list of formal arguments of a function declaration. The code uses `concat` concatenate strings and stores the intermediate results in `f_list`. The fresh cell allocated by the call to concat at line 292 is leaked when the subsequent assignment at line 293 overwrites `f_list`.

6.3.2 Conditional Leak Bug

Figure 8 presents a code fragment from *bash* where a conditional leak occurs. There are two possible deallocation sites for the cell allocated at line 354: at line 360, when `rl_translate_keyseq()` returns a non-zero value; and at the end of the program, at line 427. However, the program leaks this cell when returning at line 371.

The guarded flow graph generated by the analysis is also presented on Figure 8. The special variable `c359` is used to denote the result of calling `rl_translate_keyseq` at line 359. The condition on the right edge of the graph describes situations on which the program doesn't exit at line 371. Finally, the error message shown below the graph presents one instantiation of program conditions that reaches line 371, causing a leak. For this to happen, the program must enter the loop in line 366, and one of the conditions in line 370 must be true.

Tool	Size KLOC	Speed KLOC/s	Bug Count	FP (%)
Saturn [21]	6822	0.05	455	10%
Clouseau [11, 12]	1086	0.5	409	64%
Contradiction [14]	321	0.3	26	56%
Shape analysis [10]	68	0.6	38	60%
This analysis	671	37.9	63	14%

Figure 9. Comparison with other memory leak detectors.

Program	Sites checked	Total checks
ammp	2	6
mesa	5	14
bash	14	165
sshd	28	48

Table 2. Total double-free checks performed. No checks were performed in the other programs.

6.4 Comparison to Other Tools

Figure 9 presents a comparison of our memory leak analysis to other published tools. The Speed column indicates the number of analyzed lines of code per second; and the FP column shows the false positive ratio. We want to emphasize that speed numbers should be regarded only as rough estimates, since these tools have been implemented in different languages (C and Java), executed on different machines, and applied to different benchmarks. The table shows that our analysis provides a good trade-off between speed, and effectiveness of finding errors with a low rate of false warnings.

6.5 Double-Frees Study

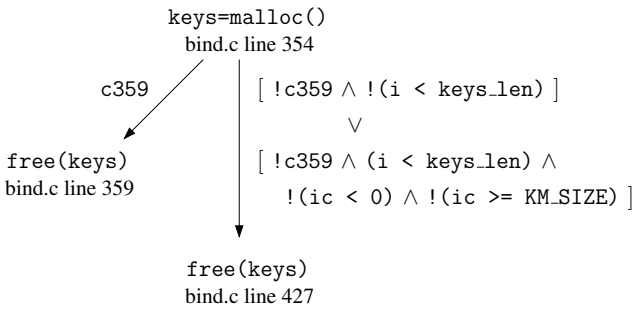
We have also tested allocations in our benchmarks for double-frees. Tests were performed only for allocations that flow into more than one free; in this case, each pair of frees was tested using our boolean satisfiability procedure. Allocations with at most one free were not tested. This may be unsound in the presence of loops, as a single cell could be deallocated multiple times at the same site, on different iterations of a loop. Table 2 shows the number of checked allocations and the total number of tests. Most sites in *sshd* have exactly 2 frees, hence only one test per allocation is needed. In contrast, one site in *bash* is freed at 9 different locations, yielding 36 double-free tests. The tool correctly reported that none of the tested sites was double-freed.

6.6 Function Behavior Study

We also evaluated the distribution of functions that are allocation and deallocation wrappers. Free wrappers are functions that deallocate a value supplied in an argument. Table 3 presents the results. Overall, very few functions are marked as wrappers. This indicates that local cells are typically allocated and freed in the same function, unless the allocated cell is stored in a non local expression. On average, only 1.8% of the functions are considered allocators, and less than 0.1% are considered deallocation wrappers. Table 3 also shows the number of source nodes that are calls to allocation wrappers, noticeable benchmarks such as *perlbmk* allocate most of their cells using allocation wrappers.

Finally, some of the applications use custom memory allocation. A good is *parser*: it allocates a large block at the beginning and manages the block internally. Our tool is not aimed at recognizing the internal custom allocator functions, and simply treats this program has having one single allocation site.

```
       /* file "bind.c" */
338:   int rl_generic_bind(...) {
...      ...
354:     keys = (char *) xmalloc (...);
...      ...
359:     if (rl_translate_keyseq (...)) {
360:          free (keys);
361:          return -1;
362:     }
...      ...
366:     for (i = 0; i < keys_len; i++) {
367:          unsigned char uc = keys[i];
368:          int ic;
369:          ic = uc;
370:          if (ic < 0 || ic >= KM_SIZE)
371:              return -1;
...           ...
426:     }
427:     free(keys);
428:     return 0;
429:   }
```




```
Error: "bind.c", line 354: allocation leaks memory
       keys = xmalloc(...);
Leak path: ! c359          bind.c, line 359
           i < keys_len    bind.c, line 366
           ic < 0          bind.c, line 370
```

Figure 8. Warning from *bash*: cell allocated at line 354 is leaked at line 371. Analysis generates message which includes the affected source code and relevant slice of the guarded VFG.

Program	Total Func.	Alloc Wrappers	Free Wrappers	Allocation Sites	Calls to Allocators
ammp	197	0	0	37	0
art	44	0	0	11	0
bzip2	92	0	0	10	0
crafty	127	0	0	12	0
equake	45	0	0	29	0
gap	872	0	0	2	0
gcc	2271	13	1	53	73
gzip	128	0	0	5	0
mcf	44	0	0	3	0
mesa	1124	22	2	67	66
parser	342	0	0	1	0
perlbmk	1094	20	1	4	317
twolf	209	6	2	2	183
vortex	941	0	0	9	0
vpr	290	17	6	2	155
bash	2200	41	0	141	135
sshd	940	60	4	118	336

Table 3. Wrapper functions and distribution of source nodes.

7. Related Work

In recent years there has been a large body of research devoted to checking safety properties. We classify the related work into flow-sensitive and flow-insensitive approaches, and restrict the discussion to the techniques that are most relevant to our work.

Flow-sensitive Safety Checking. A standard approach to safety checking is to describe the desired safety property as a finite state machine, and then perform a flow-sensitive analysis of the program to track the current state at each program point, and detect violations as transitions to the error state. There are many existing systems that implement this approach, either using dataflow analysis, as Metal [7] or ESP [5]; or using model checking, as SLAM [2] or [22]. These approaches are generic, meaning that they allow users to specify arbitrary state machines. The analyzer then uses a generic dataflow engine to check for property violations. The flow-sensitive state tracking process can be enhanced in many ways, for instance using context-sensitive inter-procedural analysis, using path-sensitive analysis [5], or predicate refinement [2]. These tools can be used to check for memory leaks by instantiating the state machine with a machine consisting of two states, and transitions from the first to the second on allocation, and from the second to the first on deallocation. In contrast to the dataflow approaches, our analysis reasons about heap violations using a sparse value-flow graph representation, not by walking the control-flow graph. The use of dataflow analysis in our system is restricted only to identifying definition-use chains and to the demand-driven computation of guards (essentially a mini-dataflow analysis between two program points).

In addition to the above techniques, several tools have been specifically developed for finding heap errors such as memory leaks, or accesses through dangling pointers. Clouseau [11, 12] is a leak detection tool that uses a notion of pointer ownership to describe those variables responsible for freeing heap cells, and formulate the analysis as an ownership constraint system. Saturn [21] reduces the problem of memory leak detection to a Boolean satisfiability problem, and then use a SAT-solver to identify potential errors. Both Saturn and Clouseau are essentially constraint formulations of a dataflow analysis problem: they encode the dataflow transfer functions as constraints in their system. A complication in these systems is that it becomes more difficult to map constraint system violations back to the program code, and report errors that can be easily understood by the programmers.

None of the above techniques is capable of reasoning about leaks and other heap errors in recursive data structures, such as lists or trees. Shape analysis refers to the class of dataflow analyses that use more advanced heap abstractions and are capable of reasoning about individual heap cells in recursive structures. Dor and Sagiv [6] 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 routines. However, TVLA has not been used for error detection in larger programs. Hackett and Rugina [10] use a shape analysis that tracks single heap cells to identify memory leaks. More recently, Orlovich and Rugina [14] have proposed a novel approach to memory leak detection where the analy-

sis assumes the presence of errors and then performs a backward dataflow analysis to disprove their feasibility. The analysis presented in this paper is not designed to reason about manipulation of recursive structures; it trades this imprecision for the efficient detection and reporting of heap errors in non-recursive structures.

Flow-insensitive Safety Checking Flow-insensitive analyses discard the control-flow in the program and treat the program as a set of assignments that can be performed in any order. Such analyses are simpler and more efficient, but less precise than their flow-sensitive counterparts. Common examples of flow-insensitive analyses are type-based analyses and flow-insensitive pointer analyses. A standard procedure is to compute definition-use chains, as the analysis in this paper, or an SSA representation[4] of the program before treating program assignments in a flow-insensitive manner, thus recovering a certain amount of flow-sensitivity. For the purpose of this discussion, we still classify them as flow-insensitive.

The CQual system [8] is a system that infers type qualifiers in C programs using a set-constraint formulation. The system can be used to perform tainting analysis, i.e., to determine that a value from a tainted source (such as reading data from the network) never flows into a sink that must not be tainted (such as a critical kernel structure). Livshits and Lam [13] propose a similar tainting analysis, but using an augmented SSA form called IPSSA. Both analyses are context-sensitive; in addition, the latter analysis uses a guarded SSA form [19] that annotates value-flow edges at φ-nodes with branch conditions. The tainting analysis problem is a source-not-sink problem, aiming at checking that source values never flow into a sink. This is a fundamentally different than the source-sink problem discussed in this paper, and none of the above approaches can be used for the memory leak problem. This is because their value-flow representation is not powerful enough to determine that a source value flows into the sink on all paths. Although in the system of Livshits and Lam, value-flow edges of φ-nodes are tagged with branch conditions, the flows via assignments are not, therefore their analysis cannot solve source-sink problems. For instance, the examples from Section 3 do not contain φ-nodes from the source allocations to sink frees, hence none of these value-flows would be guarded.

Snelting et al. [17] present a tainting analysis based on value-flows using a program dependence graph (PDG) structure. They also annotate value flows in the dependence graph using path conditions. However, they use path conditions to improve the precision for a source-no-sink problem. In contrast, we use path conditions to solve source-sink problems such as memory leak detection.

8. Conclusions

We have presented a new analysis for detecting memory leaks. The analysis uses a sparse representation of the program in the form of a value-flow graph. The analysis reasons about program behavior on all paths by computing guards for the relevant value-flow edges. The approach makes the analysis efficient, and allows it to generate concise, easy-to-understand error messages.

References

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, Snowbird, Utah, June 2001.

[2] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 2002.

[3] Daniel Le Berre and Anne Parrain. SAT4J: A satisfiability library for java. URL: http://www.sat4j.org/.

[4] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, Austin, TX, June 1989.

[5] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[6] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Checking cleanness in linked lists. In *Proceedings of the International Static Analysis Symposium*, Santa Barbara, CA, July 2000.

[7] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.

[8] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems*, 28(6):1035–1087, November 2006.

[9] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[10] Brian Hackett and Radu Rugina. Shape analysis with tracked locations. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, Long Beach, CA, January 2005.

[11] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, San Diego, CA, June 2003.

[12] David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In *Proceeding of the International Conference on Software Engineering*, Shanghai, China, May 2006.

[13] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Helsinki, Finland, September 2003.

[14] Maksim Orlovich and Radu Rugina. Memory leak analysis by contradition. In *Proceedings of the International Static Analysis Symposium*, Seoul, Korea, August 2006.

[15] Thomas Reps, Susan Horowitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, San Francisco, CA, January 1995.

[16] Radu Rugina, Maksim Orlovich, and Xin Zheng. Crystal: A program analysis system for C. URL: http://www.cs.cornell.edu/projects/crystal.

[17] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*, 15(4):410–457, October 2006.

[18] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[19] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[20] Joseph Uniejewski. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[21] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, September 2005.

[22] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004.