SAVER: Scalable, Precise, and Safe Memory-Error Repair

Anonymous Author(s)

ABSTRACT

In this paper, we present SAVER, a new memory-error repair technique for C programs. Memory errors such as memory leak, double-free, and use-after-free are highly prevalent and fixing them requires significant effort from developers. Automated program repair techniques hold the promise of reducing this burden but the state-of-the-art is still unsatisfactory. In particular, no existing techniques are able to fix those errors in a scalable, precise, and safe way, all of which are required for a truly practical tool. SAVER aims to address these shortcomings of existing techniques. To this end, we propose an approach based on a novel representation of the program called object flow graph, which summarizes the program's heap-related behavior using static analysis. We show that fixing memory errors can be formulated as a graph labeling problem over this graph and present an efficient algorithm to solve it. We evaluated SAVER in combination with an industrial-strength static bug-finder and show that 75% of the reported errors can be fixed automatically by SAVER for a range of open-source C programs.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Software testing and debugging**.

KEYWORDS

Program Repair, Program Analysis, Debugging

1 INTRODUCTION

Goal. Recent years have seen significant progress in automated tools for static error detection and their deployment in production code [7, 15, 45]. Yet, fixing those errors in practice remains mostly a manual and unscalable process. The longterm goal of our research is to bridge this gap by automating the whole process of finding and fixing common errors at compile-time.

In this paper, we focus on fixing memory errors in C programs such as memory leak, use-after-free, and double-free. We target these errors because they are highly prevalent. For example, more than 4,800 memory leaks have been reported and fixed in Linux kernel [1], far more frequently than other errors such as buffer overflows. Furthermore, fixing memory errors manually is time-consuming and error-prone; even a single error may require developers to spend several days or months until the error gets fixed correctly (e.g. [2]). We aim to reduce this burden by achieving a practical technique for automatically fixing memory errors.

Existing Techniques. Recently, several techniques with similar goals have been proposed in the program repair community (notably, [18, 32, 55]) but they suffer from significant drawbacks. On the one hand, there are techniques that scale to large programs but may produce unsafe patches. For example, FootPatch [55] is a state-of-the-art tool that can fix memory leaks in large code bases but may introduce new errors such as double free as a side-effect (e.g. see Section 2.1). On the other hand are tools that guarantee to generate safe patches but do so at the expense of scalability or repairability. For example, MemFix [32] can safely fix memory leaks, use-after-frees, and double-frees but is applicable only to small programs (<5KLoC). LeakFix [18], another safe fixing tool, is scalable but limited to simple memory-leaks, having a relatively low fix rate (e.g. <15% for GNU utilities [32]). All of these shortcomings of existing techniques make them inappropriate for practical use.

This Work. In this paper, we present SAVER, a scalable, precise, yet safe technique for automatically fixing memory errors. To guarantee safety, SAVER uses a sound verification technique and produces patches only when it is confident that the error gets fixed without violating other safety conditions. To achieve high repairability, SAVER supports various fixing strategies, including conditional deallocation and relocation of pointer dereferences. To this end, we propose a new representation of the program called *object flow graph*, which is a labeled graph summarizing the program's heap-related behavior using a static heap analysis. The key technical novelty is to formulate the problem of fixing memory errors as a labeling problem on the object flow graph and present an efficient algorithm for finding correct labels. For scalability, SAVER applies the aforementioned analysis and verification techniques selectively and locally. The analysis is selective as it uses expensive abstractions (e.g. path-sensitivity) only when doing so benefits. Also, only a fraction of the input program is analyzed by effectively slicing out the program with respect to the target error.

The experimental results show that SAVER is a practical tool, significantly outperforming existing techniques. We implemented SAVER as a stand-alone tool that can be combined with off-the-shelf memory-error detectors (e.g. [10, 17, 22, 26, 28, 54, 57, 61]). In our evaluation, we used Infer [10], a state-of-the-art static analyzer deployed within Facebook [15], which supports memory-leak detection for C programs. For 10 open-source programs (2–320 kLoC), Infer reported 95 real memory leaks with 67 false alarms. SAVER successfully fixed 71 out of the 95 true alarms, achieving a 75% fix rate, and did not attempt to fix false alarms. On the other hand, FootPatch, the existing state-of-the-art, managed to fix 15 out of the 95 memory leaks, leading to a 16% fix rate. In doing so, FootPatch generated 8 unsafe patches introducing new double frees or use-after-frees. FootPatch generated 26 patches for false alarms as well, of which 25 were unsafe. We also evaluated the effectiveness of SAVER for fixing use-after-frees and double frees by conducting case studies with 3 open-source projects in the wild, where SAVER was able to correctly fix 14 out of 34 errors.

Contributions. This paper makes the following contributions:

- We present a new technique for fixing memory errors. The key idea is to construct an object flow graph via static analysis and generates a patch by finding correct labels.
- We present SAVER, a practical and publicly available implementation of the proposed approach.
- We demonstrate the effectiveness of SAVER in realistic settings by comparing it with existing techniques.

```
1  int append_data (Node *node, int *ndata) {
2      if (!(Node *n = malloc(sizeof(Node)))
3          return -1; // failed to be appended
4      n->data = ndata;
5      n->next = node->next; node->next = n;
6      return 0; // successfully appended
7  }
8
9  Node *lx = ... // a linked list
10 Node *ly = ... // a linked list
11 for (Node *node = lx; node != NULL; node = node->next) {
12     int *dptr = malloc(sizeof(int));
13     if (!dptr) return;
14     *dptr = *(node->data);
15 (-) append_data(ly, dptr); // potential memory-leak
16 (+) if ((append_data(ly, dptr)) == -1) free(dptr);
17 }
```

Figure 1: A memory leak error (line 12) and the SAVER-generated patch (line 16).

```
1  struct node *cleanup; // list of objects to be deallocated
2  struct node *first = NULL;
3  for (...) {
4      struct node *new = xmalloc(sizeof(*new));
5      make_cleanup(new); // add new to the cleanup list
6      new->name = ...;
7      ...
8      if (...) {
9          first = new;
10 (+)     tmp = first->name;
11         continue;
12     }
13   /* potential use-after-free: `first->name` */
14 (-)   if (first == NULL || new->name != first->name)
15 (+)   if (first == NULL || new->name != tmp)
16         continue;
17   do_cleanups(); // deallocate all objects in cleanup
18 }
```

Figure 2: A use-after-free error (line 14) and the SAVER-generated patch (lines 10 and 15).

2 OVERVIEW

We illustrate key features of SAVER and how it works.

2.1 Motivating Examples

Example 1. Figure 1 describes a simplified memory-leak error in the open-source program `snort`, which can be detected by the Infer static analyzer [10] with the following error report:

```
Object allocated at line 12 is unreachable at line 15.
```

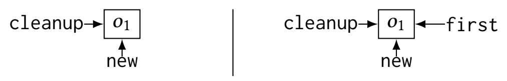
Global variables `lx` and `ly` at lines 9 and 10 are pointers to linked lists. At line 11, the loop iterates over the list `lx`. At each iteration of the loop, a new data object is allocated (line 12) and the data of the current node is copied to the allocated object (line 14). At line 15, the function call, `append_data(ly, dptr)`, stores the allocated object (`dptr`) in the list `ly` as its first element. The function `append_data` returns 0 if the data is stored successfully (line 6). However, it returns -1 when the data object fails to be appended to `ly` (line 3). A memory leak error occurs in the latter case; when `append_data` fails, the object allocated at line 12 becomes unreachable from the environment at the next iteration of the loop since the pointer variable `dptr` gets assigned a newly allocated object.

Given the program `snort` (320 kLoC) and the error report (such as the one produced by Infer), SAVER automatically generates the patch at line 16. It replaces the call to `append_data` at line 15 by the conditional statement given at line 16, correctly deallocating the object (`dptr`) only when `append_data` fails to store the object in the list `ly`. SAVER does so by inferring the program invariant that the allocated object (`dptr`) at each loop iteration becomes unreachable when `append_data` returns `-1` and thus deallocating the object under this condition is always safe.

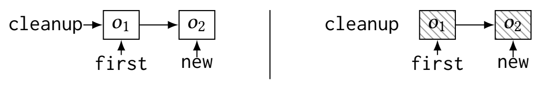
By contrast, FootPatch [55], MemFix [32], and LeakFix [18] fail to correctly fix the error in Figure 1. In particular, FootPatch produces an unsafe patch by simply inserting `free(dptr)` after line 15 without checking the return value of `append_data`, which removes the reported memory leak but introduces a more deadly use-after-free error (when the elements of list `ly` are used later). Safe fixing tools, MemFix and LeakFix are not scalable or robust enough to analyze 320k lines of code but they would fail even without these issues because their use is limited to producing patches without conditionals. Note that the error in Figure 1 is never fixed without introducing a new conditional statement. For example, inserting `free(ndata)` between lines 2 and 3 causes use-after-free in `snort` as `append_data` is called at multiple places where the object pointed to by `ndata` is used even when the return value is -1 (e.g. for printing the error code), which we omitted in Figure 1.

Example 2. Figure 2 shows a tricky use-after-free error [3]. Instead of using primitive deallocators (e.g. `free`), the program uses the `make_cleanup` and `do_cleanups` functions as a special mechanism for memory management. The code maintains a global list called `cleanup`, which holds memory objects to be deallocated. Function `make_cleanup` is used to append an object to the `cleanup` list and `do_cleanups` deallocates all objects in it.

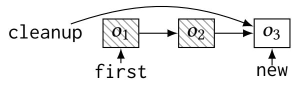
The use-after-free error occurs as follows. At the first iteration of the loop, a new object is allocated at line 4 and its address is stored in the `cleanup` list by calling `make_cleanup(new)` at line 5. Suppose the true branch of the first conditional at line 8 is taken, where a new alias (`first`) for the allocated object is made (line 9). The heap can be depicted as follows:



The left and right diagrams show the heap right after lines 5 and 9, respectively. In each diagram, o_i represents the object allocated at the i-th iteration of the loop. In the second iteration of the loop, a new object o_2 is allocated and appended to list `cleanup`, and `first` still refers to o_1 as depicted in the left diagram below:



Now assume that we take the false branches of both conditionals at lines 8 and 14 and reach the call to `do_cleanups` at line 17. Once `do_cleanups` is called, both o_1 and o_2 are deallocated as depicted with the shaded boxes in the right diagram. In addition, the link from `cleanup` to o_1 is removed. At the third iteration, suppose we take the false branch of the conditional at line 8. Then, we reach the second conditional (line 14) with the following heap:



Since `first` holds a non-null (dangling) pointer, the right-hand side of the disjunction is evaluated, where the dereference `first->name` causes the program to crash as the object o_1 is already deallocated.

SAVER fixes this error by moving the dereference expression (`first->name`) from line 14 to 10, storing its value in a temporary variable (`tmp`), and replaces `first->name` at line 14 by `tmp` as shown at line 15. Note that this patch correctly eliminates the use-after-free error because the pointer `first` is no longer dereferenced at line 15 and dereferencing `first` at line 10 is safe as the object is not yet deallocated. Note also that moving `first->name` from line 14 to 10 does not change the meaning of the program. SAVER ensures this by checking that the values of `tmp` and `first->name` are always equivalent in the second disjunct at line 15 regardless of program executions. Indeed, the SAVER-generated patch in this case is exactly the same as the developer patch.[3]

The SAVER's ability to fix such an error is clearly beyond the reach of the existing techniques. FootPatch, MemFix, and LeakFix attempt to fix memory errors only by inserting or deleting deallocators (without conditionals). However, it is impossible to fix the use-after-free error described above with this simple-minded strategy because there is no way to deallocate an unbounded number of objects with a finite number of primitive deallocators.

2.2 How SAVER Works

Now we overview how SAVER works. Consider the memory leak error in Figure 5a: the object o_1 allocated at line 1 is not freed when the false branch of the conditional is taken. To fix the error, SAVER inserts `if`($\neg C$) `free(p)` before line 7. SAVER generates the patch with the following three steps.

Step 1: Constructing Object Flow Graph. First, SAVER runs a static heap analysis to convert the input program into the object flow graph (OFG) in Figure 5b. A vertex of the OFG represents a heap object at a certain program point and a path condition. For example, vertex $(6, C, o_1)$ denotes the object o_1 available at line 6 when the true branch (C) is taken during program execution and $(6, \neg C, o_1)$ represents the same object o_1 at line 6 when the false branch ($\neg C$) is taken. An edge represents the program's control flow labeled with events that could occur for the destination object. For example, edge $(6, C, o_1) \xrightarrow{\text{free}} (7, C, o_1)$ indicates that the object o_1 is freed when it flows from line 6 to 7 under the condition C and edge $(6, \neg C, o_1) \xrightarrow{\epsilon} (7, \neg C, o_1)$ indicates that no events occur for o_1 under the condition $\neg C$. This way, the OFG summarizes the behavior of all heap-allocated objects (both o_1 and o_2) in the program.

Step 2: Relabeling Object Flow Graph. Next, SAVER attempts to fix the error by relabeling the object flow graph. Note that the memory leak is captured by the red path in the middle of the OFG; concatenating labels over the path produces the string of events:

$$\text{alloc} \cdot \epsilon \cdot \text{use} \cdot \epsilon \cdot \text{unreach}$$

which indicates that the object o_1 is allocated and used along the path but it becomes unreachable without being freed. To eliminate this memory-leak pattern, SAVER replaces the empty label (ϵ) of the edge $(6, \neg C, o_1) \xrightarrow{\epsilon} (7, \neg C, o_1)$ by the free label, producing the following correct usage pattern of heap objects:

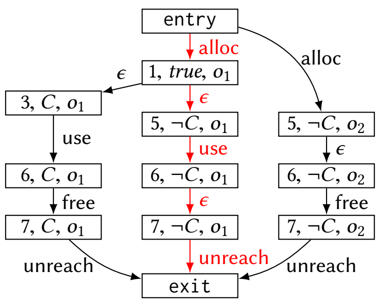
$$\text{alloc} \cdot \epsilon \cdot \text{use} \cdot \text{free} \cdot \text{unreach}$$

Note that it is unsafe to replace the first ϵ by free, as it introduces a use-after-free pattern, alloc · free · use · ϵ · unreach, which is absent in the original OFG. SAVER supports four types of labeling strategies: inserting `free`s, deleting `free`s, and relocating `use`s and `free`s. Figure 6 shows example applications of these strategies for eliminating error patterns. For example, SAVER uses the strategy (relocating use) in Figure 6c to fix the use-after-free error in Figure 2.

Step 3: Generating a Patch. The last step is to generate the patch, `if`($\neg C$) `free(p)`, from the newly labeled edge $(6, \neg C, o_1) \xrightarrow{\text{free}} (7, \neg C, o_1)$. The patch location is between lines 6 and 7. The conditional expression ($\neg C$) of the patch comes from the path condition of the destination object. The pointer expression `p` comes from the points-to information which is supposed to be associated with each vertex but omitted for simplicity in this example.

```
1  p = malloc(1); //o1
2  if (C)
3    q = p;
4  else
5    q = malloc(1); //o2
6  *p = 1;
7  free(q);
```

(a) Example code



(b) Object flow graph

Figure 5: Example program and object flow graph

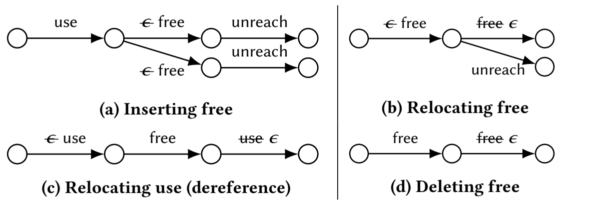


Figure 6: Fixing strategies that SAVER supports

3 APPROACH

This section describes our approach in detail. We first define programs and error reports, which are given as input to SAVER.

Program. The first input to SAVER is a program P represented by a control flow graph $(\mathbb{C}, \hookrightarrow, c_e, c_x)$, where $\mathbb{C}$ denotes the set of program points, $\hookrightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is the set of flow edges, and c_e and c_x are the entry and exit points of the program. A program point $c \in \mathbb{C}$ is associated with a command, denoted $cmd(c)$:

$$\begin{aligned} cmd &\rightarrow x := y \mid x := *y \mid *x := y \mid \mathsf{alloc}(x) \mid \mathsf{free}(x) \mid \mathsf{assume}(b) \\ b &\rightarrow x = n \mid x \neq n \mid x = y \mid x \neq y \end{aligned}$$

A command is either copy ($x := y$), load ($x := *y$), store ($*x := y$), heap allocation ($\mathsf{alloc}(x)$), deallocation ($\mathsf{free}(x)$), or $\mathsf{assume}(b)$ where b denotes a branch condition. Although we consider a simple pointer language without functions for presentation, SAVER works in interprocedural settings and supports the full C language.

Error Report. The second is an error report $R = (c_1, ev_1, c_2, ev_2)$, where c_1 and c_2 are program points and ev_1 and ev_2 are *events*. We call (c_1, ev_1) and (c_2, ev_2) source and sink, respectively. Memory errors are specified with five types of events:

$$ev_1, ev_2 \in Event = \{\mathsf{alloc}, \mathsf{free}, \mathsf{use}, \mathsf{def}, \mathsf{unreach}\}.$$

For example, the following memory leak alarm reported by a static analyzer (e.g., Infer [10])

```
An object allocated at line 1 is unreachable at line 5
```

is represented by $(1, \mathsf{alloc}, 5, \mathsf{unreach})$. Note that ev_1 and ev_2 determine the error type: memory leak, double-free, and use-after-free are represented by $(c_1, \mathsf{alloc}, c_2, \mathsf{unreach})$, $(c_1, \mathsf{free}, c_2, \mathsf{free})$, and $(c_1, \mathsf{free}, c_2, \mathsf{use}/\mathsf{def})$, respectively.

3.1 Step 1: Constructing Object Flow Graph

The first step of SAVER is to construct an object flow graph by statically analyzing the heap-related behavior of the program.

Static Heap Analysis. As object flow graphs play key roles in our approach, we have carefully designed a static heap analysis that can generate precise object flow graphs in practice. The key design decisions are path-sensitivity and heap abstraction. When fixing an error, it is important to isolate the error path from the normal execution paths. To this end, we developed a path-sensitive analysis that uses relational invariants as the path information and merges them selectively for scalability. Also, dynamic data structures such as linked lists are used extensively in real-world C programs. To accurately distinguish the memory objects stored in data structures from those outside, we represent each heap object by a pair of its allocation-site and variables that must point-to the object.

The abstract domain of the analysis is defined as follows:

$$\begin{array}{lcccl} A & \in & \mathbb{D} & = & \mathbb{C} \rightarrow \mathcal{P}(State) \\ s & \in & State & = & PC \times Store \\ \pi & \in & PC & = & \mathcal{P}(Var \times \{=, \neq\} \times (Var + \mathbb{Z})) \\ \sigma & \in & Store & = & Loc \rightarrow \mathcal{P}(Heap) \\ l & \in & Loc & = & Var + Heap \\ h & \in & Heap & = & AllocSite \times \mathcal{P}(Var) \\ a & \in & AllocSite & \subseteq & \mathbb{C} \end{array}$$

A domain element $A \in \mathbb{D}$ is a finite table that maps each program point to a set of reachable states. A state $s = (\pi, \sigma) \in State$ at program point $c \in \mathbb{C}$ consists of a path condition ($\pi \in PC$) and a store ($\sigma \in Store$). The path condition π is a collection of branches that have been taken up to the program point c, where a single branch denotes a relation between two variables (or a variable and a number). The store σ is a map from locations to heap objects, representing the may-point-to information. A location $l \in Loc$ is either a variable or of a heap object. We represent a heap object $h \in Heap$ by its allocation site ($AllocSite$) and a set of must-point-to variables ($\mathcal{P}(Var)$).

The aim of the static analysis is to compute a least fixed point ($\mathsf{lfp}F$) of the semantic function $F \in \mathbb{D} \rightarrow \mathbb{D}$ defined as follows:

$$F(X) = \lambda c. f_c\Big(\bigcup_{c' \hookrightarrow c} X(c')\Big)$$

where $f_c : \mathcal{P}(State) \rightarrow \mathcal{P}(State)$ is the transfer function at c:

$$f_c(S) = \{(f_c^{PC}(\pi), f_c^{Store}(\sigma)) \mid (\pi, \sigma) \in S\}.$$

We update path conditions by $f_c^{PC} : PC \rightarrow PC$ as follows:

$$f_c^{PC}(\pi) = \begin{cases} \pi \cup \{b\} & \cdots \ cmd(c) = \mathsf{assume}(b) \\ \pi \setminus \mathsf{Kill}(\pi, x) & \cdots \ cmd(c) = x := y \\ \pi \setminus \mathsf{Kill}(\pi, x) & \cdots \ cmd(c) = x := *y \\ \pi & \cdots \ \text{otherwise} \end{cases}$$

where $\mathsf{Kill}(\pi, x) = \{(x', \_, x'') \in \pi \mid x = x' \vee x = x''\} \cup \{(x', \_, n) \in \pi \mid x = x'\}$ denotes the relations that are killed. We update stores by f_c^{Store} as follows:

$$f_c^{Store}(\sigma) = \tau_c\Big(\phi_c(\sigma)|_{reach(\phi_c(\sigma))}\Big)$$

which first computes the effect of the command ($\phi_c : Store \rightarrow Store$), projects the stores on the reachable locations (i.e. $reach(\sigma) = \mathsf{lfp}(\lambda R. Var \cup \{l \in \sigma(l') \mid l' \in R\})$), and renames the locations in stores ($\tau_c : Loc \rightarrow Loc$). Stores are updated as follows:

$$\phi_c(\sigma) = \begin{cases} \sigma[x \mapsto \{(c, \{x\})\}] & \cdots \ cmd(c) = \mathsf{alloc}(x) \\ \sigma[x \mapsto \sigma(y)] & \cdots \ cmd(c) = x := y \\ \sigma[x \mapsto \bigcup\{\sigma(l) \mid l \in \sigma(y)\}] & \cdots \ cmd(c) = x := *y \\ \sigma[\mathrm{S}_x \mapsto \sigma(y)][\mathrm{W}_x \overset{weak}{\mapsto} \sigma(y)] & \cdots \ cmd(c) = *x := y \\ \sigma & \cdots \ \text{otherwise} \end{cases}$$

where $\mathrm{S}_x = \{(a, X) \in \sigma(x) \mid x \in X\}$ is a set of strong-updatable locations and $\mathrm{W}_x = \sigma(x) \setminus \mathrm{S}$ is a set of weak-updatable locations, and $\sigma[X \mapsto Y]$ and $\sigma[X \overset{weak}{\mapsto} Y]$ mean strong and weak updates to locations X, respectively. The variables in objects are updated by

$$\tau_c((a, X)) = \begin{cases} \{(a, X \setminus \{x\})\} & \cdots \ cmd(c) = \mathsf{alloc}(x) \text{ or } x := *y \\ \{(a, X \setminus \{x\} \cup \{x \mid y \in X\})\} & \cdots \ cmd(c) = x := y \\ \{(a, X)\} & \cdots \ \text{otherwise} \end{cases}$$

where we assume τ_c does not remove x from X when (a, X) was created by ϕ_c when $cmd(c) = \mathsf{alloc}x$. When the location is a variable, we define $\tau_c(x) = \{x\}$.

Object Flow Graph. Let $A = \mathsf{lfp}F$ be the result of the static heap analysis. From the program $P = (\mathbb{C}, \hookrightarrow, c_e, c_x)$ and the analysis result A, we construct an object flow graph. An object flow graph $G = (V, E, M, \Lambda)$ consists of four components:

$$\begin{array}{ll} V \subseteq \mathbb{C} \times PC \times Heap, & E \subseteq V \times V, \\ M \in V \rightarrow \mathcal{P}(Exp), & \Lambda \in E \rightarrow Event \cup \{\epsilon\} \end{array}$$

where V is the set of vertices, E is the set of edges, M maps vertices to points-to expressions, and Λ denotes the labels of the graph.

We generate the vertices V as follows:

$$V = \{(c, \pi, h) \mid c \in \mathbb{C} \setminus \{c_e, c_x\} \wedge (\pi, \sigma) \in A(c) \wedge h \in \mathrm{range}(\sigma)\} \\ \cup \{(c_e, \mathit{true}, \bot), (c_x, \mathit{true}, \bot)\}$$

A vertex is a triple (c, π, h) of a program point (c), a path condition (π), and a heap object (h). At each program point $c \in \mathbb{C}$, we consider every reachable state (π, σ) and generate a vertex (c, π, h) for each accessible heap object h (i.e. $h \in \mathrm{range}(\sigma)$). Also, V includes two special vertices: the entry $(c_e, \mathit{true}, \bot)$ and exit $(c_x, \mathit{true}, \bot)$ of the object flow graph, where *true* means the empty path condition and $\bot$ a dummy heap object.

Edges E are generated based on the abstract semantics of the heap analysis. That is, E includes $((c_1, \pi_1, h_1), (c_2, \pi_2, h_2))$ if (c_1, π_1, h_1) *generates* (c_2, π_2, h_2) during the analysis:

$$c_1 \hookrightarrow c_2 \wedge \pi_2 = f^{PC}_{c_2}(\pi_1) \wedge h_2 = \tau_{c_2}(h_1).$$

In addition, E includes edges from the entry $(c_e, \mathit{true}, \bot)$ to the vertices without predecessors and from the vertices without successors to the exit $(c_x, \mathit{true}, \bot)$.

The map M associates each vertex (i.e., heap object) with the pointer expressions that may evaluate to the object:

$$M((c, \pi, h)) = \{e \in \mathit{Exp} \mid (\_, \sigma) \in A(c) \wedge h \in [\![e]\!](\sigma)\}.$$

where Exp is the set of finite access paths (i.e., $\mathit{Exp} = \{x, *x \mid x \in \mathit{Var}\}$ for our language) and $[\![e]\!] : \mathit{Store} \to \mathcal{P}(\mathit{Heap})$ is the evaluation function for pointer expressions defined as follows:

$$[\![x]\!](\sigma) = \sigma(x), \quad [\![*x]\!](\sigma) = \bigcup\{\sigma(l) \mid l \in \sigma(x)\}.$$

The map Λ labels the graph with events:

$$\Lambda((c_1, \pi_1, h_1), (c_2, \pi_2, h_2)) = \begin{cases} \mathsf{alloc} & \text{if } \mathit{cmd}(c_2) = \mathsf{alloc}(x) \wedge h_2 = (c_2, \{x\}) \\ \mathsf{free} & \text{if } \mathit{cmd}(c_2) = \mathsf{free}(x) \wedge x \in M((c_2, \pi_2, h_2)) \\ \mathsf{def} & \text{if } \mathit{cmd}(c_2) = *x := y \wedge x \in M((c_2, \pi_2, h_2)) \\ \mathsf{use} & \text{if } \mathit{cmd}(c_2) = x := *y \wedge y \in M((c_2, \pi_2, h_2)) \\ \mathsf{unreach} & \text{if } (c_2, \pi_2, h_2) = (c_x, \mathit{true}, \bot) \\ \epsilon \text{ (empty event)} & \text{otherwise} \end{cases}$$

The edge $((c_1, \pi_1, h_1), (c_2, \pi_2, h_2))$ is labeled as alloc if c_2 is the command that allocates the heap object h_2 (that is, $\mathit{cmd}(c_2)$ is $\mathsf{alloc}(x)$ and h_2 is pointed to by the variable x). We label the edge as free if c_2 may deallocate the heap object h_2. In our implementation, we distinguish may-free and must-free events to more precisely check the safety of patches in the next subsection but here we omit this detail for simplicity. Labels def and use are given when the heap h_2 is defined or read, respectively. Finally, the edge gets the label unreach when the heap h_2 is no longer accessible.

3.2 Step 2: Relabeling Object Flow Graph

The second step of SAVER is to relabel the object flow graph $G = (V, E, M, \Lambda)$ so that the reported error $R = (c_1, ev_1, c_2, ev_2)$ is eliminated with the new labeling.

Error Paths. We first need to convert the error report to a set of error paths on the object flow graph. Let $V^{c1}_{ev_2}$ and $V^{c_2}_{ev_2}$ be the vertices in G that correspond to the source (c_1, ev_1) and sink (c_2, ev_2) of the error report, respectively:

$$V^{c_1}_{ev_1} = \{v \in V \mid v = (c_1, \_, \_) \wedge \exists v'.(v', v) \in E \wedge \Lambda((v', v)) = ev_1\}$$
$$V^{c_2}_{ev_2} = \{v \in V \mid v = (c_2, \_, \_) \wedge \exists v'.(v', v) \in E \wedge \Lambda((v', v)) = ev_2\}$$

Let $\mathrm{Paths}(G)$ be the set of all paths in G. Given a path $p = e_1 \cdots e_n$ (a sequence of edges), we write $\Lambda(p)$ for $\Lambda(e_1) \cdots \Lambda(e_n)$ (the concatenation of labels). The set of error paths, $EP(G, R)$, is as follows:

$$EP(G, R) = \{p \in \mathrm{Paths}(G) \mid (p_0, p_\dashv) \in V^{c_1}_{ev_1} \times V^{c_2}_{ev_2} \wedge \Lambda(p) \in \mathcal{R}^{ev_1}_{ev_2}\}$$

where p_0 and $p_\dashv$ denote the destination points of the first and last edges of p, respectively, and $\mathcal{R}^{ev_1}_{ev_2}$ denotes the set of strings over events that represent the error specified by the source and sink events. For each error type, we define $\mathcal{R}^{ev_1}_{ev_2}$ by regular expressions:

$$\mathcal{R}^{\mathsf{alloc}}_{\mathsf{unreach}} = \mathsf{alloc}(\mathsf{use} \mid \mathsf{def})^*\mathsf{unreach}$$
$$\mathcal{R}^{\mathsf{free}}_{\mathsf{free}} = \_^*\mathsf{free}\_^*\mathsf{free}\_^*$$
$$\mathcal{R}^{\mathsf{free}}_{\mathsf{use}} = \mathcal{R}^{\mathsf{free}}_{\mathsf{def}} = \_^*\mathsf{free}\_^*(\mathsf{use} \mid \mathsf{def})\_^*$$

The meaning is intuitive. $\mathcal{R}^{\mathsf{alloc}}_{\mathsf{unreach}}$ describes memory leak patterns: a memory leak error occurs if an allocated object becomes unreachable without being deallocated. $\mathcal{R}^{\mathsf{free}}_{\mathsf{free}}$ and $\mathcal{R}^{\mathsf{free}}_{\mathsf{use}}$ (or $\mathcal{R}^{\mathsf{free}}_{\mathsf{def}}$) describe double-free and use-after-free patterns, respectively.

Labeling Operators. We relabel the graph with labeling operators. A labeling operator is a function that transforms the label map (Λ) of the object flow graph. Labeling operators have four types:

$$\begin{aligned} \mathsf{free}^+_e(\Lambda) &= \Lambda\{e \mapsto \mathsf{free}\} && \text{if } \Lambda(e) = \epsilon \\ \mathsf{free}^-_e(\Lambda) &= \Lambda\{e \mapsto \epsilon\} && \text{if } \Lambda(e) = \mathsf{free} \\ \mathsf{free}^{\leadsto}_{e_1,e_2}(\Lambda) &= \Lambda\{e_1 \mapsto \epsilon, e_2 \mapsto \mathsf{free}\} && \text{if } \Lambda(e_1) = \mathsf{free} \wedge \Lambda(e_2) = \epsilon \\ \mathsf{use}^{\leadsto}_{e_1,e_2}(\Lambda) &= \Lambda\{e_1 \mapsto \epsilon, e_2 \mapsto \mathsf{use}\} && \text{if } \Lambda(e_1) = \mathsf{use} \wedge \Lambda(e_2) = \epsilon \end{aligned}$$

The free^+_e operator replaces the label of edge e by free if e has the empty label (ϵ). The free^-_e operator removes the free label of e. The $\mathsf{free}^{\leadsto}_{e_1,e_2}$ and $\mathsf{use}^{\leadsto}_{e_1,e_2}$ operators move the free and use labels from e_1 to e_2, respectively. We assume the labeling operators act as an identity function if the conditions (given on the right-hand side) are not met. Given a set $O = \{o_1, \ldots, o_n\}$ of labeling operators, we write $O(\Lambda)$ for the new label map obtained by applying every labeling operator o_i to Λ: i.e., $O(\Lambda) = (o_1 \circ \cdots \circ o_n)(\Lambda)$.

Finding Labeling Operators. Our goal is to find a set O of labeling operators that can remove the error paths in the original object flow graph. Let $G' = (V, E, M, O(\Lambda))$ be the object flow graph whose labels are replaced by O. We need to find O with the following properties:

(1) The reported error does not appear in G': i.e., $EP(G', R) = \emptyset$.
(2) No new errors, which are absent in G, are introduced in G'.
(3) O is semantics-preserving; fixing an error does not cause the program to behave differently in normal execution.

Algorithm. Our algorithm to search for the labeling operators O satisfying the three conditions is given in Algorithm 1. Given the object flow graph G and error report R, the algorithm produces a set C of all possible solutions, where a solution corresponds to a set of labeling operators meeting the conditions. The algorithm uses a worklist W that is a set of pairs; each pair (O, S) consists of a solution candidate O and a search space S. The search space is initialized at line 2, where $\mathsf{InitSearchSpace}(G, R)$ determines the set of search spaces that are appropriate for fixing the error R. At line 4, we pop a work item (O, S) and remove it from the worklist. We relabel the graph G with the current solution O (line 5) and generate the new graph G'. If G' does not contain the error R (line 6) we include O in in the solution set C (line 7). At line 9, we reduce the search space by pruning out labeling operators in S that become unsafe under the new labeling (G'). When the resulting search space S' is empty (line 10), the algorithm moves on to other work items (line 11). Otherwise, it selects an operator o from S' and branches on o by adding new items $(O \cup \{o\}, S'')$ and (O, S'') to the worklist. The algorithm repeats the procedure until the worklist becomes empty or hits a predefined time limit. Below, we describe the two key components: $\mathsf{InitSearchSpace}$ and Safe.

We initialize search spaces based on the error type. Recall that we have four types of labeling operators: free^{+}_{e}, free^{-}_{e}, $\mathsf{free}^{\rightsquigarrow}_{e_1,e_2}$, and $\mathsf{use}^{\rightsquigarrow}_{e_1,e_2}$. When $R = (c_1, ev_1, c_2, ev_2)$ indicates a memory-leak (i.e., $ev_1 = \mathsf{alloc}, ev_2 = \mathsf{unreach}$), we use free^{+}_{e} and $\mathsf{free}^{\rightsquigarrow}_{e_1,e_2}$, meaning that we attempt to fix memory leaks by inserting or relocating deallocation statements. We fix double-free errors by removing deallocation statements (free^{-}_{e}). For use-after-free errors, we attempt to relocate statements that deallocate or use the target object ($\mathsf{free}^{\rightsquigarrow}_{e_1,e_2}$, $\mathsf{use}^{\rightsquigarrow}_{e_1,e_2}$). Formally, $\mathsf{InitSearchSpace}$ is defined as:

$$\mathsf{InitSearchSpace}(G, (c_1, ev_1, c_2, ev_2)) = \begin{cases} \{\mathcal{S}^{ML}_{\mathsf{free}^+}, \mathcal{S}^{ML}_{\mathsf{free}^{\rightsquigarrow}}\} & \text{if } ev_1 = \mathsf{alloc} \land ev_2 = \mathsf{unreach} \\ \{\mathcal{S}^{DF}_{\mathsf{free}^-}\} & \text{if } ev_1 = \mathsf{free} \land ev_2 = \mathsf{free} \\ \{\mathcal{S}^{UAF}_{\mathsf{free}^{\rightsquigarrow}}, \mathcal{S}^{UAF}_{\mathsf{use}^{\rightsquigarrow}}\} & \text{if } ev_1 = \mathsf{free} \land (ev_2 = \mathsf{use} \lor ev_2 = \mathsf{def}) \end{cases}$$

where $\mathcal{S}^{ML}_{\mathsf{free}^+}$, $\mathcal{S}^{ML}_{\mathsf{free}^{\rightsquigarrow}}$, $\mathcal{S}^{DF}_{\mathsf{free}^-}$, $\mathcal{S}^{UAF}_{\mathsf{free}^{\rightsquigarrow}}$, and $\mathcal{S}^{UAF}_{\mathsf{use}^{\rightsquigarrow}}$ are search spaces for memory leak (*ML*), double-free (*DF*), and use-after-free (*UAF*):

$$\begin{aligned}
\mathcal{S}^{ML}_{\mathsf{free}^+} &= \{\mathsf{free}^{+}_{e} \mid e \in E \land \Lambda(e) = \epsilon \land e \in EP(G, R)\} \\
\mathcal{S}^{ML}_{\mathsf{free}^{\rightsquigarrow}} &= \{\mathsf{free}^{\rightsquigarrow}_{e_1,e_2} \mid e_1, e_2 \in E, \Lambda(e_1) = \mathsf{free} \land \Lambda(e_2) = \epsilon \land \\
&\qquad e_1 \notin EP(G, R) \land e_2 \in EP(G, R)\} \\
\mathcal{S}^{DF}_{\mathsf{free}^-} &= \{\mathsf{free}^{-}_{e} \mid e \in E \land \Lambda(e) = \mathsf{free} \land e \in EP(G, R)\} \\
\mathcal{S}^{UAF}_{\mathsf{free}^{\rightsquigarrow}} &= \{\mathsf{free}^{\rightsquigarrow}_{e_1,e_2} \mid e_1, e_2 \in E, \Lambda(e_1) = \mathsf{free} \land \Lambda(e_2) = \epsilon \land \\
&\qquad e_1 \in EP(G, R) \land e_2 \notin EP(G, R) \land e_1 \rightsquigarrow e_2\} \\
\mathcal{S}^{UAF}_{\mathsf{use}^{\rightsquigarrow}} &= \{\mathsf{use}^{\rightsquigarrow}_{e_1,e_2} \mid e_1, e_2 \in E, \Lambda(e_1) = \mathsf{use} \land \Lambda(e_2) = \epsilon \land \\
&\qquad e_1 \in EP(G, R) \land e_2 \notin EP(G, R) \land e_2 \rightsquigarrow e_1\}
\end{aligned}$$

When defining these search spaces, we do not consider operators that are irrelevant to the current error report R. For example, when fixing a memory leak error $(c_1, \mathsf{alloc}, c_2, \mathsf{unreach})$ by inserting a deallocation statement (free^{+}_{e}), the location (e) must be contained in the error paths $EP(G, R)$. Thus, we exclude all the free^{+}_{e} operators from the search space whose location (e) is on the outside of the error paths. We apply similar rules for other error types too.

The $\mathsf{Safe}(G, o)$ predicate checks whether the new labels of G obtained by applying the labeling operator o to G introduces no new errors (safe) or not. We check this safety of a labeling operator based on graph reachability. Let E_{ev} be the edges of G whose labels are equivalent to ev, $\rightsquigarrow$ be the reachability relation for G ($e_1 \rightsquigarrow e_2$ iff e_2 is reachable from e_1), and $E_1 \preceq_{\mathsf{dom}} e$ and $E_1 \preceq_{\mathsf{postdom}} e$ be the facts that the edge set E_1 *collectively* dominate and post-dominate the edge e, respecitvely.

Algorithm 1 Finding Labeling Operators

```
Input: Object flow graph $G$ and error report $R$
Output: A set $C = \{O_1, \ldots, O_n\}$ of sets of labeling operators
 1: $C \leftarrow \emptyset$
 2: $W \leftarrow \{(\emptyset, S) \mid S \in \mathsf{InitSearchSpace}(G, R)\}$
 3: repeat
 4:     $((O, S), W_{rest}) \leftarrow$ pop a work item from $W$
 5:     $G' \leftarrow O(G)$                          ▷ relabel $G$ with $O$
 6:     if $EP(G', R) = \emptyset$ then              ▷ $R$ does not appear in $G'$
 7:         $C \leftarrow C \cup \{O\}$               ▷ add $O$ to $C$ as a solution
 8:     end if
 9:     $S' \leftarrow \{o \in S \mid \mathsf{Safe}(G', o)\}$    ▷ prune out unsafe operators
10:     if $S' = \emptyset$ then
11:         $W \leftarrow W_{rest}$
12:     else
13:         $(o, S'') \leftarrow$ pop a labeling operator from $S'$
14:         $W \leftarrow W_{rest} \cup \{(O \cup \{o\}, S''), (O, S'')\}$
15:     end if
16: until $W = \emptyset$ or timeout
17: return $C$
```

Then, we define $\mathsf{Safe}(G, o)$ for each type of o. When inserting a deallocation statement ($o = \mathsf{free}^{+}_{e}$), we check if doing so does not introduce new double-free or use-after-free errors. This safety condition can be expressed as follows:

$$\forall e' \in E_{\mathsf{free}} \cup \{e\}.e' \not\rightsquigarrow e \land \forall e' \in E_{\mathsf{free}} \cup E_{\mathsf{use}} \cup E_{\mathsf{def}}.e \not\rightsquigarrow e'.$$

When $o = \mathsf{free}^{-}_{e}$, we need to ensure that no memory leak errors are introduced: $E_{\mathsf{free}} \preceq_{\mathsf{dom}} e \lor E_{\mathsf{free}} \preceq_{\mathsf{postdom}} e$ When $o = \mathsf{free}^{\rightsquigarrow}_{e_1,e_2}$, we check the condition:

$$\begin{aligned}
(E_{\mathsf{free}} \cup \{e_2\} \preceq_{\mathsf{dom}} e_1 \lor E_{\mathsf{free}} \cup \{e_2\} \preceq_{\mathsf{postdom}} e_1) \land \\
\forall e \in (E_{\mathsf{free}} \setminus \{e_1\}) \cup \{e_2\}.e_1 \not\rightsquigarrow e_2 \land \\
\forall e \in E_{\mathsf{free}} \cup E_{\mathsf{use}} \cup E_{\mathsf{def}} \setminus \{e_1\}.e_2 \not\rightsquigarrow e_1
\end{aligned}$$

where the first line ensures the absence of memory leak, the second line checks double-free, and the third line guarantees no double-free and use-after-free errors are introduced. (In the above conditions, our implementation distinguishes may-free and must-free events for more precise safety guarantees, which is omitted here for simplicity.) Finally, when $o = \mathsf{use}^{\rightsquigarrow}_{e_1,e_2}$, we need to ensure that no use-after-free errors are introduced ($\forall e \in E_{\mathsf{free}}.e \not\rightsquigarrow e_2$) and relocating use does not cause the program to have different semantics:

$$(\nexists e \in E_{\mathsf{def}}.e_2 \rightsquigarrow e \land e \rightsquigarrow e_1) \land e_2 \preceq_{\mathsf{dom}} e_1$$

We check the semantics-preserving property by ensuring that no definitions exist between e_1 and e_2 and e_2 dominates e_1.

Note that our algorithm produces a set C of solutions such that each solution $O \in C$ consists of labeling operators of the same type. For example, a single O does not contain both $\mathsf{free}^{+}_{e_1}$ and $\mathsf{free}^{-}_{e_2}$. We made this design choice because this single type restriction does not impair the repairability of our algorithm too much but improves its efficiency significantly. Without this restriction, even unsafe operators may become safe as the algorithm progresses. For example, consider the path $v_1 \xrightarrow{\epsilon} v_2 \xrightarrow{\mathsf{free}} v_3$, where the operator $\mathsf{free}^{+}_{(v_1,v_2)}$ is unsafe at the moment but it becomes safe when $\mathsf{free}^{-}_{(v_2,v_3)}$ is allowed later. Thus, we should consider all the possible combinations of the labeling operators in each search space, which can be avoided with the restriction as unsafe operators never become safe.

3.3 Step 3: Generating a Patch

The last step of SAVER is to generate a patch from the set C of labeling operator sets produced by Algorithm 1. When $C = \emptyset$, SAVER fails to fix the error and report that failure. Otherwise, we consider each operator set $O \in C$ and try to convert it to a patch. When the conversion is successful, SAVER returns the resulting patch as its final outcome. If O is not convertible, we move on to the next candidate $O' \in C$. If no operators in C are convertible, SAVER fails to generate a patch.

Next, we describe how to convert a set O of operators into a patch. Basically, we convert each labeling operator in O into a conditional deallocator using the path condition and expression obtained from the object flow graph. We explain the detail depending on the type of operators in O. Suppose $O = \{\mathsf{free}^+_{e_1}, \dots, \mathsf{free}^+_{e_n}\}$. We first partition O based on program points using the equivalence relation:

$$\mathsf{free}^+_{e_i} \sim \mathsf{free}^+_{e_j} \iff \mathit{dest}(e_i) = \mathit{dest}(e_j)$$

where $\mathit{dest}(\_, (c, \_, \_)) = c$ extracts the destination program point from the given edge. Let Q be the set of all equivalence classes associated with corresponding program points:

$$Q = \{(\mathit{dest}(e), O) \mid O \in O/{\sim} \land \mathsf{free}^+_e \in O\}$$

where $O/{\sim}$ denotes the quotient set of O by $\sim$. Now, we convert each partition $(c_i, O_i) \in Q$ into the conditional deallocator:

$$\texttt{if}(\pi_i)\ \texttt{free}(\texttt{exp}_i)$$

and put it at the program point c_i of input the program. The path condition π_i is collected from the object flow graph as follows:

$$\pi_i = \bigvee_{(\_, \pi_j, \_) \in V_i} \pi_j$$

where V_i is the set of the destination objects, i.e., $V_i = \{v \mid \mathsf{free}^+_e \in O_i \land e = (\_, v)\}$. We choose an safe expression exp_i from the map M of the object flow graph such that exp_i may point to the objects in V_i but never points to other objects at c_i:

$$\mathsf{exp} \in \bigcap_{v \in V_i} M(v) \land \mathsf{exp} \notin \bigcup_{V'_i \setminus V_i} M(v)$$

where V'_i is the set of all objects available at c_i whose path conditions are compatible with π_i: $V'_i = \{(c_i, \pi', \_) \in V \mid \pi_i \land \pi' \text{ is satisfiable}\}$. If no expressions satisfying the condition are available, the current partition (c_i, O_i) is not convertible and so is O.

Other cases are simpler because we do not need to synthesize the expression (exp_i). When $O = \{\mathsf{free}^-_{e_1}, \dots, \mathsf{free}^-_{e_n}\}$, we take the same steps described above to obtain π_i for each partition (c_i, O_i) but in this case we insert $\texttt{if}(\neg\pi_i)\ \texttt{free}(\texttt{exp}_i)$ at c_i after deleting the deallocation statement originally located at c_i. Here, we reuse the expression exp_i of the original deallocator.

When $O = \{\mathsf{free}^{\rightsquigarrow}_{e_1, e'_1}, \dots, \mathsf{free}^{\rightsquigarrow}_{e_n, e'_n}\}$, we take similar steps but partition O with the following equivalence relation:

$$\mathsf{free}^{\rightsquigarrow}_{e_i, e'_i} \sim \mathsf{free}^{\rightsquigarrow}_{e_j, e'_j} \iff \mathit{dest}(e_i) = \mathit{dest}(e_j) \land \mathit{dest}(e'_i) = \mathit{dest}(e'_j)$$

which results in the partitions $Q = \{(c_1, c'_1, O_1), \dots, (c_n, c'_n, O_n)\}$. We convert each (c_i, c'_i, O_i) into a patch by removing the original deallocator at c_i, inserting $\texttt{if}(\neg\pi_i)\ \texttt{free}(\texttt{exp}_i)$ at c_i, and inserting $\texttt{if}(\neg\pi'_i)\ \texttt{free}(\texttt{exp}_i)$ at c'_i. Here the path conditions π_i and π'_i are obtained with respect to c_i and c'_i, respectively, in a similar way described above. The expression exp_i comes from the original deallocator at c_i. When $O = \{\mathsf{use}^{\rightsquigarrow}_{e_1, e'_1}, \dots, \mathsf{use}^{\rightsquigarrow}_{e_n, e'_n}\}$, we similarly compute $Q = \{(c_1, c'_1, O_1), \dots, (c_n, c'_n, O_n)\}$. For each (c_i, c'_i, O_i), we obtain the path conditions π_i and π'_i and generate a patch by replacing the load command $x := *y$ at c_i by $\texttt{if}(\pi_i)\ x := t\ \texttt{else}\ x := *y$ with a fresh variable t, and put $\texttt{if}(\pi'_i)\ t := *y$ at c'_i.

3.4 Improving Scalability

The core algorithm described so far is able to generate expressive yet safe patches but is not applicable to large programs. In practice, we use two techniques to improve its scalability.

Slicing. First, we slice the input program with respect to the given error report $R = (c_1, ev_1, c_2, ev_2)$. We use this technique to reduce the cost of applying SAVER in interprocedural settings. Let f_{src} and f_{sink} be the functions that contain the source(c_1) and sink(c_2) program points. We first find the function f_{parent} that is the nearest common caller of f_{src} and f_{sink} on the call-graph of the program. The function f_{parent} becomes the entry point of the SAVER's heap analysis, slicing out all functions in the program that are unreachable from f_{parent}. Often, the resulting sliced program is not small enough to be analyzed. Thus, we use a pre-analysis (a context-sensitive and flow-insensitive points-to analysis) to further slice out the functions that do not have side-effects on the heap objects that have dependences on the objects specified by the error report R. We also slice out large recursive functions as well, which are expensive to analyze but doing so provides little precision benefit. In place of the functions sliced out, we use the result of the pre-analysis to preserve the soundness of the heap analysis. For example, SAVER starts from f_{parent} with the pre-analysis result.

Selective Path-Sensitivity. Second, we apply path-sensitivity of the static heap analysis selectively, only when doing so is likely to improve the final fix rates. To support selective path-sensitivity, we modify the definition of the semantic function F as follows:

$$F(X) = \lambda c.\mathsf{merge}\Big(f_c\Big(\bigcup_{c' \hookrightarrow c} X(c')\Big)\Big)$$

where $\mathsf{merge}(S) = \{\bigsqcup S' \mid S' \in (S/\mathsf{mergable})\}$ partitions the states S and merges each partition into a single state with the equivalence relation $\mathsf{mergable} \subseteq \mathit{State} \times \mathit{State}$. When two states $((\pi_1, \sigma_1), (\pi_2, \sigma_2))$ are joined, the path-condition is generalized (i.e., $\pi_1 \cap \pi_2$) and the points-to information is combined (i.e., $\sigma_1 \sqcup \sigma_2$). We define two states are mergeable if they are equivalent with respect to (1) the set of null-pointing-variables, (2) the set of variables pointing to aliased objects, and (3) the set of pairs of return variables and their values. We designed these three criteria since they capture the key features of program executions related to memory errors and help to analyze normal executions precisely while merging erroneous paths. The first feature describes a set of variables only pointing to null, which captures execution paths along which a pointer is nullified or failed to be allocated. The second feature indicates whether objects are appended to a data-structure such as lists or not. The third feature is to distinguish between states with different return values which are usually used as a signal of memory errors.

4 EVALUATION

In this section, we evaluate SAVER. The main objective is to see how effectively SAVER can fix memory errors in practice and compare it with existing techniques (Sections 4.1). In addition, we discuss the importance of the techniques for improving scalability (Section 4.2).

Implementation. We implemented SAVER as a stand-alone tool in 5,400 lines of OCaml. We used the front-end of Infer [23], which translates C programs into an intermediate representation (called SIL). We used 1-CFA for the pre-analysis in Section 3.4 and full context-sensitivity (except for recursion) for the heap analysis in Section 3.1. For common memory-related standard libraries (e.g. memcpy, strdup), we used hand-crafted models to consider their effects during the heap analysis. We set the timeout of Algorithm 1 to 10 minutes. All experiments were done on a machine with Intel Core i7-7700 processor and 32GB RAM, running Ubuntu-16.04.

4.1 Effectiveness for Fixing Errors

Recall that our original motivation is to combine SAVER with static bug-finders, so that errors are not only found but also fixed automatically. To demonstrate this, we used the memory-leak detector of Infer [10, 23], an open-source tool deployed within Facebook, and compare SAVER and FootPatch [55] in this context. We used Infer-0.9.3, since FootPatch is implemented on top of it.

Setup. We used 10 open-source C programs shown in the first column of Table 1. The three programs (rappel, Swoole, and lxc) came from [55] since they were used for evaluating FootPatch regarding memory leaks. In addition, we collected 7 benchmarks from GitHub that contain at least one memory leak detectable by Infer.

Running Infer on those 10 programs produced a total of 162 alarms. We manually classified the alarms into 95 true and 67 false positives. Then we (automatically) converted each alarm into an error report for SAVER. We included false positives as well in our evaluation, because we assume a use case of repair tools in combination with static bug-finders in an end-to-end way without requiring humans to classify static analysis alarms into true or false.

When running Infer, we enabled the `--headers` option to analyze header files as well. Without this option, Infer skips translating headers into IR, which results in imprecision and increases false alarms. This is why the number(18) of alarms for Swoole in Table 1 is less than that(20) reported in [55].

In Table 1, we compared SAVER with FootPatch only, since other tools, MemFix [32] and LeakFix [18], were not scalable or robust enough to analyze the benchmark programs. MemFix did not terminate for the benchmark programs except for rappel, for which it successfully fixed the reported error. LeakFix also ran on rappel but produced no patches. For other programs, LeakFix produced runtime errors. We ran FootPatch in its global mode [55] to allow it to discover more patch candidates from the entire program. We obtained FootPatch from its public website.[1]

[1]https://github.com/squaresLab/footpatch

For each patch generated by SAVER and FootPatch, we manually checked whether the patch fixed the target error correctly. For true alarms, we say a patch is correct ($\checkmark_T$) if it removes the reported memory-leak alarm completely (e.g. fixing all memory leaks between the source and sink points specified by each alarm) and introduces no new errors. If the generated patch introduces a new error, we counted it as unsafe ($\text{✗}_T, \text{✗}_F$). The remaining case (i.e., the patch is safe but fails to fix the error completely) is counted as incomplete ($\triangle_T$).

Result. Table 1 shows the experimental results. For the 95 true positives, SAVER generated 72 patches. Among them, 71 were correct and fixed errors completely, leading to a 75% fix rate (71/95). One key contributor to this high fix rate was the ability to generate conditional patches. For example, all of the correct patches for snort-2.9.13 involve conditional.

It is notable that SAVER generated no patches for false alarms. This is mainly because SAVER aims to ensure the patch safety; in most cases, SAVER naturally fails to "fix" false alarms because otherwise it needs to find a way to modify a program that is already correct without introducing errors, which is much more challenging than transforming an incorrect program into correct one.

Meanwhile, FootPatch generated 24 patches for true alarms, and managed to repair 15 errors, leading to a 16% fix rate (15/95). The remaining 9 patches were classified into 1 incomplete and 8 unsafe ones (introducing use-after-frees or double-frees). For the 67 false alarms, FootPatch generated 26 patches where 25 were unsafe. Note that most of the false alarm patches are unsafe, implying that "fixing" false alarms correctly is challenging in practice and a practical tool needs to ensure safety to avoid it. In total, FootPatch generated 50 patches for all 162 (true and false) alarms and 33 (66%) of them were unsafe and introduced new errors.

Each of SAVER and FootPatch generated one incomplete patch. SAVER failed to completely fix an error in snort-2.9.13. Consider the following simplified situation:

```
int f(void *p) {
  if (...) return 0; // memory leak
  if (...) return -1; // memory leak
  return 0; // no memory leak }
int g(void *p) {
  x = f(p);
  /*SAVER: if(x==-1) free(p);*/ }
```

where the function `f` can return 0 with and without memory leaks. To fix this, SAVER inserted a conditional patch, `if(x==-1) free(p)`, at the end of `g`, which partially eliminates the memory leak error but some leaks still remain. In this case, it would not be possible to fix the memory leak completely without modifying the body of `f` so that the normal and erroneous paths are distinguished by the associated return values.

FootPatch generated an incomplete patch for recutils-1.8 because of its simple fixing strategy. The `buf_new` function in rectuils-1.8 allocates a base object whose field is also allocated by `buf_new`, both of which cause memory leaks. However, FootPatch inserted a single deallocator for the base object and thus failed to free its field object. By contrast, SAVER identified both leaky objects and generated a correct path by inserting multiple deallocators.

Table 1: Comparison of SAVER and FootPatch on fixing memory leaks detected by Infer. For each program, #T and #F denote the numbers of true and false alarms (i.e. error reports) produced by Infer, respectively. Pre(s) reports the time taken by the pre-analysis of SAVER(pre-analysis is run only once and its result is shared by every error fix). Fix(s) reports the total time taken by each tool in attempting to fix the reported errors. The patch statistics are given in columns G, ✓, △ and ✗, where the subscripts T and F indicate whether the result is for true or false alarms, respectively. G: # of generated patches. ✓: # of successful patches that fixed errors (without introducing new errors). △: # of incomplete patches that are safe but fail to completely fix errors. ✗: # of unsafe patches that introduce new errors.

		Infer		SAVER								FootPatch [55]						
Program	kLoC	#T	#F	Pre(s)	Fix(s)	G_T	✓$_T$	△$_T$	✗$_T$	G_F	✗$_F$	Fix(s)	G_T	✓$_T$	△$_T$	✗$_T$	G_F	✗$_F$
rappel (ad8efd7)	2.1	1	0	0.5	0.3	1	1	0	0	0	0	5.3	1	1	0	0	0	0
flex (d3de49f)	22.3	3	4	5.8	1.7	0	0	0	0	0	0	26.2	0	0	0	0	1	1
WavPack (22977b2)	31.2	1	2	9.6	24.3	0	0	0	0	0	0	37.9	0	0	0	0	2	2
Swoole (a4256e4)	44.5	15	3	32.6	4.0	11	11	0	0	0	0	207.9	9	7	0	2	1	1
p11-kit (ead7a4a)	62.9	33	9	203.3	203.5	24	24	0	0	0	0	227.4	6	5	0	1	2	2
lxc (72cc48f)	63.0	3	5	56.0	4.3	3	3	0	0	0	0	134.6	0	0	0	0	1	1
x264 (d4099dd)	73.2	10	0	56.1	7.3	10	10	0	0	0	0	229.4	2	2	0	0	0	0
recutils-1.8	92.0	10	11	39.6	39.6	8	8	0	0	0	0	349.9	3	2	1	0	0	0
inetutils-1.9.4	116.9	4	5	24.2	2.7	4	4	0	0	0	0	107.9	0	0	0	0	0	0
snort-2.9.13	320.8	15	28	1527.8	112.6	11	10	1	0	0	0	1039.6	3	0	0	3	19	18
Total	828.9	95	67	1804.7	343.5	72	71	1	0	0	0	2366.1	24	15	1	8	26	25

Table 2: Effectiveness for use-after-frees and double-frees

		Use-after-free		Double-free	
Program	kLoC	#Comm.	#Fixed	# Comm.	#Fixed
lxc	63.0	8	4	6	0
p11-kit	62.9	2	1	2	2
grub	247.9	10	5	6	2
Total	373.8	20	10	14	4

Use-After-Free and Double-Free. We also evaluated the effectiveness of SAVER for fixing use-after-frees and double-frees. For this evaluation, we used 34 error reports manually collected from open-source projects. We could not use Infer for this evaluation because it was not effective for finding these kinds of errors—it detected no errors but only produced false alarms for our benchmarks—and we could not find other alternative tools publicly available.

Table 2 shows the benchmarks. We collected them from three open-source projects that contain at least one "use-after-free" and "double-free" keywords in their commit messages in GitHub. lxc and p11-kit are those from the memory-leak benchmarks. We also chose grub from the GNU packages. The number of error commits from each project is given in column **#Comm.** We collected *all* error commits made for use-after-frees and double-frees from the three projects and manually generated 34 error reports by inspecting the commit messages or fixes by developers. For each report, we ran SAVER on the version of the program where the corresponding error commit was made. For some commits, we could not use their exact versions, because they were not always stable releases. In those cases we tried to address the build errors by modifying the source code and Makefiles as minimally as possible.

For the 34 use-after-frees and double-frees, SAVER correctly fixed 14 errors (a 41% fix rate) in total without introducing new errors. SAVER used three strategies for fixing those errors. For use-after-frees, SAVER fixed 10 of 20 errors by moving free or use statements and 4 of 14 double-free errors by deleting frees.

Limitations. Our evaluation also identified one major limitation of SAVER: SAVER often fails to fix errors when they are involved in custom allocators or deallocators. For example, consider the following code snippet describing a double-free in lxc:

```
1  void put_ctx(ctx *ctx) {
2    ... // some side-effect
3    free(ctx); // freed here
4  }
5  void clone_payload(struct s* s){
6    put_ctx(s->init); // second_call
7  }
8  ...
9  init = s->init;
10 put_ctx(init); // first call
11 clone_payload(s); // double-free
```

The function `put_ctx` is a custom deallocator that has a side-effect. It is first used at line 10 to deallocate the object pointed to by `init` and then called again at line 11 in the body of the function `clone_payload`. Because `s->init` and `init` are aliases, a double-free occurs at the second call. However, it is not possible to safely fix this error by removing `free`s, for example, at line 3 because doing so introduces memory leaks. It is also not possible to remove the second call to `put_ctx` because it changes the meaning of the program (because the side-effect is also removed). Therefore, such an error cannot be fixed safely with the current fixing strategies of SAVER. This was the most frequent failure patten (accounting for more than 60%) in Table 2.

4.2 Effectiveness of Techniques for Scalability

We found that the techniques for improving scalability (Section 3.4) are critical components of SAVER. In particular, the slicing technique reduced the cost dramatically. For example, snort-2.9.13 (the largest benchmark) has 7,469 functions but it is sliced to a small program with 14 functions (99.8% reduction) by the technique. Also,

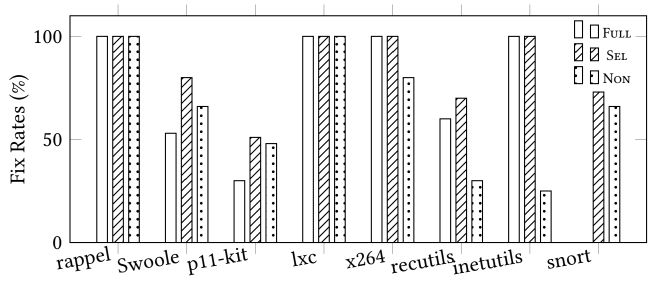


Figure 7: Fix rates with different path-sensitivities

the technique for selective path-sensitivity was essential for achieving high fix rates. Figure 7 compares the fix rates of SAVER (after slicing) with and without our merge heuristic, where Full, Sel, and Non represent SAVER with full path-sensitivity, selective path-sensitivity, and path-insensitivity, respectively. The results show that ours (Sel) performs the best among the three modes, striking a good balance between precision and cost. Overall, Sel achieves higher fix rates than Non and Full, since Non is imprecise and Full often fails to generate patches within the time limit.

4.3 Threats to Validity

In our evaluation, we used 11 open-source programs but they may not be representative or are not enough to objectively evaluate the performance of error repair tools. Our evaluation focused on comparing SAVER with FootPatch regarding memory leaks. However, these two tools are incomparable in general, as FootPatch supports other types of errors such as null dereference and SAVER supports use-after-free and double-free.

5 RELATED WORK

Automated Program Repair. Automated program repair techniques have received an increasing amount of attention from the software engineering community in the last decade [19, 40]. We compare our work with a few major approaches below.

Existing repair techniques can be classified as general-purpose or special-purpose, depending on whether they are designed to fix any kinds or specific kinds of errors. SAVER is a special-purpose technique that focuses on fixing memory errors such as memory leak, use-after-free, and double-free. Other specialized techniques focus on buffer/integer overflows [11, 47], null dereferences [16, 37, 60], concurrency errors [4, 25, 33] and energy bugs [6], among others. Techniques for localizing, mitigating, and fixing memory errors have been studied extensively as well [9, 13, 18, 20, 32, 44, 50–52, 55, 59, 63]. Among them, the directly related to ours are FootPatch [55], MemFix [32], and LeakFix [18]. FootPatch is a scalable technique that fixes pointer safety errors such as memory leaks and null dereferences by applying local reasoning based on separation logic. However, FootPatch does not ensure safety and may introduce new errors as it checks the patch correctness against the given error report only. MemFix uses a sound static analysis to always generate safe patches but it suffers from low scalability. LeakFix is also a safe fixing tool based on data flow analysis but it focuses on simple patches. Furthermore, FootPatch, MemFix, and LeakFix are unable to produce conditional patches, inherently failing to fix diverse errors in practice. To our knowledge, SAVER is the first technique for memory errors that achieves high scalability and repairability without compromising on safety.

General-purpose repair techniques [27, 29, 30, 34, 36, 38, 39, 41, 48, 56, 58] are in principle able to repair all types of errors. Most of these are test-based techniques, which use test cases to verify the correctness of the generated patches (some techniques use program verification [31], metamorphic testing [24], or contracts [42], but a majority of approaches are based on test cases). General repair techniques are classified into generate-and-validate [27, 34, 36, 56] and semantics-based approaches [29, 30, 38, 39, 41]. Generate-and-validate approaches use search algorithms such as genetic programming [56] to explore the space of candidate patches until it finds one that passes all test cases and accelerate the search process using machine learning [36] or fix templates [27]. Semantics-based approaches [29, 30, 38, 39, 41] formulates the patch problem as a constraint satisfaction problem by executing the program symbolically and solves the problem with SMT solvers. Although these approaches are general, they are unlikely more effective than special-purpose techniques for specific errors. For example, Xu et al. recently showed that a specialized technique can be significantly more effective general-purpose techniques for fixing null pointer exceptions [60]. Furthermore, general test-based techniques are inappropriate for fixing memory errors because memory leaks, for example, cannot be fully specified by the input-output test cases. Also, test-based techniques are inherently unsafe as the test cases cannot be a complete specification [35, 43, 49, 62, 64]. Bavishi et al. [8] recently presented a synthesis-based technique for fixing static anlaysis violations, which is general but does not guarantee the patch safety.

Compile-time Memory Management. Compile-time memory management techniques [5, 12, 46, 53] are similar to our work in that they automatically insert memory deallocation statements in the program. For example, Tofte and Talpin [53] and Aiken et al. [5] presented region-based memory management techniques for functional languages to reduce the cost of garbage collection. Shaham et al. [46], Cherem and Rugina [12], and Guyer et al. [21] present static analysis techniques for compile-time memory management for Java. Dillig et al. [14] presented a technique for automating resource management (e.g. network socket) in Java. However, the primary goal of these techniques is to optimize performance (e.g. reducing space consumption) in garbage-collected languages such as Java and ML. In contrast, our focus in this paper is on fixing errors in programs written in unsafe languages.

6 CONCLUSION

Fixing memory errors is tricky because new and more deadly errors are easily introduced. In this paper, we proposed a new technique that can safely fix such errors in a scalable and precise way. To this end, we proposed a new repair algorithm based on object flow graphs. We implemented the technique as a stand-alone tool, SAVER, and evaluated its effectiveness with the Infer static analyzer. For open-source programs, SAVER was proven to be effective in combination with Infer; it was able to safely fix 75% of the memory-leak errors detected by Infer, demonstrating that this combination can be actually useful in practice.

REFERENCES

[1] [n.d.]. https://github.com/torvalds/linux/search?q=fix+memory+leaks&unscoped_q=fix+memory+leaks&type=Commits.

[2] [n.d.]. https://github.com/torvalds/linux/commit/852fef69.

[3] [n.d.]. https://github.com/bminor/binutils-gdb/commit/c91550fc5d8dae5f1140bca649690fa13e5276e9.

[4] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 289–299. https://doi.org/10.1109/ICSE.2017.34

[5] Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better Static Memory Management: Improving Region-based Analysis of Higher-order Languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 174–185. https://doi.org/10.1145/207110.207137

[6] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Transactions on Software Engineering* 44, 5 (May 2018), 470–490. https://doi.org/10.1109/TSE.2017.2689012

[7] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-based Null Safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 740–750. https://doi.org/10.1145/3338906.3338919

[8] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

[9] Michael D. Bond and Kathryn S. McKinley. 2008. Tolerating Memory Leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, USA, 109–126. https://doi.org/10.1145/1449764.1449774

[10] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 459–465.

[11] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: Automatic Integer Error Repair with Proper-type Inference. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 996–1001. http://dl.acm.org/citation.cfm?id=3155562.3155693

[12] Sigmund Cherem and Radu Rugina. 2006. Compile-time Deallocation of Individual Objects. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*. ACM, New York, NY, USA, 138–149. https://doi.org/10.1145/1133956.1133975

[13] James Clause and Alessandro Orso. 2010. LEAKPOINT: Pinpointing the Causes of Memory Leaks. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 515–524. https://doi.org/10.1145/1806799.1806874

[14] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. 2008. The CLOSER: Automating Resource Management in Java. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1375634.1375636

[15] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[16] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 349–358. https://doi.org/10.1109/SANER.2017.7884635

[17] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: Scalable Path-sensitive Memory Leak Detection for Millions of Lines of Code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 72–82. https://doi.org/10.1109/ICSE.2019.00025

[18] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-leak Fixing for C Programs. In *ICSE*.

[19] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (Jan 2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[20] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2019. Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering* (26 Jul 2019). https://doi.org/10.1007/s10664-019-09731-8

[21] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 364–375. https://doi.org/10.1145/1133981.1134024

[22] David L. Heine and Monica S. Lam. 2003. A Practical Flow-sensitive and Context-sensitive C and C++ Memory Leak Detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 168–181. https://doi.org/10.1145/781131.781150

[23] Facebook Inc. 2018. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. Available: https://fbinfer.com.

[24] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. 2017. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software* 126 (2017), 127 – 140. https://doi.org/10.1016/j.jss.2016.04.002

[25] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[26] Yungbum Jung and Kwangkeun Yi. 2008. Practical Memory Leak Detector Based on Parameterized Procedural Summaries. In *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/1375634.1375653

[27] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811. http://dl.acm.org/citation.cfm?id=2486788.2486893

[28] Ted Kremenek. 2008. Finding software bugs with the Clang static analyzer. LLVM Developers' Meeting. Available: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf.

[29] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 376–379. https://doi.org/10.1145/3092703.3098225

[30] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[31] X. B. D. Le, Q. L. Le, D. Lo, and C. Le Goues. 2016. Enhancing Automated Program Repair with Deductive Verification. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 428–432. https://doi.org/10.1109/ICSME.2016.66

[32] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-based Repair of Memory Deallocation Errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 95–106. https://doi.org/10.1145/3236024.3236079

[33] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. 2018. PFix: Fixing Concurrency Bugs Based on Memory Access Patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 589–600. https://doi.org/10.1145/3238147.3238198

[34] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[35] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 702–713. https://doi.org/10.1145/2884781.2884872

[36] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[37] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. [n.d.]. SapFix: Automated End-to-end Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice.*

[38] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 448–458. http://dl.acm.org/citation.cfm?id=2818754.2818811

[39] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[40] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/

3105906

[41] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

[42] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[43] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[44] Derek Rayside and Lucy Mendel. 2007. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 194–203. https://doi.org/10.1145/1321631.1321661

[45] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[46] Ran Shaham, Eran Yahav, Elliot K Kolodner, and Mooly Sagiv. 2003. Establishing local temporal heap safety properties with applications to compile-time memory management. In *International Static Analysis Symposium*. Springer, 483–503.

[47] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 124–135. https://doi.org/10.1109/DSN.2014.25

[48] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/2737924.2737988

[49] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[50] Tatsuya Sonobe, Kohei Suenaga, and Atsushi Igarashi. 2014. Automatic Memory Management Based on Program Transformation Using Ownership. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*. 58–77.

[51] Kohei Suenaga, Ryota Fukuda, and Atsushi Igarashi. 2012. Type-based safe resource deallocation for shared-memory concurrency. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. 1–20.

[52] Kohei Suenaga and Naoki Kobayashi. 2009. Fractional Ownerships for Safe Memory Deallocation. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. 128–143.

[53] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-value lambda-calculus Using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 188–201. https://doi.org/10.1145/174675.177855

[54] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 405–419. https://doi.org/10.1145/3064176.3064211

[55] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[56] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[57] Yichen Xie and Alex Aiken. 2005. Context- and Path-sensitive Memory Leak Detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 115–125. https://doi.org/10.1145/1081706.1081728

[58] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 416–426. https://doi.org/10.1109/ICSE.2017.45

[59] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 270–282. https://doi.org/10.1145/1993498.1993530

[60] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-flow-guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 512–523. https://doi.org/10.1109/ICSE.2019.00063

[61] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal Context Reduction: A Pointer-analysis-based Static Approach for Detecting Use-after-free Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 327–337. https://doi.org/10.1145/3180155.3180178

[62] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 831–841. https://doi.org/10.1145/3106237.3106274

[63] Bin Yu, Cong Tian, Nan Zhang, Zhenhua Duan, and Hongwei Du. 2019. A dynamic approach to detecting, eliminating and fixing memory leaks. *Journal of Combinatorial Optimization* (16 Mar 2019). https://doi.org/10.1007/s10878-019-00398-x

[64] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 913–923. http://dl.acm.org/citation.cfm?id=2818754.2818864