Safe Memory-Leak Fixing for C Programs

Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, Hong Mei
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, School of Electronics Engineering and Computer Science,
Peking University, Beijing, 100871, P. R. China
{gaoqing11, xiongyf04, miyq13, zhanglu, xiebing, meih}@sei.pku.edu.cn, {weikunyang, zhouzhaoping}@pku.edu.cn

***Abstract*—Automatic bug fixing has become a promising direction for reducing manual effort in debugging. However, general approaches to automatic bug fixing may face some fundamental difficulties. In this paper, we argue that automatic fixing of specific types of bugs can be a useful complement.**

This paper reports our first attempt towards automatically fixing memory leaks in C programs. Our approach generates only safe fixes, which are guaranteed not to interrupt normal execution of the program. To design such an approach, we have to deal with several challenging problems such as inter-procedural leaks, global variables, loops, and leaks from multiple allocations. We propose solutions to all the problems and integrate the solutions into a coherent approach.

We implemented our inter-procedural memory leak fixing into a tool named $LeakFix$ and evaluated $LeakFix$ on 15 programs with 522k lines of code. Our evaluation shows that $LeakFix$ is able to successfully fix a substantial number of memory leaks, and $LeakFix$ is scalable for large applications.

I. Introduction

Recently, a lot of research effort has been put into automatic bug fixing [1, 2, 3, 4, 5, 6]. Given a violated correctness condition, these approaches try to modify the code to satisfy the condition. However, automatic bug fixing faces two fundamental difficulties. First, the correctness condition is often under-specified in practice. Current approaches usually rely on test cases or assertions, both of which are usually inadequate in the code, and rarely ensure correctness. Second, the search space is often very large (even infinite), and it is very difficult to find an efficient fixing algorithm in general. Current approaches usually run in hours and may produce undesirable fixes.

Due to these fundamental difficulties, we argue that instead of general bug fixing, we should also study fixing approaches for specific types of bugs. In this paper we report our attempt of developing an approach that fixes a specific type of bugs – memory leaks in C programs. There are several reasons to choose memory leaks as our target problem. First, dealing with memory leak is an important problem in software development. While many approaches [7, 8, 9, 10, 11, 12, 13] have been proposed to detect memory leaks, it is still difficult to fix a memory leak [14, 15]. Second, memory leaks cannot be easily handled by general bug-fixing approaches, as we cannot easily specify the condition of "no leak" as an assertion or a test case. Third, the "no leak" condition is general. We can build it into our approach without relying on user-defined test cases and assertions. Fourth, the problem of fixing memory leaks takes a much simpler form than fixing general bugs, as the main task is to find a suitable location to insert the deallocation statement.

We sincerely thank Zhenbo Xu and Jian Zhang at Institute of Software, Chinese Academy of Science, for their advice on implementation.

This work is supported by the National Basic Research Program of China under Grant No. 2014CB347701, and the National Natural Science Foundation of China under Grant No. 61202071, 61225007, 61421091, and 61332010.

Yingfei Xiong is the corresponding author.

```
1  record* p;
2  int bad_record_id;
3  while (has_next()) {
4    if (search_condition != null)
5      p = get_next();
6    else
7      p = search_for_next(search_condition);
8    if (is_broken(p)) {
9      bad_record_id=p->id;
10     break;
11   }
12   free(p);
13 }
14 ... // operations on bad_record_id
15 return;
```

Fig. 1. The code of procedure `check_records`

To understand the difficulty of fixing a memory leak, let us take a look at an example program in Fig. 1. This is a contrived example mimicking recurring leak patterns we found in real C programs. Procedure `check_records` checks whether there is any bad record in a large file, and the caller could either check all records, or specify a search condition to check only part of records. In this example, both `get_next` and `search_for_next` will allocate and return a heap structure, which is expected to be freed at line 12. However, the execution may break out the loop at line 10, causing a memory leak.

Many existing detection approaches report only the allocation that may be leaked, which may be far from the place where the leak occurs. In this example, the leaked allocation will be inside the procedures `get_next` and `search_for_next`, where the actual leak occurs in another procedure. Fastcheck [9] is a detection approach that gives a path where the memory is leaked. This noticeably reduces the search space for identifying the leak, but it is still difficult to correctly fix the leak. To fix the leak, we have to insert a deallocation statement satisfying the following conditions. (1) In any execution, the memory chunk has to be allocated before the deallocation. (2)

There is no double free: no other deallocation will free this memory chunk. (3) The memory chunk will not be used after the deallocation. The developer needs to have a comprehensive understanding of the code to find a suitable location and a suitable pointer to be freed. In this example, if the developer chooses to insert a deallocation of `p` before line 10, s/he has to look into the procedure `get_next` and `search_for_next` to make sure that they return allocated memory chunks in all cases. Also, s/he has to ensure that the code after the loop has no deallocation or references to `p`. If `p` is passed out the current procedure, the caller procedure should also be examined.

In our approach we try to ensure that all the fixes we generate satisfy the above three conditions, collectively known as the *safety* of fixes. We argue that safety is essential here because, if an approach cannot guarantee the safety of the fixes, developers still have to manually review all fixes to identify faulty fixes, and arguably this review process would not be noticeably easier than directly fixing the leaks, because the developers still have to understand the logics of the code. On the other hand, if the safety is guaranteed, the developers could simply run the approach and trust the fixed code.

It is not easy to meet the goal above, as the design of the approach faces several challenging problems, namely inter-procedural leaks, global variables, leaks of multiple allocations, and loops, which will be discussed in detail in Section II. Furthermore, we cannot rely on the techniques in existing detecting approaches to deal with these challenges because, as far as we are aware, all existing detection approaches report false positives.

Since leak detection is strongly interrelated with pointer analysis, existing leak detection approaches often build their approaches as a special pointer analysis process [9, 12]. Recently, there are significant improvements on the efficiency of pointer analysis [16, 17, 18], so it makes sense to reuse existing pointer analysis algorithms as black boxes. An important design choice is that we treat pointer analysis as an independent component in our approach, so that we can easily reuse different pointer analysis algorithms. This design choice also greatly simplifies our approach, as many complexities are transferred to pointer analysis. We currently use DSA [16], an inter-procedural, flow-insensitive, context-sensitive with heap cloning, field-sensitive, unification-based, and SSA-based pointer analysis in our implementation. Nevertheless, our approach is not specific to a particular pointer analysis implementation. Pointer analyses of different sensitivities can be used to increase the precision of the analysis or to improve the analysis speed.

The main contributions of this paper are summarized as follows:

- The first characterization of the problem of memory-leak fixing in C programs, including three conditions for safe fixes.
- A novel approach to automatically detecting and fixing leaks in C programs, ensuring the safety of the fixes and handling various challenging cases.
- An implementation of our approach, and a non-trivial empirical study on SPEC2000 to evaluate the performance of the tool, which shows promising results.

The rest of the paper is organized as the follows. Section II defines the problem of memory-leak fixing, highlights challenges, and describes the basic idea of our approach. Section III presents the details of our approach. Section IV presents the implementation of our approach. Section V presents the evaluation of our tool, *LeakFix*. Section VI discusses further issues for memory-leak fixing. Section VII reviews existing approaches to handling memory leaks and automatic bug fixing. Section VIII concludes the paper.

II. Approach Overview

A. Problem Definition

There are many ways to modify the code to fix memory leaks. To minimize the disruption to the user code logic, we focus on inserting only one `free(`exp`)` statement, where exp is an expression that evaluates to a pointer. The safety of such a fix is defined below.

Definition 1. [Safe fixes] A safe memory leak fix is an insertion of a deallocation statement s into the program such that the following three conditions are satisfied for any execution path where s is executed.

1) A memory chunk c is allocated before the execution of s, which releases c;
2) there is no other deallocation statement that releases c;
3) there is no use of c after the execution of s, i.e., c is dead at the insertion point.

Besides safety, a further requirement is that we should insert the deallocation statement as early as possible, so that the memory chunks are deallocated as soon as we do not need them.

B. Basic Idea

We first outline the basic idea of our approach, which will be refined and improved when we discuss challenging problems in the next sub section.

As we cannot rely on existing approaches to detecting memory leaks, our approach tries to identify and fix memory leaks at the same time. For each memory allocation statement in the code, we check whether there is any leak on some path, and insert a deallocation statement to fix the leak.

Before we detect and fix the leaks, we first perform pointer analysis on the whole program. The pointer analysis we perform is an inter-procedural, flow-insensitive, context-sensitive with heap cloning, field-sensitive, unification-based, and SSA-based pointer analysis, so that we get an SSA-based points-to graph for each procedure. Each node in a points-to graph represents a memory object, and each edge in a points-to graph represents the points-to relations between the objects.

As a memory chunk may be allocated in one procedure and used in various other procedures, the first step is to decide, given a memory allocation, in which procedure we should check and fix its leak. Based on memory usage, we

could identify three types of procedures related to a memory allocation m:

- *Alloc procedures.* This type of procedures may pass to its caller a newly allocated memory chunk.
- *Use procedures.* This type of procedures may access a memory chunk, directly or indirectly.
- *Dealloc procedures.* This type of procedure may free a memory chunk, directly or indirectly.

A procedure can belong to multiple types, or none, if the procedure is not related to the memory allocation. To ensure safety of fixes, we require that our identification covers all possible use procedures and dealloc procedures. In our running example, `search_for_next` and `get_next` are alloc procedures, and `is_broken` is a use procedure for both allocations. Note `free` is a special procedure for recognizing deallocations, and is not used for procedure type identification.

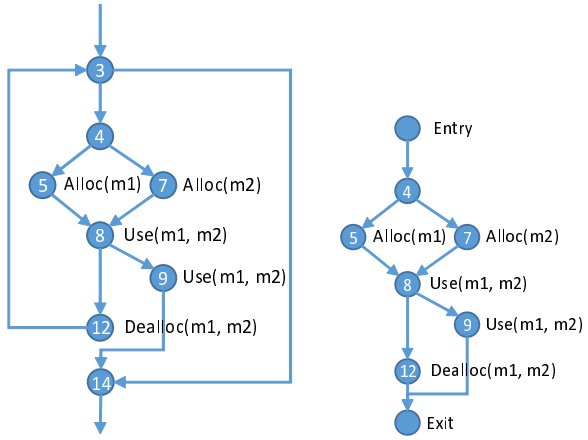
After we identify these procedures, we could identify and fix the leaks of m in the procedure which calls an alloc procedure of m but itself is not an alloc procedure of m. This is based on Xie and Aiken's rule [8]: any allocation in a procedure P that does not escape from P is leaked if it is not deallocated in P. In our example, we should check and fix leaks in the procedure `check_records`. The three types of procedures can be identified by analyzing the points-to graphs.

When we have identified these types, we could detect and fix leaks within the procedure. We do this by abstracting the program into an abstract control flow graph (CFG), as shown in Fig. 2(a). In this abstract control flow graph, we keep only information related to memory usage and deallocation. We recognize statements in the same way as procedures and mark nodes as Alloc, Use and/or Dealloc on the nodes. Identifiers `m1` and `m2` represent memory allocations to be analyzed within this procedure. Same as procedures, we require that the set of use and dealloc nodes to be complete. Note that all dealloc nodes are naturally use nodes, and we do not show the use notation for simplicity.

With this graph, our task is to find an edge e where, given an allocated memory chunk m, (1) we can construct at e an expression exp that always evaluates to a pointer to m, (2) none of the paths covering e contains `Dealloc(`m`)`, and (3) none of the outgoing paths from e contains `Use(`m`)`. If such an edge exists, we can insert a `free(`exp`)` statement at e. We refer to the three conditions as *basic edge conditions*. The basic edge conditions correspond to the three conditions of safe fixes. Regarding the first condition, if there is an expression exp that always returns m, m must have been allocated before reaching this edge in all execution paths. The rest two conditions is guaranteed because we require completeness of use and dealloc nodes.

The three conditions can be checked using standard techniques. The first condition can be checked based on points-to graphs. The rest two conditions can be checked with dataflow analyses.

However, it is not easy to realize this basic idea. There are several challenging problems that need to be coped with, as described in the next section.



(a) Whole procedure (b) Extracted loop

(The numbers in the nodes are the respective line numbers in the original program. The memory chunk returned by `get_next` is denoted as `m1` and the memory chunk returned by `search_for_next` is denoted as `m2`.)

Fig. 2. Abstract CFGs of `check_records`

C. Challenging Problems and Solutions

1) Pointer Analysis Enhancements: To check basic edge conditions, we need to identify an expression that always evaluates to a pointer to m. In other words, we need to answer must-alias queries. However, modern pointer analysis algorithms are all points-to analysis based on singe static assignment form (SSA) [18, 16, 17, 19], which builds SSA-based points-to graphs. We cannot directly apply these pointer analysis algorithms to our approach, and must perform several enhancements as follows:

First, in SSA, each programmer-specified use of a variable is reached by exactly one assignment of that variable, and the program contains ϕ functions that distinguish values of variables transmitted on distinct incoming control flow edges [20]. In SSA-based pointer analysis, the program is first translated to SSA form and the points-to graphs contain only the information about the SSA program. Unlike leak detection approaches [11, 12], which could directly work on converted SSA program, our approach needs to modify the source code to fix leaks, and cannot work on the converted SSA program.

To utilize the modern SSA-based pointer analysis algorithms, we modify the SSA conversion algorithm and record a traceability links between original variables and SSA variables during the conversion. More concretely, the traceability links record the corresponding SSA variables of the original variables at each node program point. With this traceability links, we can map an SSA-based points-to graph to a points-to graph on the original program. Note that this conversion utilize the partial flow-sensitivity brought by the SSA form. Though the original analysis is flow-insensitive, the converted points-to graphs are different at each CFG node.

Second, to obtain a pointer that always points to m at a certain program point, we can identify whether there is a node in the points-to graph that has only one successor m. However, modern pointer analysis algorithms are usually only

sound with respect to valid pointers. In other words, besides pointing to its successors on the points-to graph, a pointer may also be uninitialized, or be null. If we consider the possibility of being uninitialized or null, we can never identify a pointer only pointing to m.

To deal with the problem of uninitialized values, we perform a context-insensitive inter-procedural dataflow analysis to find which variables are definitely initialized. In intra-procedure analysis, the lattice elements are sets of pointers in the program that definitely have been initialized. A pointer may be a global variable, a local variable and a set of heap variable. A global variable is represented by its name. A local variable is represented by its procedure name and its name. A heap variable is represented by its allocation statement and, if the allocated memory chunk is a struct, a sequence of field names. The meet operator is set intersection. The transfer functions adds a pointer p to the set when an assignment statement $lval$=$rval$ is executed, where $lval$ can be determined to be definitely p using the current points-to graph and the known initialized variables. It is easy to show the transfer functions are all monotone because variables are only added. The analysis is performed forwardly. We extend the intra-procedural analysis into inter-procedural using standard bottom-up-summary-based approach [21].

The null value is more interesting. According to the C standard, `free(null)` has no effect, and thus we could safely ignore the fact that a pointer may point to null. The downside of this strategy is that we may add useless deallocation statement. When p is always `null`, a statement of $free(p)$ is useless. As a matter of fact, we find this strategy leads to a significant number of useless deallocations to be added, because in the code the following pattern is very common.

```
p=malloc(); if (p != null) {...;free(p);}
```

Unless the pointer analysis is path sensitive, `else free(p);` will be appended to above program, but this deallocation is useless.

To avoid too many useless deallocations, we perform a simple refinement to the points-to graphs. We recognize basic null tests in conditional statements such as `if(exp)`, `if(exp==null)`, or `if(exp!=0)`. If we can definitely determine `exp` evaluates to a node n in the points graph, we remove all outgoing edges of n in the points-to graphs on the corresponding branch where `exp` should be `null`. As will be shown later, this refinement leads to the insertion of zero useless deallocation statement in our evaluation.

2) Procedure Identification: It is not easy to build an algorithm for identifying these procedures. First, there are many ways for a procedure to pass a pointer to a caller, e.g., via return value, pointers to pointers, heap structures, and global variables. The algorithm has to consider all of them. Second, with pointers, a program may access allocated memory chunks indirectly. For example, a program may take a parameter `p`, but read `p->p1->p2`. Thus, we cannot only track how pointers are passed between procedures, but have to track how a program uses the pointers. Third, the types of a procedure may depends on the types of other procedures. For example, procedure a calls procedure b, where b uses an allocated chunk m_1, then a is also a use procedure of m_1 because of calling b. The algorithm must identify the types for all procedures in a convergent and terminating manner. It is not easy to meet all the conditions above. As a matter of fact, though some detection algorithms are locally sound [22], no detection algorithm is sound on the inter-procedural level as far as we know.

To address these issues, we build our procedure identification algorithm on top of points-to graphs. This design choice leads to a simple algorithm, as we transfer the complex part of the analysis to pointer analysis. More concretely, we only needs to build flow-insensitive procedure summaries of four sets, which record which nodes on the points-to graph are allocated, escaped, used, and freed in the current procedure. From the summaries we could easily identify the types of a procedure with respect to a particular allocation.

Our summaries are context-insensitive, but we could easily extend the summaries to be context-sensitive using the standard cloning technique.

3) Global Variables: To ensure safety of fixes, when a procedure `f` assigns an allocated memory chunk to a global variable, we have to consider `f` as an alloc procedure because the memory chunk escapes from `f`. However, if this memory chunk is not freed anywhere, all callers of `f` will also be identified as alloc procedures, and so do their callers, etc. As a result, we can only fix this leak within the `main` procedure. Though this fix is technically safe, it may not be efficient because the deallocation may be too far from its last use. This problem does not exist in leak detection algorithms [9, 12], as a detection in `main` still detects the leak.

To overcome this problem, given a procedure that passes an allocated memory chunk via global variables, we identify it as an alloc procedure only when any of its direct and indirect callers will use the returned memory chunk. When none of the callers uses the escaped memory chunk, the memory chunk is effectively not escaped and it is safe for us to free it within the procedure.

We perform this identification on the call graph and check whether any caller can reach a use/dealloc procedure on the call graph. More concretely, we first identify all use and dealloc procedures and then examine the call graph. Let p be the procedure that passes an allocated chunk via global variables. We identify p as alloc only when there exists another procedure c and u, where u is a use or dealloc procedure, u and p are both reachable from c on the call graph, and there is at least one path from c to u that does not go through p.

4) Multiple Allocations: Existing techniques on memory leaks adopt a per-allocation basis: given one allocation statement, we check whether this allocated memory chunk leaks. Our basic approach follows the same basis.

However, although the per-allocation basis is enough to detect leaks, it is not enough to fix leaks. In our running example, the leak should be fixed before line 10, but our basic approach cannot fix it because `p` points to both allocated

chunks and we cannot find an expression that always points to one allocated memory chunk. Based on our experience, this kind of code pattern, where two allocations are assigned to the same pointer at different paths and used as one, is common in C programs.

To cope with this problem, instead of considering one allocation at one time in basic edge conditions, we consider a set of allocations together, called *set-based edge conditions*. More concretely, we try to find a set of allocated memory chunks M and an edge e, where (1) there is an expression exp at e where in any execution, the memory chunk pointed by exp is a member of M, (2) none of the path covering e contains `dealloc(`m`)` for any $m \in M$, and (3) none of the outgoing path from e contains `use(`m`)` for any $m \in M$. The safety of the three conditions can be reasoned in a way similar to basic edge conditions. Given any execution path, the memory chunk pointed by exp at e must have been allocated based on the first condition, and there is no use after e and no deallocation on the whole path according to the latter two conditions.

5) Loops: With set-based edge conditions, we are still not able to fix the leak in our running example. There exists paths containing both the node of `Dealloc(m1, m2)` at line 12 as well as the edge between line 9 and 14, and we have to conservatively determine that the edge is not suitable for leak fixing. This is because our abstraction (as well as state-of-art pointer analysis techniques) distinguishes memory allocations by their code locations, and cannot distinguish memory allocations executed in different iterations in a loop.

Handling loops is in general a difficult problem in static analysis. However, in the case of memory leaks, it is rare for a program to allocate a memory chunk in one iteration and use it in another iteration. Most programs, as well as our running example, allocate and use a memory chunk in one iteration, and this memory chunk should also be deallocated in the same iteration. As a result, we can treat the body of the loop as an independent procedure and check within the procedure. Many existing approaches on leaks [23, 9, 12] also only track leaks in one iteration, though the techniques are somewhat different.

Fig. 2(b) shows the extract procedure from the body of the loop in our running example. Any edge connecting to the loop on the original CFG is transfered as an edge from the entry. All statements that breaks out of the loop, such as `break`, as well as all statements that start a new iteration, such as `continue` or the last statement in the loop body, are considered as control transfer to the exit. The points-to graph for each CFG node is reused from the original CFG.

It is important to ensure safety when we treat the body of the loop as an independent procedure. Given a node labelled as `Alloc(`m`)` in the extracted procedure, we check whether any memory chunk allocated by m within an iteration would escape this iteration. If so, all memory chunks must be used within the iteration, and thus it is safe to treat the loop body as an independent procedure for m. The concrete algorithm is shown in Algorithm 1.

Algorithm 1 Test whether the leaks of allocation m can be checked in the extracted loop procedure

```
if Any node outside the loop is labelled as Alloc(m) then
  return False
Perform liveness analysis on the original procedure
n_e ← the exit node in the extracted CFG
for each predecessor n of n_e do
  n' ← the corresponding node of n in the original CFG
  g ← the points-to graph at n'
  for each variable v that can reach m on g do
    if v is live after n' then
      return False
return True
```

III. Approach Details

As described in the previous section, our approach consists of the following stages.

1) Perform pointer analysis and build the mapping between SSA variables and variables in the original program.
2) Build method summaries, and identify three types of procedures for each allocation.
3) Within each procedure that needs to be analyzed, check and fix leaks using set-based edge conditions.

In this section we describe the details of the three stages.

A. Pointer Analysis

In this stage we perform the pointer analysis, as well as all enhancements described in Section II-C1. The results of this step is an SSA-based points-to graph for each procedure, as well as converted points-to graphs of the original program for each CFG node.

We assume each node on a points-to graph can be mapped to a set of code elements. The code elements include global variables, local variables, and allocation statements. An allocation statement stands for all memory chunks that may be allocated by this statement. The edges on the graph are labelled with the field names, and the name is "*" for pointer dereference.

It is also useful to have the inverse mapping: given an expression e, we would like to find all possible nodes this expressions could evaluate to. The follows show the main components of defining such a function $nodes$. In the definition, function $var\_to\_node(v)$ finds a node for a variable v. Function $succ(n, f)$ return a set of successors of node n where the edges are labelled with f. If f is omitted, it returns all successors of n.

$$
\begin{aligned}
nodes(var) &= \{var\_to\_node(var)\} \\
nodes(e.f) &= \bigcup_{n \in nodes(e)} \{succ(n, f)\} \\
nodes(*e) &= \bigcup_{n \in nodes(e)} \{succ(n, *)\} \\
nodes(e_1[e_2]) &= \bigcup_{n \in nodes(e_1)} \{succ(n)\}
\end{aligned}
$$

B. Procedure Identification

In this stage, we build summaries for procedures and identify their types with respect to heap memory usage. We also identify and annotate statements with respect to heap memory usage at this stage.

A summary for procedure p consists of four sets of nodes on the points-to graph of the procedure. The first, $SAlloc_p$, is a complete set of nodes that are possibly allocated in p. The second, $SUse_p$ is a complete set of nodes that are possibly read in p. The third, $SDealloc_p$ is a complete set of nodes that are possibly deallocated in p. The fourth, $SEscape_p$ is a complete set of nodes that are possibly escaped (i.e., passed to its caller) from the current procedure.

We build the first three sets in two steps. First, we perform intra-procedural flow-insensitive analysis to compute the sets locally for each procedure. Second, we perform inter-procedural analysis by updating the sets along the call graph. In the intra-procedural analysis, $SAlloc_p$ is constructed by adding all nodes on the points-to graph of p that correspond to allocation statements in procedure p. Set $SUse_p$ is constructed by a union of $nodes(exp)$ for each exp where exp is either a top-level expression used in procedure p or a sub expression of a top-level expression. Set $SDealloc_p$ is constructed by a union of $nodes(exp)$ for each `free(`exp`)` statement in procedure p.

In the second step, for each procedure on call graph, we update its three sets by merging them with sets in its successor nodes by union, as follows.

$$X_p := (\bigcup_{q \in succ(p)} X_q) \cup X_p,$$
where X is *SAlloc*, *SUse*, or *SDealloc*.

We update all summaries iteratively until we reach a fixed point.

It is easy to reason the termination and convergence of the algorithm using the monotone framework [24]. Sets of nodes with the union operator form a lattice of finite height. The transfer functions for the second step are just the id function, and thus are monotone.

After we have built the first three sets, we proceed with $SEscape_p$. We check each allocation $a \in SAlloc_p$ to see whether it should be added to $SEscape_p$. For any procedure q who is a predecessor of p on the call graph, i.e., q calls p, we check whether a is in the points-to graph of q. If so, a must have been escaped from p and thus we add a to $SEscape_p$. However, if a has only one predecessor which is a global variable in the points-to graph of q, we first check the use of a as described in Section II-C3. If a is not used in any caller, we do not added it to $SEscape_p$.

Given the procedure summaries, it is easy to identify the procedure types. If there is a node m where $m \in SAlloc_p \cap SEscape_p$, we mark p as `Alloc(m)`. If there is a node m that corresponds to allocation statement and $m \in SUse_p$, we mark p as `Use(m)`. Similarly, if there is a node m that corresponds to allocation statement and $m \in SDealloc_p$, we mark p as `Dealloc(m)`.

During the summary building, we also identify the statements related to the use or deallocation of heap memory. When we add a node m to $SUse_p$ during the analysis of an expression e, we label the CFG node that containing e as `Use(`m`)` if m corresponds to an allocation statement. Similarly, we label the corresponding CFG node as `Dealloc(`m`)` when we add a heap node m to $SDealloc_p$ during the local analysis.

C. Leak Detection and Fix

In this stage, we perform intra-procedural analysis to detect and fix leaks. For each procedure p, we first determine what allocations should be checked in p. An allocation m should be checked in p if (1) a statement in p is labelled as `Alloc(`m`)` or a successor of p on the call graph is labelled as `Alloc(`m`)`, and (2) p is not labelled as `Alloc(`m`)`. We denote the set of allocations that should be checked in procedure p as M_p.

Next, we extract loops as independent procedures as described in Section II-C. For each extracted loop procedure l, we identify the set of allocations that can be fixed in l, denoted as M_l. Then we update M_p by subtracting M_l, i.e., $M_p := M_p - M_l$.

After this step, we have a set of procedures, either original or extracted, and sets of allocations that should be checked in each procedure. Next we perform a series of analyses to fix leaks according to set-based edge conditions. Basically, we shall perform the following analyses:

1) A forward dataflow analysis to find what deallocations may have been reached before or at each CFG node (2nd condition).
2) A backward dataflow analysis to find what deallocations and uses of allocations may be reached after or at each CFG node (3rd condition).
3) A traversal on the edges to identify the variables that only points to the allocations that can be deallocated at each CFG edge (1st condition).
4) A forward, greedy algorithm that aims to select the earliest points to insert deallocations.

In the following we should illustrate the four analyses one by one.

Analysis 1. This is a forward dataflow analysis consists of the following components.

Data at each CFG node n: Df_n, a set of allocations that may have been deallocated before or at the current CFG node
Meet operator: set union
Transfer function at node n: $f1_n(Df) =$ $\begin{cases} Df \cup \{m_1, \ldots, m_k\} & n \text{ labelled as } \texttt{Dealloc(}m_1,\ldots,m_k\texttt{)} \\ Df & \text{otherwise} \end{cases}$

It is easy to reason that these components conform to the monotone framework [24]: sets with union form a lattice, the number of allocations is finite, and the transfer functions are in the standard GEN-KILL form.

Analysis 2. This is a backward dataflow analysis consists of the following components.

Data at each node n: (Db_n, U_d). Db_n is a set of allocations that may be deallocated after or at the current CFG node, and U_d is a set of allocations that may be referenced after or at the current CFG node.
Meet operator: set union of each component
Transfer function at node n:
$f2_n((Db, U)) = (f1_n(Db), f2'_n(U))$,
where $f2'_n(U) =$
$\begin{cases} U \cup \{m_1, \ldots, m_k\} & n \text{ labelled as } \texttt{Use}(m_1, \ldots, m_k) \\ U & \text{otherwise} \end{cases}$

It is also easy to reason that these components conform to the monotone framework: products of lattices are still lattices, and products of monotone functions are still monotone.

Analysis 3. After the first two analyses, the allocations that we can free at each edge e is $A_e = M_p - Df_{e.from} - Db_{e.to} - U_{e.to}$. We use $e.from$ to denote head of e and use $e.to$ to denote the tail of e. Now the task is to find expressions exp where $nodes(exp) \subseteq A_e$ at the points-to graph of $e.from$. To achieve this, we first locate all such nodes n, where $succ(n) \subseteq A_e$. Next we locate a path from any local/global variable to n through depth search in the reverse direction. Finally, we convert the path to an expression exp. Since the points-to graph is unification-based, we know that $nodes(exp) \subseteq A_e$. We find all such expressions and denote the result set of expressions at edge e as Exp_e.

Analysis 4. After Analysis 3, we have a set of expressions that we can deallocate at each edge. However, inserting a deallocation at any edge will also add new dealloc labels to the node, making some other deallocations unable to insert. As a result, we need an algorithm that inserts all deallocations in a systematic way.

The algorithm is shown in Algorithm 2. In the algorithm, function $outgoing/incoming$ returns the outgoing/incoming edges of a node, and function sub returns all sub expressions of an expression. This algorithm approaches in a way similar to a forward dataflow analysis, though the data transfer happens at edges rather than nodes. At each edge, we store two elements: $ToInsert_e$ is a sequence of expressions that should be inserted as deallocations at this edge, and $Freed_e$ is a set allocations that have been freed by this node or previous nodes. The algorithm traverses the control graph forwardly. At each edge, we greedily select the expressions that deallocates most allocations, and store in $Freed_e$ the allocations this expression frees. We need to be careful that the expression only deallocates the allocations that have not been deallocated, and any part of the expression has not been deallocated. The edges are updated iteratively until we reach a fixed point.

This algorithm always terminates, as in every iteration we decrease $Freed_e$. This algorithm may not always converge to a globally optimal result, as we select only the local optimal expression at each edge. However, this algorithm performs well in practice as it is rare that we have to insert a large number of deallocations in one procedure, and locally optimal results leads to globally optimal results in most cases.

Algorithm 2 Select a set of deallocations at each edge.

$ToVisit \leftarrow outgoing(entry)$
for each $e \leftarrow ToVisit$ **do**
  $ToVisit \leftarrow ToVisit - \{e\}$
  $OldFreed_e \leftarrow Freed_e$
  $Freed_e \leftarrow \bigcup_{n \in incoming(e.from)} Freed_e$
  $ToInsert_e \leftarrow \{\}$
  while $\exists exp \in Exp_e : nodes(exp) \subseteq (A_e - Freed_e)$ **do**
    $U \leftarrow \{exp \mid nodes(exp) \subseteq (A_e - Freed_e)\}$
    $U' \leftarrow \{x \in U \mid \forall s \in sub(x) : \neg(nodes(s) \subseteq Freed_e)$
    select $exp \in U'$ where $|nodes(exp)|$ is the largest
    $ToInsert_e$.append($\{exp\}$)
    $Freed_e \leftarrow Freed_e \cup nodes(exp)$
  if $OldFreed_e \neq Freed_e$ **then**
    $ToVisit \leftarrow ToVisit - outgoing(e.to)$

Finally, we need to map the expressions in $ToInsert_e$ at each edge e back to the insertions of deallocations in code. This requires some engineering efforts to deal with different cases. Sometimes we may need to insert a missing `else` branch or delicately choose an insertion point. Nevertheless, it is always possible to map an insertion at the edge on the CFG graph back into the code.

IV. Implementation

We have implemented our approach as an open source tool, *LeakFix*[1]. Our implementation uses LLVM, and the pointer analysis algorithm is DSA [16]. DSA is an inter-procedural flow-insensitive SSA-based algorithm, and we chose DSA because it is the most stable implementation that is available on the newest version of LLVM.

The points-to graphs generated by DSA contain traceability information that maps nodes to global variables. However, the traceability information from nodes to local variables and heap objects are not included in the graphs. To use the graphs in our algorithms, we run an additional pass to recover the information from the code. First, locally at each procedure, we check all assignment statements to recover the local traceability information for local variables and allocation statements. Second, inter-procedurally on the call graph, we merge the local information at each procedure by the parameters and return values passed between procedures.

We integrate the implementation into an LLVM-based compiler and a linker, namely *Clang* and *GNU gold linker with LLVM plugin*, respectively. In the compiling phase, we use LLVM frontend *Clang* to compile each source code file into an LLVM bitcode file, and record the traceability from source code to its SSA form in bitcode files. In the linking phase, while we use the GNU gold linker with LLVM plugin to link all generated bitcode files together, we read the trace files and perform the three analysis steps as described in Section III.

[1]available at http://sei.pku.edu.cn/%7gaoqing11/leakfix

V. Evaluation

A. Research Questions

Our evaluation aims to answer the following research questions:

RQ1: How effective is our tool in fixing real-world memory leaks?

RQ2: What are the execution time of our tool for memory-leak fixing?

B. Evaluation Setup

We evaluated *LeakFix* on SPEC2000, which is widely used as evaluation benchmarks by existing memory-leak detection papers [22, 9, 10, 12]. We chose SPEC2000 because we wanted to compare our results with these memory-leak detection approaches. All our experiments were executed on Ubuntu 13.04 virtual machine with 3GB memory, and the host is running a 2.66GHz Intel Core5 processor with 8GB memory on Windows 7.

Table I shows information of SPEC2000 programs sorted by program size, taken from an existing paper [9]. The second column shows program size, the third column shows the number of functions, and the last column shows the number of allocation statements in each program.

TABLE I
PROGRAM INFORMATION

Program	Size (Kloc)	#Func	#Allocation
art	1.3	44	11
equake	1.5	45	29
mcf	1.9	44	3
bzip2	4.6	92	10
gzip	7.8	128	5
parser	10.9	342	1
ammp	13.3	197	37
vpr	17.0	290	2
crafty	18.9	127	12
twolf	19.7	209	2
mesa	49.7	1124	67
vortex	52.7	941	8
perlbmk	58.2	1094	4
gap	59.5	872	2
gcc	205.8	2271	53

We count the number of fixes we inserted, and compare them to the number of leaks detected by existing memory-leak detection tools. Note that we may insert more than one fix for one leak as the memory may be leaked from different paths. We also manually check if there are useless fixes. During the manual check, we try to identify any of the two cases where a fix can be useless: (1) the deallocated expression is always a null pointer, and (2) the inserted deallocation is on a dead path. These conditions usually can be easily falsified by seeking for counter-examples.

C. Effectiveness of LeakFix

Table II shows the reported leaks of existing memory-leak detection tools (i.e., LC [22], Fastcheck [9], SPARROW [10], and SABER [12]). The numbers outside parentheses correspond to detected real leaks, while the numbers inside parentheses correspond to false positives. All the numbers are taken from the corresponding papers.

Table III shows the fixed leaks in our tool and the maximum detected memory leaks among memory-leak detection tools. Because some tools are not publicly available, we only use the maximum numbers reported in the corresponding papers. The second column shows the numbers of allocations for which $LeakFix$ generates at least one fix. The third column shows the maximum numbers of detected memory leaks among the detection tools. The fourth column presents the percentages of leaks that our approach provides fixes, the fifth column shows the number of fixes our approach inserted, and the last column shows the number of useless fixes.

TABLE II
LEAKS REPORTED BY DETECTION TOOLS

Program	LC	Fastcheck	SPARROW	SABER
art	1(0)	1(0)	1(0)	1(0)
equake	0(0)	0(0)	0(0)	0(0)
mcf	0(0)	0(0)	0(0)	0(0)
bzip2	1(1)	0(0)	1(0)	1(0)
gzip	1(2)	0(0)	1(4)	1(0)
parser	0(0)	0(0)	0(0)	0(0)
ammp	20(4)	20(0)	20(0)	20(0)
vpr	0(0)	0(1)	0(9)	0(3)
crafty	0(0)	0(0)	0(0)	0(0)
twolf	0(0)	2(0)	5(0)	5(0)
mesa	2(0)	0(2)	9(0)	7(4)
vortex	0(26)	0(0)	0(1)	0(4)
perlbmk	1(0)	1(3)	N/A[1]	8(4)
gap	0(1)	0(0)	0(0)	0(0)
gcc	N/A[1]	35(2)	44(1)	40(5)
total	26(34)	59(8)	81(15)	83(20)

[1]Data is not available in the corresponding paper.

TABLE III
FIXED LEAKS AMONG MAXIMUM DETECTED LEAKS

Program	#Fixed	#Maximum Detected	Percentage(%)	#Fixes	#Useless Fixes
art	0	1	0	0	0
equake	0	0	N/A	0	0
mcf	0	0	N/A	0	0
bzip2	1	1	100	1	0
gzip	1	1	100	1	0
parser	0	0	N/A	0	0
ammp	20	20	100	36	0
vpr	0	0	N/A	0	0
crafty	0	0	N/A	0	0
twolf	0	5	0	0	0
mesa	0	9	0	0	0
vortex	0	0	N/A	0	0
perlbmk	1	8	13	1	0
gap	0	0	N/A	0	0
gcc	2	44	5	2	0
total	25	89	28	41	0

From Table III, we make the following observations:

First, *LeakFix* can successfully generate fixes for a substantial number of allocations with leaks in real-world programs. In total, *LeakFix* generates 41 fixes on 25 leaks, accounting for 28% leaks among the maximum detected leaks. We find that there is no useless fix, i.e., there is no dead code where leaks occur.

Second, in some programs, *LeakFix* is able to generate fixes for all the detected leaks, while in programs like *gcc*, *LeakFix* did not generate many fixes.

Third, *LeakFix* can generate more than one fixes for a `malloc` node. On average, we generate 1.6 fixes for one leak.

We further look into the fixed leaks to understand what kind of leaks are common in practice and what kind of leaks we can fix. Many fixed bugs are in the following code pattern:

```
1  p = malloc();
```

```
2  if (...){
3    ...
4    return;//leak here
5  }
6  ...
7  free(p);
```

This indicates that conditional leaks are common in real-world programs and our approach is effective in fixing these leaks.

We also look into the unfixed leaks to identify why our approach is not able to fix these leaks. We found two main reasons. First, many leaks in gcc are in following code pattern:

```
1  char* p=".."
2  if (..){
3    p=malloc();
4    ...
5  }
6  use(p);//leak here
```

To fix the leak, we have to insert a conditional deallocation as p does not always point to a heap object, which is not considered in our approach. Second, many leaks in the programs are not fixed because of the flow-insensitivity of the pointer analysis algorithm we used. If we replace the pointer analysis with a flow-sensitive one, we should be able to fix these leaks.

D. Execution Time of LeakFix

Table IV shows the execution time of *LeakFix*. Since our approach is built into a compiling and linking process, we report the time used by both the original compiler and the linker in the second column. The third column shows pointer analysis time, and the fourth column shows the time spent in fixing. The fifth column shows the total time spent in the compiler, the linker and *LeakFix*. The last column shows the percentage of the time spent by *LeakFix* among the total time.

TABLE IV
TIME CONSUMPTION

Program	Compiling and Linking (sec)	LeakFix (sec)		Total (sec)	Percentage(%)
		Pointer Analysis	Fix Analysis		
art	0.20	0.02	0.01	0.23	13.0
equake	0.21	0.01	0.02	0.24	12.5
mcf	1.19	0.02	0.01	1.22	2.5
bzip2	0.36	0.03	0.02	0.41	12.2
gzip	1.31	0.04	0.04	1.39	5.8
parser	1.68	0.18	0.07	1.93	13.0
ammp	2.98	0.12	0.37	3.47	14.1
vpr	2.51	0.20	0.31	3.02	16.9
crafty	3.53	0.16	0.23	3.92	9.9
twolf	6.22	0.27	0.20	6.69	7.0
mesa	9.36	5.36	5.97	20.69	54.8
vortex	9.00	0.94	0.83	10.77	16.4
perlbmk	9.50	18.20	39.20	66.90	85.8
gap	6.03	7.36	22.36	35.75	83.1
gcc	10.99	31.76	95.81	142.99	89.2

From Table IV, we make the following observations.

First, the overall time is acceptable. The smallest program, *art*, is 1.3kloc and costs 0.23s. The largest program, *gcc* with about 206kloc, costs about 143s.

Second, the percentage of *LeakFix* time lies in 5%-20% in most cases. However, in large programs such as *gap* and *gcc* the percentage is as high as 90%. The reason may be that compiling and linking is in linear time, while our algorithm requires traversing the points-to graphs. When program size becomes larger, the call graph and points-to graph may be more complicated. Also, the percentage varies by different programs. This is possibly due to different number of allocations and functions in different programs.

Third, the percentage of pointer analysis time among total *LeakFix* time is around 20%-70%. In the largest three programs ($perlbmk$, gap, and gcc), the ratio is around 30%. Since pointer analysis is not the main bottleneck of performance, we could possibly use more accurate pointer analysis algorithms to enhance the precision of our approach.

In summary, the execution time of *LeakFix* is acceptable, and *LeakFix* is scalable for large applications.

E. Threats to Validity

1) Internal Validity: The main threat to internal validity is the possible faults in the implementation of our approach. To reduce this threat, four of the authors participated the implementation, and the code is cross-reviewed and cross-tested.

2) External Validity: The main threat to external validity lies in the representativeness of the benchmark. To reduce the threat, we chose SPEC2000, which is frequently used for evaluating memory-leak detection approaches. To further reduce the threat, we plan to include more subjects as future work.

VI. DISCUSSION

First, though our approach is able to generate multiple fixes for one leak, it does not guarantee that the generated fixes fully fix this leak. As a result, our approach is best used together with leak detection approaches, where leak detection approaches can be used to detect whether there are unfixed leaky paths for an allocation.

Second, our approach is built upon the result of pointer analysis, so the precision of our approach is largely decided by the pointer analysis. For example, in DSA, all pointer elements in a pointer array are merged together, and therefore we cannot free any individual elements in arrays. However, if we use a more accurate pointer analysis that distinguishes different elements in an array, we can free the elements safely.

Third, our approach handles leaks within one iteration of a loop. As a result, we cannot handle the cases where one loop allocates a set of objects (say, an array of heap objects) and another loop releases them. However, while this code pattern is common in object-oriented languages, it is not observed in our experiment subjects. We suspect this is due to efficiency: it is less efficient to allocate a set of objects and an array of their pointers than to allocate a heap array of object, where the latter is not possible in many object-oriented languages such as Java.

Fourth, to ensure safety and reduce analysis time, our approach is conservative in analyzing library functions. For example, in the statement $str = strcat()$, we do not consider str to be initialized because it increases time cost to analyze each library function. To maintain high speed and increase precision, we could build summaries [2] for library functions.

[2]Please refer to Tang et al. [25] for recent advances in summary building.

VII. Related Work

General bug fixing. Recently there has been a lot of work focusing on bug fixing. Several researchers [1, 2, 3] tried to find automatic means to fix general types of bugs. These approaches typically take a violated correctness condition, modify the code and satisfy the condition. The correctness condition comes from either test cases [2, 3] or assertions such as pre-conditions and post-conditions [1]. However, both forms of correctness conditions used in these approaches are usually inadequate, and rarely cover the full space of correctness. Moreover, it is difficult to specify the correctness condition of fixing memory leaks as test cases or assertions.

Dedicated approaches for specific types of bugs also exist. For example, Jin et al. proposed approaches [26, 27] to automate the whole process of fixing a wide variety of concurrency bugs. However, none of the dedicated approaches within our knowledge target memory leaks in C.

Fixing faults can be bad, and may introduce new bugs. Gu et al. [28] formalize the bad fix problem, and define two dimensions of a fix. The "coverage" dimension measures the extent to which the fix handles all triggering faults correctly, while the "disruption" dimension counts deviations from a program's intended behavior introduced by a fix. In our work, we ensure the correctness of the fix, i.e., we avoid the "disruption" dimension of a bad fix, while try to improve the "coverage" dimension of fixes.

Static approaches to memory leaks. The main line of static approaches is to detect leaks [7, 8, 29, 9, 10, 13, 11, 12, 30]. Most detection approaches report only the locations where the leaked memory is allocated. As in many cases the allocation and the deallocation are in quite distinct places, the developers still need lots of efforts to fix the leak. Even in *Fastcheck* [9], which reports leaky paths as well as the allocation, this information is still not enough for memory-leak fixing. Furthermore, all approaches within our knowledge report false positives, making it difficult for developers to rely on these approaches to automatically fix leaks.

The approaches closest to ours are compile-time deallocation for Java [23, 31, 32]. These approaches insert dedicated deallocation statements into Java bytecode to reduce the number of references needed to be scanned by the garbage collector, so as to enhance runtime performance. However, none of these approaches can be easily migrated to C language due to the following reasons. First, when inserting a deallocation into a C program, we must ensure no double-free is caused, where approaches on Java does not have this problem, and it is also not easy to extend existing approaches to handle double-free. For example, Free-me [23] uses live analysis to find the place for deallocation, and live analysis fundamentally only identifies uses of memory chunks but not whether they are freed. Furthermore, Cherem and Rugina's approach [31, 32] strictly obeys the semantic of Java, reclaiming an object only when all its references are lost. This will lead to inefficient memory usage because an object may be referenced for a long time without being used.

There is also research on automated resource management for Java [33, 34]. CLOSER [33] inserts `close()` method for resources that cannot be managed by garbage collector. FACADE [34] is a framework for transforming the data path of Big Data applications, and significantly reduce cost for runtime memory management. Both approaches are not fully automatic, relying on developers for annotation.

Dynamic approaches to memory leaks. The most widely-used dynamic approach to handling memory leaks is garbage collection [35, 36, 37, 38], which adds runtime overhead to the program. There is also research on dynamic memory-leak detection [39, 40, 41, 42, 43, 44, 45], which provides the location of an allocation which will be leaked later. However, as discussed before, this information alone is still far away from fixing the leak. Approaches for dynamic repair of memory leaks also exist [46, 47], but they fix memory leaks at runtime, rather than creating patches for them.

Some researchers have realized the importance of fixing the leaks and approach the problem from different angles. LEAKPOINT [14] tries to provide information that is useful in locating a fix: Instead of reporting only the allocation, LEAKPOINT also reports the location where the reference to the allocated memory is lost or last used in a dynamic path. However, as fixing a leak requires us to consider all possible paths, this information is not sufficient to automatically generate fixes. Rayside and Mendel [48] use object ownership profiling to find and fix junk, a kind of memory where the memory is still referenced but is never used. However, as the name suggests, their approach only give statistic reports to assist humans to fix the leaks, and is not designed for automatic fixing. Xu et al. [15] propose a three-tier approach that uses varying levels to allow programmers with little program knowledge to find the root cause of memory leak quickly. This work also cannot fix errors automatically. Furthermore, all the above approaches rely on dynamic analysis, where our approach is a static analysis.

VIII. Conclusion

This paper proposes a static approach to automatic memory-leak fixing, while ensuring that fixes are safe. We show that the safety of leak fixes can be formally specified, while an approach inserting only safe fixes can be built upon sound pointer analysis. We also show that different challenging cases can be handled by integrating various techniques: inter-procedural leaks and global variables by specialized procedure summaries, loops by extracting loops as independent procedures, and multiple allocations by set-based edge conditions. The resulted approach fixed in total 29% of leaks in our evaluation.

As our first attempt toward type-specific bug fixing, the result is quite encouraging: the algorithm fixes a substantial number of bugs that cannot be captured by general-purpose bug-fixing approaches. This result indicates that the direction of type-specific bug fixing is promising and worth future investigation.

References

[1] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA 2010*, 2010, pp. 61–72.

[2] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *IEEE Congress on Evolutionary Computation*. IEEE, 2008, pp. 162–168.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE '09*, 2009, pp. 364–374.

[4] Y. Qi, X. Mao, Y. Wen, Z. Dai, and B. Gu, "More efficient automatic repair of large-scale programs using weak recompilation," *Science China Information Sciences*, vol. 55, no. 12, pp. 2785–2799, 2012.

[5] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[6] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *ESEC/FSE*, 2009, pp. 315–324.

[7] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive c and c++ memory leak detector," in *PLDI '03*, 2003, pp. 168–181.

[8] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *ESEC/FSE-13*, 2005, pp. 115–125.

[9] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *PLDI '07*, 2007, pp. 480–491.

[10] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *ISMM '08*, 2008, pp. 131–140.

[11] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *ICSE '10*, 2010, pp. 535–544.

[12] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *ISSTA 2012*, 2012, pp. 254–264.

[13] J. Wang, X.-D. Ma, W. Dong, H.-F. Xu, and W.-W. Liu, "Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis," *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 347–356, 2009.

[14] J. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *ICSE*, 2010, pp. 515–524.

[15] G. Xu, M. D. Bond, F. Qin, and A. Rountev, "Leakchaser: helping programmers narrow down causes of memory leaks," in *PLDI '11*, 2011, pp. 270–282.

[16] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *PLDI*, 2007, pp. 278–289.

[17] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *ESEC/FSE '11*, 2011, pp. 343–353.

[18] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *POPL '09*, 2009, pp. 226–238.

[19] ——, "Flow-sensitive pointer analysis for millions of lines of code," in *CGO '11*, 2011, pp. 289–298.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 25–35.

[21] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On abstraction refinement for program analyses in datalog," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 239–248.

[22] M. Orlovich and R. Rugina, "Memory leak analysis by contradiction," in *SAS'06*, 2006, pp. 405–424.

[23] S. Z. Guyer, K. S. McKinley, and D. Frampton, "Free-me: a static analysis for automatic individual object reclamation," in *PLDI '06*, 2006, pp. 364–375.

[24] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, no. 3, pp. 305–317, 1977.

[25] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei, "Summary-based context-sensitive data-dependence analysis in presence of callbacks," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 83–95.

[26] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI '11*, 2011, pp. 389–400.

[27] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *OSDI'12*, 2012, pp. 221–236.

[28] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *ICSE '10*, 2010, pp. 55–64.

[29] B. Hackett and R. Rugina, "Region-based shape analysis with tracked locations," in *POPL '05*, 2005, pp. 310–323.

[30] Z. Xu, J. Zhang, and Z. Xu, "Melton: a practical and precise memory leak detection tool for c programs," *Front. Comput. Sci.*, vol. 9(1), pp. 34–54, 2015.

[31] S. Cherem and R. Rugina, "Uniqueness inference for compile-time object deallocation," in *ISMM '07*, 2007, pp. 117–128.

[32] ——, "Compile-time deallocation of individual objects," in *ISMM '06*, 2006, pp. 138–149.

[33] I. Dillig, T. Dillig, E. Yahav, and S. Chandra, "The closer: Automating resource management in java," in *Proceedings of the 7th International Symposium on Memory Management*, ser. ISMM '08, 2008, pp. 1–10.

[34] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "Facade: A compiler and runtime for (almost) object-bounded big data applications," in *Proceedings of the*

20th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '15, 2015.

[35] R. Jones and R. Lins, *Garbage collection: algorithms for automatic dynamic memory management*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[36] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software: Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[37] P. Wilson, "Uniprocessor garbage collection techniques," in *Memory Management*. Springer Berlin Heidelberg, 1992, vol. 637, pp. 1–42.

[38] H.-J. Boehm, "Bounding space usage of conservative garbage collectors," in *POPL '02*, 2002, pp. 93–100.

[39] M. D. Bond and K. S. McKinley, "Bell: bit-encoding online memory leak detection," in *ASPLOS-XII*, 2006, pp. 61–72.

[40] W. DePauw and G. Sevitsky, "Visualizing reference patterns for solving memory leaks in java," in *ECOOP '99*, 1999, pp. 116–134.

[41] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," in *ASPLOS-XI*, 2004, pp. 156–164.

[42] M. Jump and K. S. McKinley, "Cork: dynamic memory leak detection for garbage-collected languages," in *POPL '07*, 2007, pp. 31–38.

[43] W. DePauw, D. Lorenz, J. Vlissides, and M. Wegman, "Execution patterns in object-oriented visualization," in *COOTS'98*, 1998, pp. 16–16.

[44] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *ICSE '08*, 2008, pp. 151–160.

[45] N. Mitchell and G. Sevitsky, "Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications," in *ECOOP 2003 C Object-Oriented Programming*, ser. Lecture Notes in Computer Science, L. Cardelli, Ed. Springer Berlin Heidelberg, 2003, vol. 2743, pp. 351–377.

[46] G. Novark, E. D. Berger, and B. G. Zorn, "Exterminator: Automatically correcting memory errors with high probability," *Commun. ACM*, vol. 51, no. 12, pp. 87–95, Dec. 2008.

[47] H. H. Nguyen and M. Rinard, "Detecting and eliminating memory leaks using cyclic memory allocation," in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM '07. New York, NY, USA: ACM, 2007, pp. 15–30.

[48] D. Rayside and L. Mendel, "Object ownership profiling: a technique for finding and fixing memory leaks," in *ASE*, 2007, pp. 194–203.