MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C

Junhee Lee
Korea University
Republic of Korea
junhee_lee@korea.ac.kr

Seongjoon Hong*
Korea University
Republic of Korea
seongjoon@korea.ac.kr

Hakjoo Oh†
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

We present MemFix, an automated technique for fixing memory deallocation errors in C programs. MemFix aims to fix memory-leak, double-free, and use-after-free errors, which occur when developers fail to properly deallocate memory objects. MemFix attempts to fix these errors by finding a set of free-statements that correctly deallocate all allocated objects without causing double-frees and use-after-frees. The key insight behind MemFix is that finding such a set of deallocation statements corresponds to solving an exact cover problem derived from a variant of typestate static analysis. We formally present the technique and experimentally show that MemFix is able to fix real errors found in open-source programs. Because MemFix is based on a sound static analysis, the generated patches guarantee to fix the original error without introducing new errors.

CCS CONCEPTS

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

KEYWORDS

Program Repair, Program Analysis, Debugging

ACM Reference Format:
Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236079

1 INTRODUCTION

In programming languages like C and C++, memory-deallocation errors occur when dynamically allocated objects are not deallocated properly. Because these languages entrust memory management to developers, all unused objects must be manually identified and deallocated. Unfortunately, this manual memory-management likely ends up with three types of common errors: a programmer may forget to deallocate an object (memory-leak), may deallocate an object more than once (double-free), or may deallocate an object too early even before it is used (use-after-free).

Table 1: The number of commits that mention memory leak, double-free, use-after-free, and buffer/integer-overflow.

Repo.	#commits	ML	DF	UAF	Total	*-overflow
linux	721,119	3,740	821	1,986	**6,363**	5,092
php	105,613	1,129	148	197	**1,449**	649
git	49,475	350	19	95	**442**	258
openssl	21,009	220	36	12	**264**	61

Memory-leak (ML), double-free (DF), and use-after-free (UAF) are one of the most troubling errors in C programs. For example, they are prevalent in popular open-source software projects (Table 1), even more common than buffer- and integer-overflow. Each of those errors must be carefully examined and fixed because otherwise they become significant sources of security vulnerabilities [5, 53]: memory-leak (CWE-401) may cause a denial of service attack, and double-free (CWE-415) and use-after-free (CWE-416) may allow an unprivileged user to execute arbitrary code. For example, vulnerabilities CVE-2017-10810, CVE-2017-6353, and CVE-2017-8824 recently found in Linux kernel are due to ML, DF, and UAF, respectively.

However, manually fixing memory deallocation errors is time-consuming and error-prone even for experienced developers. Correctly fixing a memory-leak, for example, requires a developer to examine not only the leaky path but also every single path from the source to target, because inserting a deallocation statement in one path may introduce double-free or use-after-free errors in other paths. As a result, it is common in practice for human-written patches only to introduce new errors and the original error gets fixed after multiple rounds of incorrect patches (e.g., see Section 2.1).

In this paper, we present MemFix, an automated technique for fixing memory deallocation errors in C programs. Given a buggy program, MemFix aims to repair the program by finding a set of deallocation statements that correctly deallocate all allocated memory objects without causing double-frees and use-after-frees. We show that finding such a set of deallocation statements is essentially to solve an exact cover problem [13] that can be derived by a static analysis. The static analysis is a variant of type-state analysis [14], which tracks possible deallocation statements for each object. Finding an exact cover is a well-known NP-complete problem, for which

*The first and second authors contributed equally to this work.
†Corresponding author

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236079

we use an off-the-shelf SAT solver. Because MemFix is based on a sound static analysis, the MemFix-generated patches are guaranteed to correctly fix the target error without introducing new errors that are absent in the original program.

The experimental results demonstrate that MemFix is able to fix various errors in open-source C programs. We evaluated MemFix with three different benchmark sets: Juliet Test Suite, GNU Coreutils, and a set of model programs that contain real errors found in popular C repositories. For Juliet Test Suite, MemFix was able to fix all errors related to memory leak, double-free, and use-after-free. For Coreutils, MemFix automatically repaired all deallocation errors in 12 out of 24 programs. For 100 model programs from open-source repositories, MemFix was able to generate patches for 37 cases. We show that these results represent a significant advance over the state-of-the-art by comparing MemFix with LeakFix [15], a recently developed tool for fixing memory leaks in C programs.

Contributions. This paper makes the following contributions:

- We present MemFix, a unified algorithm for fixing memory-leak, double-free, and use-after-free errors in C programs. The key technical novelty is the use of static analysis to derive an exact cover problem whose solution corresponds to a correct patch for memory deallocation errors.
- We provide experimental evaluation of MemFix with three benchmark sets of open-source C programs. We compare its performance with the state-of-the-art tool, LeakFix [15].
- We make the tool and benchmarks publicly available.[1]. In particular, we provide a new benchmark set, comprising 100 test programs (55–664 LoC), which was abstracted from real errors reported in open-source C repositories.

2 OVERVIEW

In this section, we motivate and illustrate MemFix with examples.

2.1 Motivating Example

Figure 1 illustrates how a double-free error found in Linux kernel gets fixed by a developer and MemFix. The original buggy code in Figure 1(a) is eventually fixed in Figure 1(d) through three rounds of patches, taking 10 months in total. Figure 1(e) shows the patch generated by MemFix. We simplified code for presentation.

Original Code. The original code in Figure 1(a) has two double-free errors. At lines 1–5, buffers `in` and `out` are allocated, used, and deallocated. At lines 7 and 13, `in` and `out` are re-allocated with increased size. Double-frees would occur when these buffers fail to get re-allocated. For example, when `malloc` fails and returns `NULL` at line 7, the program control is transferred to the error handling code at line 20, where `free` is called on `out` that is already deallocated at line 4. Similarly, when `malloc` at line 13 fails, `in` is deallocated twice at lines 15 and 21.

First Patch. Figure 1(b) shows the initial patch made by a developer.[2] To prevent double-frees, the developer nullified the buffers at lines 9 and 16. As a result, when `in` or `out` fail to get re-allocated, they are assigned `NULL` so that the subsequent calls to `free` are safe.

This is a correct fix although it is not ideal. It is not ideal because the fix just avoids the errors without eliminating their root causes. That is, the double-free patterns are still present in Figure 1(b): along the path $15 \rightarrow 17 \rightarrow 21$, buffer `in` is deallocated twice. Similarly, along the path $4 \rightarrow 10 \rightarrow 22$, `free` is called on `out` consecutively. To avoid double-frees along those paths, the developer simply nullified the buffers in-between the consecutive `free`s instead of removing the problematic patterns.

Second Patch. Indeed, the developer got confused in understanding the code and thought that a double-free is still present in the code.[3] To "fix" this imaginary bug, (s)he moved `free(out)` at line 4 to line 12, right before the re-allocation of `out`. This is a more desirable place for `free(out)` as it eliminates the double-free pattern for `out`, making the code simpler to understand. However, this change introduced a new memory-leak error. Note that, when buffer `in` fails to get allocated at line 7, `out` is nullified (line 9), so the object that `out` points to is no longer reachable.

Third Patch. This memory leak was reported by another developer and the original developer patched the code once again, as shown in Figure 1(d).[4] To fix the leak, the developer moved `free(out)` at line 12 back to its original place (line 4). Also, the developer nullified `out` at line 6 to prevent the imaginary double-free that (s)he thinks was introduced by the first patch. The resulting code is error-free. However, it became even more redundant and confusable than the first patch in Figure 1(b).

Patch by MemFix. Figure 1(e) shows the fix made by MemFix. To fix the double-free errors in the original code (Figure 1(a)), MemFix eliminated their root causes with two simple changes: it removed `free(in)` from line 15 and moved `free(out)` from line 4 to 12. The idea is to deallocate buffers right before they get re-allocated, which is the simplest patch for the original code. The generated patch is instructive for a developer to better understand how the problem can be fixed. Furthermore, because MemFix is sound and formally ensures that the generated patch is correct, it frees the developer from the error-prone task of manually verifying the correctness of the patch.

2.2 Overview of MemFix

We illustrate the algorithm of MemFix using a simple example in Figure 2. The code in Figure 2(a) has a double-free error, which occurs along the lines $1 \rightarrow 7 \rightarrow 9 \rightarrow 10$. At line 1, an object (denoted o_1) is allocated and stored in pointer `p`. At line 7, both `q` and `p` refer to the same object (o_1) that is later deallocated twice at lines 9 and 10, causing a double-free. Our technique fixes this error by moving `free(p)` at line 10 to line 4, as shown in Figure 2(b).

MemFix works with two key ideas: 1) we use a static analysis that collects patch candidates for each allocated object, and 2) we reduce the problem of finding a correct patch into an exact cover problem over allocated objects. We explain these steps below.

Static Analysis for Collecting Patch Candidates. We first analyze the code to collect patch candidates. The control-flow graph and analysis results at each node are presented in Figure 3. The

[1] http://prl.korea.ac.kr/MemFix
[2] https://github.com/torvalds/linux/commit/ed6590a
[3] https://github.com/torvalds/linux/commit/df3e1ab7
[4] https://github.com/torvalds/linux/commit/852fef69

```
in = malloc(1);
out = malloc(1);
... // use in, out
free(out);
free(in);

in = malloc(2);
if (in == NULL) {

  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);

  goto err;
}
... // use in, out
err:
  free(in);
  free(out);
  return;
```

(a) Original code (double-free)

```
in = malloc(1);
out = malloc(1);
...
free(out);
free(in);

in = malloc(2);
if (in == NULL) {
  out = NULL; // +
  goto err;
}

out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL; // +
  goto err;
}
...
err:
  free(in);
  free(out);
  return;
```

(b) First patch (2007.9) (correct)

```
in = malloc(1);
out = malloc(1);
...
// -
free(in);

in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}
free(out); // +
out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
...
err:
  free(in);
  free(out);
  return;
```

(c) Second patch (2008.6) (memory-leak)

```
in = malloc(1);
out = malloc(1);
...
free(out); // +
free(in);
out = NULL; // +
in = malloc(2);
if (in == NULL) {
  out = NULL;
  goto err;
}
// -
out = malloc(2);
if (out == NULL) {
  free(in);
  in = NULL;
  goto err;
}
...
err:
  free(in);
  free(out);
  return;
```

(d) Third patch (2008.7) (correct)

```
in = malloc(1);
out = malloc(1);
...
// -
free(in);

in = malloc(2);
if (in == NULL) {

  goto err;
}
free(out); // +
out = malloc(2);
if (out == NULL) {
  // -

  goto err;
}
...
err:
  free(in);
  free(out);
  return;
```

(e) Fix by MemFix (correct)

Figure 1: (a) Original buggy code. (b)–(d) A series of patches made by a developer. (e) Fix by our technique.

```
p = malloc(1); // o1
if (...) {
  q = malloc(1); // o2

}
else
   q = p;
... = *q; // use q
free(q);
free(p); // double-free
```

(a) Buggy code

```
p = malloc(1);
if (...) {
  q = malloc(1);
  free(p); // +
}
else
   q = p;
... = *q;
free(q);
// -
```

(b) Fixed code

Figure 2: Example for illustrating our technique.

analysis is a variant of typestate analysis [14], which maintains points-to and patch information for each allocated object as a tuple of the following form[5]:

$$\langle o, must, mustNot, patch, patchNot \rangle$$

where o is an object represented by its allocation-site, *must* is a set of pointers that must point-to o, *mustNot* is a set of pointers that definitely do not point-to o, *patch* is a set of patches that are guaranteed to safely deallocate the object, and *patchNot* is a set of potentially unsafe patches. We call the tuple an *object state*. For example, the analysis computes the following tuple after line 1:

$$\{\langle o_1, \{p\}, \emptyset, \{(1,p)\}, \emptyset \rangle\} \quad (1)$$

At the first line, an object o_1 is allocated and pointer p points-to that object. A patch is a pair (n, e) of a program location (n) and a pointer expression (e), which denotes a deallocation statement, `free(`e`)`, at line n. For example, the safe patch $(1, p)$ in the object state above indicates that we can deallocate the object o_1 by inserting `free(`p`)` after line 1. Right now, *mustNot* and *patchNot* are empty.

At line 3, a new object (o_2) is allocated and the existing object states get updated as follows:

$$\{\langle o_2, \{q\}, \emptyset, \{(3,q)\}, \emptyset \rangle, \langle o_1, \{p\}, \{q\}, \{(1,p),(3,p)\}, \emptyset \rangle\} \quad (2)$$

The first tuple denotes the new object allocated at line 3: the allocation-site is o_2, q points-to o_2, and we can safely deallocate o_2 at line 3 via `free(`q`)`. Note that allocating object o_2 may change the state of object o_1: since q definitely no longer point-to o_1, we add q to the *mustNot* set of o_1. Also, *patch* of o_1 includes $(3, p)$ as well as $(1, p)$ because it is safe to deallocate o_1 at line 1 or 3 via `free(`p`)`.

Next, consider the false branch (line 7), where the analysis maintains the information for object o_1 as follows:

$$\{\langle o_1, \{p, q\}, \emptyset, \{(1,p),(7,p),(7,q)\}, \emptyset \rangle\} \quad (3)$$

Because of the copy statement `q = p`, both `q` and `p` point-to o_1. The object can be deallocated with either `free(`p`)` or `free(`q`)` after the statement (line 7).

[5] In this overview, we simplified the object representation for brevity. See Section 3.2 for the complete representation.

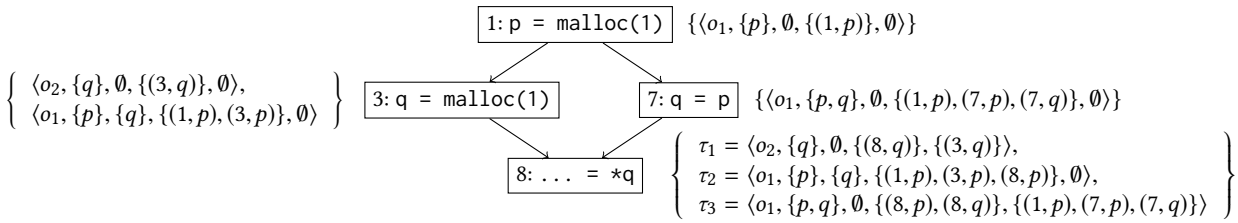

Figure 3: Static analysis for collecting patch candidates

At the join point (before we analyze line 8), the analysis results from both branches, i.e., (2) and (3), are combined:

$$\left\{ \begin{array}{l} \langle o_2, \{q\}, \emptyset, \{(3,q)\}, \emptyset \rangle, \\ \langle o_1, \{p\}, \{q\}, \{(1,p),(3,p)\}, \emptyset \rangle, \\ \langle o_1, \{p,q\}, \emptyset, \{(1,p),(7,p),(7,q)\}, \emptyset \rangle \end{array} \right\} \quad (4)$$

Note that our analysis is disjunctive and maintains object states separately for each different branch unless the resulting states are the same. The second and last tuples denote the states of the object o_1 that follow the true and false branches, respectively.

With the states in (4) as input, the analysis produces the following states as output after line 8:

$$\left\{ \begin{array}{l} \tau_1 = \langle o_2, \{q\}, \emptyset, \{(8,q)\}, \{(3,q)\} \rangle, \\ \tau_2 = \langle o_1, \{p\}, \{q\}, \{(1,p),(3,p),(8,p)\}, \emptyset \rangle, \\ \tau_3 = \langle o_1, \{p,q\}, \emptyset, \{(8,p),(8,q)\}, \{(1,p),(7,p),(7,q)\} \rangle \end{array} \right\} \quad (5)$$

Because q is used at line 8, the existing patches for the objects that can be referenced by q are no longer safe (i.e., potential use-after-free). Thus, we remove the patch candidates from the states and declare them as unsafe. For example, consider the first state in (4), i.e., $\langle o_2, \{q\}, \emptyset, \{(3,q)\}, \emptyset \rangle$, whose object can be referenced by q. We remove $(3,q)$ from *patch* to *patchNot*. Also, we indicate that a new patch $(8,q)$ is safe after line 8, yielding τ_1 in (5). Similarly, τ_3 in (5) is obtained by removing the existing patches and adding new patches ($(8,p)$ and $(8,q)$) since q points-to o_1. Note that, however, the existing patches of the second tuple in (4) remain the same because in this case q definitely does not point-to o_1 and therefore the corresponding object cannot be used.

The analysis finishes with the object states in (5). All deallocation statements in the original code are ignored during analysis.

Finding Correct Patches by Solving Exact Cover Problem. Once we collect all object states as well as their patch candidates, we try to find a set of patches that correctly deallocate all object states (i.e., no memory leaks) while not introducing double-frees and use-after-frees.

We find the correct patches by reducing the problem into an exact cover problem as follows. From the analysis results in (5), we first collect the safe and unsafe patches across all object states:

$$\begin{array}{rcl} \mathit{Safe} & = & \{(1,p),(3,p),(8,p),(8,q)\} \\ \mathit{UnSafe} & = & \{(1,p),(3,q),(7,p),(7,q)\}. \end{array}$$

Candidate patches are those in *Safe* but not in *Unsafe*:

$$\mathit{Cand} = \mathit{Safe} \setminus \mathit{Unsafe} = \{(3,p),(8,p),(8,q)\}.$$

These are possible patches that do not cause use-after-free errors. However, using all of them may cause double-frees. We have to find a subset of the candidate patches that does not introduce double-frees while deallocating all object states, which corresponds to solving an exact cover problem represented by the following incidence matrix:

	τ_1	τ_2	τ_3
$(3,\mathbf{p})$	0	**1**	0
$(8,p)$	0	1	1
$(8,\mathbf{q})$	**1**	0	**1**

The matrix has one row for each element of *Cand* and one column for each state in (5). The entry in row c and column τ is 1 if patch c is a safe patch for state τ (i.e., c is included in *patch* of τ) and 0 otherwise. For example, τ_1 contains $(8,q)$ in *patch*, so the entry in row $(8,q)$ and column τ_1 is 1. Then, we aim to select rows such that each column is contained in exactly one selected row. In the example, $\{(3,p),(8,q)\}$ is the solution, as highlighed above, that covers all states (hence no memory leaks) and each state is covered by at most one patch (no double-frees). The exact cover problem is NP-complete [13]. We solve the problem by encoding it as boolean satisfiability and leveraging an off-the-shelf SAT solver [20]. MemFix is able to fix the bug iff the boolean formula is satisfiable.

Applying the generated patch $\{(3,p),(8,q)\}$ to the original buggy code is easy. We first remove all deallocation statements from the code in Figure 2(a) and then insert `free(p)` after line 3 and `free(q)` after line 8, resulting in the code in Figure 2(b).

3 THE MEMFIX ALGORITHM

Now, we formally present MemFix. Section 3.1 defines a core language. Section 3.2 describes the first step of MemFix, the patch-collecting static analysis. The second step, choosing a set of correct patches by solving an exact cover problem, is described in Section 3.3.

3.1 Language

We formalize MemFix on top of a simple pointer language. Let P be a program to repair. We represent the program by a control flow graph $(\mathbb{C}, \hookrightarrow, c_e, c_x)$, where $\mathbb{C}$ denotes the set of program points, $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flow of the program, $c_1 \hookrightarrow c_2$ meaning that c_1 is a predecessor node of c_2, c_e is the entry node, and c_x is the exit node of the program. The entry and exit nodes are unique. Each program point is associated with a command defined by the following grammar:

$$\mathit{cmd} \rightarrow \mathsf{set}(p,e) \mid \mathsf{alloc}(p) \mid \mathsf{free}(p), \quad p \rightarrow x \mid *x, \quad e \rightarrow p \mid \mathsf{null}$$

A pointer expression (p) is either a variable (x) or its dereference ($*x$). An expression is a pointer expression or null. A command is an assignment (set(p, e)), an allocation statement (alloc(p)), or a deallocation statement (free(p)). alloc(p) creates a new object pointed to by p. free(p) deallocates the object that p points to. Let Var be the finite set of program variables in P. Let $AllocSite \subseteq \mathbb{C}$ be the finite set of allocation-sites, i.e., nodes whose associated commands are allocation statements, in P. We write $cmd(c)$ for the command associated with c. For simplicity, we describe our algorithm with the simple language. Extending the algorithm to support other C features such as procedure calls and structures introduces no new foundational ideas.

3.2 Collecting Patches via Static Analysis

The first step of MemFix is to analyze the program to collect patch candidates. The analysis is a variant of typestate analysis [14], which aims to identify the set of possible patches for each allocated object. We assume the basic knowledge of the abstract interpretation framework [10].

3.2.1 Abstract Domain. Our analysis is flow-sensitive and disjunctive; that is, it computes a set of *object states* for each program point. The abstract domain $\mathbb{D}$ of the analysis is defined as a function from program points to sets of reachable object states: $\mathbb{D} = \mathbb{C} \to \mathcal{P}(\mathbb{S})$. An object state $s \in \mathbb{S}$ describes an abstract object and is represented by a tuple of the form:

$$\langle o, may, must, mustNot, patch, patchNot \rangle$$

where $o \in AllocSite$ is the allocation-site of the object, $may \subseteq AP$ is a set of access paths that may point-to the object, $must \subseteq AP$ is a set of access paths that must point-to the object, $mustNot \subseteq AP$ is a set of access paths that definitely does not point-to the object, $patch \subseteq \mathbb{C} \times AP$ is a set of patches that can safely deallocate the object, and $patchNot \subseteq \mathbb{C} \times AP$ is a set of potentially unsafe patches.

AP denotes the set of pointer access paths that can be generated for the given program. For the language in Section 3.1, AP is equivalent to the set of pointer expressions (p), i.e., $AP = \{x, *x \mid x \in Var\}$, which is finite. When pointer expressions are unbounded, we approximate AP to be finite by limiting the length of the access paths [14].

A $patch$ is a pair $(c, p) \in \mathbb{C} \times AP$, consisting of a program point c and an access path p. A patch (c, p) represents a deallocation statement, free(p), positioned right after the program point c.

Our abstract domain is similar to that of the typestate analysis by Fink et al. [14]. Both analyses represent an abstract object with the must and must-not point-to sets. Key difference, however, is that we further distinguish object states with safe and unsafe patches, as opposed to the typestate of the object used by Fink et al. [14].

During analysis, the following invariants are maintained in any object state: 1) may includes $must$ but excludes $mustNot$, i.e., $may \supseteq must \wedge may \cap mustNot = \emptyset$; and 2) $patch$ and $patchNot$ are disjoint, i.e., $patch \cap patchNot = \emptyset$.

3.2.2 Abstract Semantics. The abstract semantics is defined as the least fixed point $\mathrm{lfp}F \in \mathbb{D}$ of the semantics function $F \in \mathbb{D} \to \mathbb{D}$:

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

where $X \in \mathbb{D}$ and $f_c : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ denotes the abstract semantics of the command associated with the program point c:

$$f_c(S) = \begin{cases} S' & \text{if } cmd(c) = \text{set}(lv, e) \\ S' \cup \{\langle c, \{x\}, \{x\}, \emptyset, \{(c,x)\}, \emptyset \rangle\} & \text{if } cmd(c) = \text{alloc}(x) \\ S & \text{if } cmd(c) = \text{free}(lv) \end{cases}$$

We will define S' shortly. Note that a new object state is created from an allocation statement: the allocation-site is the current program point c, the object is definitely pointed to by x, and the only safe patch available is (c, x). Note also that the analysis ignores deallocation statements, which has the effect of removing them from the program prior to the analysis.

The set S' is defined with two transfer functions τ_c and ϕ_c:

$$S' = \bigsqcup_{s \in S} (\tau_c \circ \phi_c)(s).$$

The transfer functions $\tau_c : \mathbb{S} \to \mathbb{S}$ and $\phi_c : \mathbb{S} \to \mathbb{S}$ update the patch and points-to sets of an object state, respectively. Manipulating the patch information (i.e., τ_c) is new but handling the must and must-not access paths (i.e., ϕ_c) is common in typestate analysis (e.g., [14]). We first define the transfer function τ_c below. Given an object state at the program point c

$$s = \langle o, may, must, mustNot, patch, patchNot \rangle,$$

τ_c updates the patch information as follows:

$$\tau_c(s) = \langle o, may, must, mustNot, patch', patchNot' \rangle$$

$$patch' = \begin{cases} G \setminus patchNot' & o \text{ can be used at } c \\ (patch \cup G) \setminus patchNot' & o \text{ is not used at } c \end{cases}$$

$$patchNot' = \begin{cases} patchNot \cup U \cup patch & o \text{ can be used at } c \\ patchNot \cup U \cup D & o \text{ is not used at } c \end{cases}$$

where the sets G, U, and D are defined below.

- G is the set of patches that are newly generated at c:

  $$G = \{(c, p) \mid p \in must\}.$$

  When an access path p definitely points-to the object, we can use the access path to deallocate the object by inserting free(p) at c. Thus, $patch'$ always includes G. If the object o is not used at c, $patch'$ preserves $patch$ but, when o can be used at c, $patch'$ does not include $patch$ because the existing patches ($patch$) might cause use-after-free errors and no longer guarantee the safety. Finally, we exclude $patchNot'$ from the resulting sets in order to ensure the invariant that $patch'$ and $patchNot'$ are disjoint.
- U is the set of patches that we cannot guarantee the safety:

  $$U = \{(c, p) \mid p \in may \setminus must\}.$$

  These patches may be unsafe because p may point to the object at runtime. Thus, we include them in $patchNot'$.
- D is the set of patches that potentially cause double-frees:

  $$D = patch \cap G$$

  which should be included in $patchNot'$. We detect double-frees by checking whether the generated patch $(c, p) \in G$ already exists in the current safe patches ($patch$). If so, the deallocation free(p) at c can be executed more than once, causing a double-free. When the object is used at c, we also

add the set of patches that may cause use-after-frees (i.e., *patch*). Note that *patch* is a superset of *D*.

Next we define the transfer function ϕ_c that updates the points-to sets. We assume that may-point-to, may-alias, and must-alias analyses are available through the following functions:

- mayptsto $\in \mathbb{C} \times AP \to \mathcal{P}(AllocSite)$: Given a program point c and an access path p, mayptsto(c, p) returns the set of abstract objects that p may point-to at c.
- mayalias $\in \mathbb{C} \times AP \to \mathcal{P}(AP)$: Given c and p, it returns the set of access paths that may be aliased with p at c.
- mustalias $\in \mathbb{C} \times AP \to \mathcal{P}(AP)$: Given c and p, it returns the set of access paths that are definitely aliased with p at c.

We prepare these functions by running standard pointer and alias analyses [3, 14, 41] before the main analysis. Below, we assume mayalias and mustalias are lifted to receive sets as argument, e.g., mayalias$(c, \{x, y\})$ = mayalias$(c, x) \cup$ mayalias(c, y).

Given an object state $s = \langle o, may, must, mustNot, patch, patchNot\rangle$ at c, ϕ_c updates *may*, *must*, and *mustNot* as follows:

$$\phi_c(s) = \langle o, may', must', mustNot', patch, patchNot\rangle$$

where may' is defined as follows:

$$may' = \{p \in AP \mid o \in \text{mayptsto}(c, p)\} \setminus mustNot'.$$

It includes the access paths that may point-to the object, from which $mustNot'$ is removed to ensure the object invariant. The sets $must'$ and $mustNot'$ are defined depending on the type of statements. For example, when $cmd(c) = \text{alloc}(x)$, $must'$ and $mustNot'$ are:

$$must' = \text{mustalias}(c, must \setminus \text{mayalias}(c, \{x, *x\}))$$
$$mustNot' = \text{mustalias}(c, mustNot \setminus \text{mayalias}(c, \{x, *x\}) \cup \{x\})$$

Because x refers to a new object after the allocation, we remove all the access paths that are reachable from x (i.e. mayalias$(c, \{x, *x\})$) from *must* and *mustNot*. In addition, $mustNot'$ includes x since we know that x definitely does not point-to the old object. Other cases are defined similarly.

3.3 Solving Exact Cover Problem

The second step of MemFix is to establish and solve an exact cover problem. The static analysis computes safe and unsafe patches separately for each object. However, a patch that is safe for an object may be unsafe for others. Thus, MemFix aims to choose a set of patches that are simultaneously safe for all allocated objects. MemFix does so by solving an exact cover problem derived from the static analysis.

We first describe the basic method (Section 3.3.1), which captures the key idea behind our approach but works correctly with an assumption on the input program. We will explain the assumption and how to discharge it in Section 3.3.2.

3.3.1 Basic Method. Let $R \subseteq \mathbb{S}$ be the set of reachable states available at the exit node of the program according to the static analysis: i.e., $R = (\text{lfp}F)(c_x)$. Then, we define safe, unsafe, and candidate patches as follows:

$$Safe_R = \bigcup \{patch \mid \langle \_, \_, \_, \_, patch, \_\rangle \in R\}$$
$$UnSafe_R = \bigcup \{patchNot \mid \langle \_, \_, \_, \_, \_, patchNot\rangle \in R\}$$
$$Cand_R = Safe_R \setminus UnSafe_R$$

$Safe_R$ contains the patches that are guaranteed to safely deallocate some object. $UnSafe_R$ is the set of patches that may be unsafe for some object. Excluding $UnSafe_R$ from $Safe_R$, we obtain the set of candidate patches that we can use in repairing the program.

Let $M : Cand_R \to \mathcal{P}(R)$ be the function from candidate patches to the reachable states that can be safely deallocated by the corresponding patches:

$$M(c) = \{\langle o, may, must, mustNot, patch, patchNot\rangle \in R \mid c \in patch\}$$

For example, M describes the incidence matrix in Section 2.2. Then, the problem of finding correct patches is defined as follows.

Definition 3.1 (The Correct Patch Problem). Find a subset $C \subseteq Cand_R$ of candidate patches such that

- C covers the reachable states R, i.e., $R = \bigcup_{c \in C} M(c)$, and
- the chosen subsets in $M(c)$ (where $c \in C$) are pairwise disjoint, i.e., $M(c_1) \cap M(c_2) = \emptyset$ for all $c_1, c_2 \in C$.

The first condition means that all allocated objects must be deallocated (i.e. no memory-leaks). The second one means that every allocated object is deallocated no more than once (i.e. no double-frees). We guarantee the absence of use-after-frees as well because the patches that may cause use-after-free are all collected in $UnSafe_R$ and already excluded from $Cand_R$.

Note that this is an instance of the exact cover problem, a well-known NP-complete problem [13]. We solve the exact cover problem by encoding it as boolean satisfiability and leveraging an off-the-shelf SAT solver. Let $R = \{r_1, \ldots, r_m\}$ be the set of reachable object states and $Cand_R = \{c_1, \ldots, c_n\}$ be the set of candidate patches for R. Let $C \subseteq Cand_R$ be the solution of the patch problem (Definition 3.2). We introduce boolean variables S_i $(1 \le i \le n)$ and T_{ij} $(1 \le i \le n, 1 \le j \le m)$ to encode the solution of the patch problem and the function M:

$$S_i \iff c_i \in C, \qquad T_{ij} \iff r_j \in M(c_i).$$

That is, S_i is true iff the patch candidate $c_i \in Cand$ is included in the solution C, and T_{ij} is true iff the object state $r_j \in R$ is deallocated by the patch c_i. Then, we can encode the two conditions in Definition 3.2 by boolean constraints φ_1 and φ_2:

$$\varphi_1 = \bigwedge_{j=1}^{m} \bigvee_{i=1}^{n} T_{ij} \wedge S_i$$
$$\varphi_2 = \bigwedge_{j=1}^{m} \bigwedge_{i_1=1}^{n} \bigwedge_{i_2=1}^{n} \big((i_1 \neq i_2) \implies \neg\big((T_{i_1 j} \wedge S_{i_1}) \wedge (T_{i_2 j} \wedge S_{i_2})\big)\big)$$

The formula ϕ_1 encodes the first condition of Definition 3.2: for any reachable object r_j, some patch c_i in the solution must deallocate the object. The formula ϕ_2 encodes the second condition: for any reachable object r_j, two different patches c_{i_1} and c_{i_2} in the solution do not deallocate the object r_j at the same time. Finding a satisfying assignment of $\phi_1 \wedge \phi_2$, which assigns truth values to variables S_i,

determines the solution C. MemFix succeeds to repair the input program iff $\phi_1 \wedge \phi_2$ is satisfiable.

Note that in our approach, patches in the solution C always deallocate mutually exclusive concrete objects. This property is ensured by U in the patch transfer phase, which collects patches that are uncertain to free an object. The property is sufficient to ensure the safety of a generated patch, although we do not guarantee that abstract object states always represent mutually exclusive concrete objects.

3.3.2 Ensuring Safety during Patch Generation. Now we explain the assumption behind the basic method and how to address it. Consider the code: `p=malloc(); *p=malloc();`, where two objects o_1 and o_2 are allocated and pointed to by `p` and `*p`, respectively. Our method finds out that the object o_1 can be deallocated by `free(p)` and o_2 by `free(*p)` at the end of the code. Thus, the method generates one of the following two fixes:

```
(1)  p=malloc(); *p=malloc(); free(*p); free(p);
(2)  p=malloc(); *p=malloc(); free(p); free(*p);
```

However, the second one is not safe because the object pointed to by `p` is deallocated by `free(p)` and then dereferenced by the subsequent deallocation `free(*p)`, causing a use-after-free. Note that this type of use-after-free is caused by the inserted patches, not by the ordinary uses present in the original code (for which the our method guarantees the safety).

We can simply address this problem by assuming that the input program is written in a way that a temporary variable is introduced whenever a pointer expression is dereferenced. For example, we assume that the code above has been transformed to the following before we apply our algorithm: `p=malloc(); *p=malloc(); tmp=*p;` where variable `tmp` is created to store the value of the pointer expression `*p`. Then we can avoid the problem of the basic method by generating patches whose pointer expressions are always program variables: e.g.,

```
p=malloc(); *p=malloc(); tmp=*p; free(p); free(tmp);
```

This does not cause use-after-free errors as no pointers are dereferenced by the deallocation statements inserted by our algorithm.

Another way of addressing the problem without introducing temporary variables is to extend the previous algorithm to consider the additional constraint that a patch should not use an object that was previously deallocated by other patches. Let $R_c \subseteq \mathbb{S}$ be the set of reachable states available at the node c: i.e., $R_c = (\text{lfp}F)(c)$. We write R for $\bigcup_{c \in \mathbb{C}} R_c$. The function $M : Cand_R \to \mathcal{P}(R)$ is defined in the same way but with the new R:

$$M(c) = \{\langle a, may, must, mustNot, patch, patchNot\rangle \in R \mid c \in patch\}$$

We define $U : Cand_R \to \mathcal{P}(R)$, which is the function that maps candidate patches to reachable objects that may be used by the pointer expressions of the patches: $U((c, x)) = \emptyset$ and

$$U((c, *x)) = \{\langle o, may, must, mustNot, patch, patchNot\rangle \in R_c \mid o \in \text{mayptsto}(c, x) \wedge x \notin mustNot\}$$

A patch of the form (c, x) does not use any object state. A patch $(c, *x)$ may access the objects that x may point to at c.

In order for a patch to be safe (i.e. no use-after-free), the patch should not use an object that was previously deallocated by another patch. To ensure this, for any pair $(c_1, c_2) \in C \times C$ of selected patches, M and U should be disjoint, i.e., $M(c_1) \cap U(c_2) = \emptyset$. The extended patch problem is defined as follows.

Definition 3.2 (The Extended Patch Problem). Find a subset $C \subseteq Cand_R$ of candidate patches such that

- C covers the reachable states R_{c_x} at the exit node, i.e., $R_{c_x} = \bigcup_{c \in C} M(c)$, and
- the chosen subsets in $M(c)$ (where $c \in C$) are pairwise disjoint, i.e., $M(c_1) \cap M(c_2) = \emptyset$ for all $c_1, c_2 \in C$.
- the subsets in $\{M(c)\}_{c \in C}$ and $\{U(c)\}_{c \in C}$ are disjoint, i.e., $M(c_1) \cap U(c_2) = \emptyset$ for all $c_1, c_2 \in C$.

The last condition can be easily encoded by a boolean formula in a similar way described above. Our implementation solves the extended patch problem and does not introduce temporary variables.

4 EVALUATION

In this section, we experimentally evaluate MemFix. The main objective is to evaluate the ability of MemFix in fixing memory deallocation errors in practice. We also compare MemFix with the existing tool, LeakFix [15], for fixing memory leaks. All experiments were done on a linux machine with Intel Core i5-4590 and 8GB RAM.

4.1 Implementation

We implemented a prototype of MemFix on top of Sparrow[6], which provides a general framework for performing abstract interpretation of C programs. We instantiated the framework with the abstract domain and semantics function described in Section 3. Our focus is on faithfully implementing the analysis and algorithm in Section 3, rather than on optimizing its performance. We discuss the detail and assumption of the current implementation below.

Although we presented the algorithm for the simple language in Section 3.1, our implementation supports most of the features of the C programming language, including procedure calls, structures, pointer arithmetics, etc. The current implementation of the patch-generating static analysis (Section 3.2) performs a fully context-sensitive (except for recursion) using the call-strings method [38]. As pre-analysis, we use a standard context-insensitive and flow-sensitive may-points-to and may-alias analysis and a context- and flow-sensitive must-alias analysis [3]. Our implementation supports the general class of pointer access paths in C programs, including structure fields and a chain of pointer dereferences. However, because the number access paths can be infinite in the presence of dynamic arrays and inductive data structures (e.g., linked list), we approximate them using a pre-determined bound on their length. The current bound is the maximum length of the longest pointer access path in the given program.

Our implementation supports the C standard memory-allocators (`malloc` and `calloc`) except for `realloc`. To support the full semantics of `realloc`, a patch must introduce a conditional deallocation statement, which is beyond the scope of the current algorithm. We also encoded some memory-allocators in our benchmarks such as `strdup` for the evaluation.

We implemented the algorithm to choose an optimal patch when multiple patches are available. It is easy to modify our algorithm

[6] https://github.com/ropas/sparrow

to find a patch that is optimal according to some criteria. For example, suppose we would like to find a set of patches such that (1) the number of inserted deallocation statements is minimal and (2) objects are deallocated as early as possible. To find such an optimal solution, we use a partial MAX-SAT solver with the soft constraints (1) and (2). In particular, we deallocate objects as early as possible by maximizing the number of object states that are deallocated by patches.

4.2 Benchmarks

We evaluated MemFix with three benchmark sets: Juliet Test Suite for C [4], GNU Coreutils[7], and a set of model programs abstracted from popular open-source C repositories. The first two benchmark sets are well-known for evaluation of program analysis tools [6, 16, 25, 32]. The last one was created by us with the goal of evaluating software repair tools for memory deallocation errors. We make this benchmark set publicly available with our tool, so that it can be objectively used in the future. All reported LoCs are counted without comment and blank lines.

Juliet Test Suite. Juliet Test Suite consists of small but diverse programs that contain 118 CWE vulnerabilities. We used a subset of the collection, comprising 210 testcases, relevant to memory leak (CWE-401), double-free (CWE-415), and use-after-free (CWE-416). For each error type (e.g., memory leak), the benchmark set contains a variety of vulnerability patterns that differ in syntax and semantics. Testcases for each CWE were categorized by their functional variants that describe flaw types of CWE. For example "int_malloc" means a memory block allocated to integer pointer leaks. Among the functional variants, we did not consider semantically redundant testcases such as ones that are only different in types (e.g., (char*)malloc, (int*)malloc)). We also excluded the testcases for `realloc`.

Coreutils. Among the total 126 programs in the Coreutils-8.29 package, we chose 24 programs that use dynamic memory allocation, have the main function, and do not use `realloc`.

Open-Source Repositories. We collected real memory deallocation errors from popular open-source C repositories. The benchmarks consist of a set of model programs of 55–664 LoC abstracted from the most recent 100 error-fixing commits in 5 GitHub open-source C repositories. For each commit, we created a model program that captures the key reason of the memory deallocation errors. We did our best to preserve the original features of the program (e.g., pointer arithmetic, data structures and function pointers) so that the parts of the program related to the errors remain intact in the model programs. Therefore, although the model programs are small compared to the original, fixing the errors in them is nontrivial. We chose five C repositories which have at least 20 memory deallocation error commits including at least 10 double-free and use-after-free fix commits. We collected 20 error fixing commits for each repository from the end of 2017 year in reverse: 10 commits from memory-leak and 10 from double-free and use-after-free.

4.3 Results

[7] https://www.gnu.org/software/coreutils/coreutils.html

Table 2: Evaluation results on Juliet Test Suite.

CWE-ID	Functional Variants	Bugs	MemFix	LeakFix
401	int_malloc	38	38	29
	struct_malloc	38	38	29
415	free_int	38	38	20
	free_struct	38	38	19
416	malloc_free_int	20	20	20
	malloc_free_struct	20	20	20
	return_freed_ptr	18	18	0
Total		210	210	137

Table 3: Evaluation on GNU Coreutils.

			MemFIx		LeakFix	
Programs	LoC	#Al.	#Ins.	sec	#Ins.	sec
yes	553	1	1	< 1.0	✗	< 1.0
users	577	1	1	< 1.0	✗	< 1.0
unexpand	707	1	1	< 1.0	✗	< 1.0
tee	779	1	1	< 1.0	1	< 1.0
mktemp	794	4	✗	1.3	✗	< 1.0
tsort	920	3	✗	1.4	✗	< 1.0
paste	982	3	3	2.4	△/3	< 1.0
date	1,054	1	1	3.5	✗	< 1.0
cut	1,056	1	✗	2.0	✗	< 1.0
nl	1,063	4	4	4.0	✗	< 1.0
pinky	1,120	3	4	5.2	✗	< 1.0
cat	1,209	3	✗	9.3	✗	< 1.0
ln	1,258	2	✗	5.2	✗	< 1.0
printf	1,288	1	1	3.0	✗	< 1.0
stdbuf	1,605	3	3	1.3	✗	< 1.0
wc	1,669	1	1	7.3	△/2	< 1.0
shred	1,822	5	✗	31.1	✗	< 1.0
cp	1,926	8	✗	430.7	✗	< 1.0
install	2,076	1	✗	13.4	✗	< 1.0
who	2,156	8	✗	36.8	✗	< 1.0
tr	2,304	10	✗	20.0	✗	< 1.0
expr	2,378	9	✗	13.0	✗	< 1.0
stat	2,439	10	6	130.3	✗	< 1.0
dd	3,475	2	✗	52.2	✗	< 1.0

Juliet Test Suite. Table 2 shows the results on Juliet Test Suite. We provided the buggy versions of the programs to MemFix as input, and manually checked the correctness of patches. To evaluate LeakFix, we removed all free-statements from the programs, generating memory leaks, and checked whether LeakFix can patch the buggy programs. The results show that MemFix succeeds to patch all testcases while LeakFix is able to fix 137 among 210. Both tools took less than a minute to generate the patches for each testcase.

GNU Coreutils. Table 3 shows the results on 24 programs from Coreutils. For evaluation, we made those programs buggy by removing all free-statements. The goal of the evaluation here is to see whether MemFix and LeakFix can automatically repair the memory leaks without causing other types of errors. In this set

Table 4: Evaluation on open-source C repositories

Repo.	ML	DF	UAF	Total
	Fix/#Pgm.	Fix/#Pgm.	Fix/#Pgm.	Fix/#Pgm.
Binutils	4/10	1/5	2/5	7/20 (35%)
Git	1/10	1/4	2/6	4/20 (20%)
OpenSSH	6/10	5/7	1/3	12/20 (60%)
OpenSSL	5/10	3/5	1/5	9/20 (45%)
Tmux	5/10	0/3	0/7	5/20 (25%)
Total	21/50 (42%)	10/24 (42%)	6/26 (23%)	37/100 (37%)

of experiments, we manually replaced `(x)strdup` by semantically-equivalent code using `malloc` in order for LeakFix to understand.

#Al. reports the number of target allocation-sites in the programs. #Ins. is the number of free-statements inserted by the tools when the repair process is successful. We report a program is fixed if all the target allocation-sites are adequately deallocated by a generated patch. We manually checked the correctness of the generated patches. The results show that MemFix can repair 12 out of 24 programs. LeakFix generated patches for 3 programs, of which 2 programs were fixed partially (i.e. some errors remain).

Open-source C Repositories. Table 4 shows the results on the model programs constructed from open-source repositories. For the total of 100 model programs, where one program contains a single error, MemFix was able to 37% of them. For memory leak (ML), MemFix succeeded to fix 42% (21/50) on average. For double-free and use-after-free, MemFix was able to generate patches for 33.3% (8/24) and 23.1% (6/26) of the errors. The public version of LeakFix was unstable to handle these programs and we failed to objectively compare MemFix with LeakFix on this benchmark. For example, LeakFix often produced obviously-incorrect patches (e.g. freeing a variable twice) when aliased pointers are used extensively.

We found that these open-source programs frequently use low-level C features, and therefore fixing memory deallocation errors in them is much more challenging than the benchmark programs in Juliet Test Suite or Coreutils. For example, the model programs collected from Git often store memory objects in arrays or linked lists and extensively use reallocation (i.e. `realloc`) to process a set of strings (e.g. directories). Since the current version of MemFix cannot effectively handle such features, the portion of successful patches is relatively low in Git. OpenSSH usually uses primitive allocators and deallocator to manage allocated objects except for key-related objects. OpenSSH manages memory of key structures conditional to key-types, which requires tracking the type flags of allocated objects to precisely collect patch candidates. Allocators of OpenSSL are quite complex since they allocate memory according to types; behaviors of allocators(e.g. allocation of fields or not) of OpenSSL depends on its type. Tmux, a terminal multiplexer, manages its allocated objects by red-black-tree data structure. Consequently, we failed to track must-points-to access paths for patch candidates.

4.4 Limitations

MemFix has a number of limitations as well. We identify them and discuss how to overcome the limitations below.

```
1  int *foo(struct st *p) {
2    int *q;
3    if(p->flag) q = malloc(1);
4    else q = p->f;
5    return q;
6  }
7  int main() {
8    struct st p; int *q;
9    p.f = malloc(1);
10   q = foo(&p);
11   // use q
12   free(q);
13   free(p.f); // double-free
14 }
```

(a) Double-free

```
1  int main() {
2    p = malloc(n);
3    for(i=0; i<n; i++)
4      p[i].f = malloc(1);
5    // use p[i].f
6    if(...)
7      return 1; // leak
8    for(i=0; i<n; i++)
9      free(p[i].f);
10   return 0;
11 }
```

(b) Memory leak

Figure 4: Errors that MemFix cannot fix.

One limitation of MemFix is that it can only fix an error by inserting or removing deallocation statements. In experiments with open-source repositories (Table 4), we found that some memory deallocation errors in the wild require other fixing strategies such as inserting conditional statements, temporary variables, or changing passing pointers (e.g., dst = src) to newly allocated object (e.g., dst = strdup(src)). A common failure point is when a new conditional statement is required to fix the error. For example, consider the program in Figure 4(a), simplified from Git. To fix the double-free error at line 13 without causing memory leak, we have to deallocate `q` at line 12 only when the flag `p->flag` is true. That is, we need to modify `free(q)` at line 12 into `if(p->flag) free(q);` by introducing a conditional statement.

Another situation, where MemFix fails, is when deallocation statements exist implicitly in code. For example, the following code (1) shows one example of use-after-free error which cannot be fixed by MemFix found in Git.

```
(1)  p=alloc(); q=p; p=realloc(p, sz); use(q);
(2)  p=alloc(); q=strdup(p); p=realloc(p, sz); use(q);
```

In this code, `q` points-to the object allocated at the first statement, but this object may be deallocated by realloc-statement and use-after-free may occur at `use(q)`. However, this code cannot be fixed by inserting or deleting free-statements since free-statement does not exist explicitly. The developer fixed this code by allocating a new object and passing it to `q` instead of the existing object which can be deallocated by the realloc-statement.

Another limitation comes from the pointer analysis. For example, Figure 4(b) shows an example program, where MemFix cannot fix the memory-leak error at line 7. To fix the error, the pointer analysis needs to be able to separately consider the array elements. However, as the pointer analysis employed by MemFix abstracts all array elements as a single abstract location, MemFix cannot analyze the code precisely. We also found that the context-insensitive may-pointer information is not precise enough to generate patches in practice.

Finally, the scalability of static analysis should be improved. We plan to adopt advanced analysis techniques (e.g., [33, 34]) accumulated in the static analysis community over the last decades.

5 RELATED WORK

A large amount of work on automatic program repair has been done over the last decade (e.g., [1, 7, 12, 15, 21–23, 26, 28–31, 39, 40, 43, 46, 49–52]). We classify the existing work into special-purpose and general-purpose techniques and compare them with MemFix. We also discuss techniques for automatic memory management.

Special-Purpose Program Repair Techniques. MemFix belongs to the family of techniques that focus on fixing specific classes of errors. Existing techniques aim to automatically fix memory leak [15, 43, 52], buffer overflow [39], integer overflow [7], null pointer exceptions [12], error handling bugs [46], race errors [1], etc. To our knowledge, MemFix is the first technique that is able to fix memory deallocation errors (i.e., memory-leak, double-free, and use-after-free) in a unified fashion.

In particular, existing techniques for fixing and diagnosing memory leaks [9, 15, 43, 51, 52] cannot fix double-frees or use-after-frees. Gao et al. [15] present a technique, called LeakFix, that automatically fixes memory-leak errors in C programs. LeakFix uses pointer and dataflow analyses to safely insert a `free` statement into a program point where the allocated object can be deallocated only once without being used subsequently. Sonobe et al. [43] present a type-based technique for fixing memory leaks in imperative programs. The technique is based on the type system using fractional ownerships [18, 44, 45], which guarantees that well-typed programs do not have memory-deallocation errors at runtime. The idea is to run the existing type inference algorithm [45] and annotate the input program with type-casts, which express how ownerships of pointers should be converted. For each implicit type-cast in the resulting program, a `free` statement is introduced to make the type-conversion explicit. Yan et al. [52] present a technique that combines static and dynamic analysis to avoid memory leaks. These techniques are only able to insert deallocation statements and therefore limited to fixing memory leaks. LeakPoint [9] and LeakChaser [51] are essentially localization tools and cannot fix errors automatically. By contrast, MemFix is designed to fix memory deallocation errors in general with a novel algorithm that solves the exact cover problem induced by a static analysis.

General-Purpose Program Repair Techniques. The general test-based approaches (e.g., [21–23, 26, 28–31, 40, 49, 50]) to automatic program repair is not adequate for fixing memory deallocation errors. These approaches work with a set of testcases, some of which expose the error in the program, and aim to find a patched program that behaves correctly on all inputs in the test-suite. These techniques can be classified into generate-and-validate and semantics-based techniques. Generate-and-validate approaches [21, 26, 28, 49] repeatedly search for candidate patches within a pre-defined search space until a program that can be validated against the test-suite is found. Semantics-based approaches [22, 23, 29–31] use symbolic execution to derive contraints on the correct patch and synthesize the patch by using program synthesis or constraint solving. Although remarkable progress has been made, it is hard to fix memory deallocation errors with a test-based technique. Besides their inherent overfitting problem [27, 36, 42, 54, 55], the safety condition for memory deallocation (i.e., no memory-leaks, double-frees, and use-after-frees) cannot be completely specified with input-output testcases.

A few techniques enhance the test-based approaches by using, for example, program verification [24], metamorphic testing [19], and contracts [35]. Although these techniques mitigate the overfitting problem [24, 35] or relieves the burden of writing testcases [19], it is still nontrivial to provide a complete specification for proving the absence of memory deallocation errors. In this work, we present a static-analysis-based approach, where the correctness conditions are automatically generated by a static analysis.

Automatic Memory Management Techniques. Several techniques [2, 8, 11, 17, 37] have been proposed to optimize memory usage in automatic memory management systems, which also inserts deallocation statements in a program. However, their goal is to optimize performance, rather than fixing memory deallocation errors. Aiken et al. [2] introduced explicit region operations (i.e., region allocation and deallocation) and presented an algorithm to improve efficiency of a region-based memory management system [47, 48] by inserting allocation and deallocation statements. However, this technique works only on region type-annotated programs. Shaham et al. [37] presented a static analysis to verify safety of free-inserted Java programs, which can verify the safety of list-manipulating programs but requires manual insertion of deallocation statements. Compile-time object deallocation techniques [8, 17] insert free-statement in Java bytecode programs at compile-time to reduce overhead of garbage collection. However, these techniques insert free only at unreachable points and cannot free all objects. Dillig et al. [11] presented an automated resource management technique to optimize resource usage in Java. This technique analyzes and approximates lifetimes of objects and inserts static and dynamic disposal statements.

6 CONCLUSION

Debugging memory-deallocation errors is a taxing and error-prone task. In this paper, we presented MemFix, a new technique for automatically debugging memory-leak, double-free, and use-after-free errors in C programs. Experimental results show that MemFix is able to repair various errors from open-source programs.

Notably, MemFix is based on a sound static analysis, which provides several fundamental benefits. First, it sheds light on the connection between the problem of finding correct patches for memory-deallocation errors and the exact cover problem. Second, it formally guarantees the correctness of the patches; the MemFix-generated patches eliminate the target error without introducing new errors. As future work, we plan to push this direction towards deploying the technology in practical development setting.

Acknowledgement. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B2014062).

REFERENCES

[1] 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

[2] 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

[3] George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. 2017. A Datalog Model of Must-alias Analysis. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. 7–12.

[4] Frederick E Boland Jr and Paul E Black. 2012. The Juliet 1.1 C/C++ and Java test suite. *Computer (IEEE Computer)* 45, Computer (IEEE Computer) (2012).

[5] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 133–143. https://doi.org/10.1145/2338965.2336769

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[7] 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

[8] 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

[9] 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

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[11] 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

[12] 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

[13] Exact Cover. 2018. Exact Cover — Wikipedia, The Free Encyclopedia. (2018). https://en.wikipedia.org/wiki/Exact_cover Accessed: 2018-03-01.

[14] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. 133–144.

[15] 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 *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. 459–470.

[16] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology* 68 (2015), 18–33.

[17] 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

[18] 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

[19] 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

[20] Tommi Junttila and Petteri Kaski. 2010. Exact Cover via Satisfiability: An Empirical Study. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10)*. 297–304.

[21] 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

[22] 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

[23] 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

[24] 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

[25] Hongzhe Li, Jaesang Oh, Hakjoo Oh, and Heejo Lee. 2016. Automated source code instrumentation for verifying potential vulnerabilities. In *IFIP International Information Security and Privacy Conference*. Springer, 211–226.

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. 2016. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 780–785.

[33] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 229–238. https://doi.org/10.1145/2254064.2254092

[34] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

[35] 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

[36] 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

[37] 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.

[38] Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis*. Prentice-Hall, Englewood Cliffs, NJ, Chapter 7, 189–234.

[39] 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

[40] 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

[41] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

[42] 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

[43] 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.

[44] 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.

[45] 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.

[46] Yuchi Tian and Baishakhi Ray. 2017. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 752–762. https://doi.org/10.1145/3106237.3106300

[47] 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

[48] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.

[49] 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

[50] 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

[51] 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

[52] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2016. Automated Memory Leak Fixing on Value-flow Slices for C Programs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. 1386–1393.

[53] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*. 42–54.

[54] 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

[55] 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