electronics

Article

SafeMD: Ownership-Based Safe Memory Deallocation for C Programs

Xiaohua Yin [1], Zhiqiu Huang [1,*], Shuanglong Kan [2] and Guohua Shen [1]

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China; xhyin@nuaa.edu.cn (X.Y.); ghshen@nuaa.edu.cn (G.S.)
[2] Department of Computer Science, TU Kaiserslautern, 67653 Kaiserslautern, Germany; shuanglong@cs.uni-kl.de
* Correspondence: zqhuang@nuaa.edu.cn

Abstract: Rust is a relatively new programming language that aims to provide memory safety at compile time. It introduces a novel ownership system that enforces the automatic deallocation of unused resources without using a garbage collector. In light of Rust's promise of safety, a natural question arises about the possible benefits of exploiting ownership to ensure the memory safety of C programs. In our previous work, we developed a formal ownership checker to verify whether a C program satisfies exclusive ownership constraints. In this paper, we further propose an ownership-based safe memory deallocation approach, named SafeMD, to fix memory leaks in the C programs that satisfy exclusive ownership defined in the prior formal ownership checker. Benefiting from the C programs satisfying exclusive ownership, SafeMD obviates alias and inter-procedural analysis. Also, the patches generated by SafeMD make the input C programs still satisfy exclusive ownership. Usually, a C program that satisfies the exclusive ownership constraints is safer than its normal version. Our evaluation shows that SafeMD is effective in fixing memory leaks of C programs that satisfy exclusive ownership.

Keywords: C; memory leaks; memory deallocation; Rust; ownership

Citation: Yin, X.; Huang, Z.; Kan, S.; Shen, G. SafeMD: Ownership-Based Safe Memory Deallocation for C Programs. *Electronics* **2024**, *13*, 4307. https://doi.org/10.3390/electronics13214307

Academic Editor: Carlo Mastroianni

Received: 27 September 2024
Revised: 29 October 2024
Accepted: 31 October 2024
Published: 1 November 2024

1. Introduction

C is widely used for implementing system and embedded software, which are usually safety-critical systems [1,2]. However, their manual memory management can easily produce memory leaks (MLs) in C programs. Memory leaks mainly occur when a programmer allocates an object but forgets to deallocate it. Memory leaks may have a large negative impact on software systems if not carefully examined and fixed. In fact, memory leaks are direct sources of security vulnerabilities. Some memory leak vulnerabilities have been disclosed in Linux kernels (e.g., CVE-2022-27819 [3], CVE-2017-10810 [4]).

Recently, an emerging programming language designed for highly safe systems, i.e., Rust [5], has received an increasing amount of attention. Compared with C/C++, Rust introduces an ownership system (OwS) to provide memory safety at compile time, which can avoid many memory errors, such as dangling pointers, data races and memory leaks. The basic idea of OwS is exclusive ownership, i.e., at any time, each resource has a unique owner. When the unique owner of a resource goes out of its scope, the resource can be automatically dropped without using the garbage collector. Because of the unique owner, this automatic drop scheme is safe, i.e., it does not incur new errors like use-after-free (UAF) and double-free (DF).

In light of Rust's promise of safety, the emergence of OwS in Rust provides a new insight to guarantee the memory safety of C programs. Therefore, in our previous work [6], as shown in Figure 1, we developed a formal ownership checker, named SafeOSL, to verify whether a C program satisfies the exclusive ownership constraint. If a C program passes the checking of SafeOSL, it means that this C program satisfies exclusive ownership. For such a special C program, in this paper, we further propose an ownership-based memory deallocation, named SafeMD, to fix memory leaks. The output of SafeMD is a C program that is free of memory leaks, and, most importantly, the C program, after the repair of SafeMD, still satisfies exclusive ownership. Usually, a C program that satisfies the exclusive ownership is safer than its normal version.

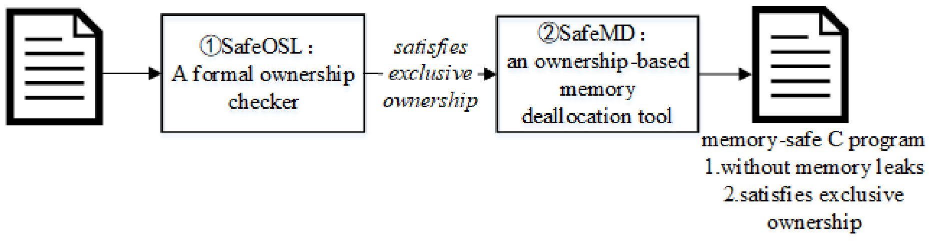
①SafeOSL : A formal ownership checker
satisfies exclusive ownership
②SafeMD : an ownership-based memory deallocation tool
memory-safe C program
1.without memory leaks
2.satisfies exclusive ownership

Figure 1. The ownership-based framework of ensuring memory safety of C programs.

Many static techniques on memory-leak fixing have been proposed in the program repair community [7–11]. If the input C programs satisfy exclusive ownership, these techniques suffer from some drawbacks when fixing memory leaks. Firstly, some techniques can fix memory leaks but may introduce new errors, like UAF and DF. Secondly, some techniques can safely fix memory leaks but are complex as they often rely on alias and inter-procedural analysis. Thirdly, some techniques can safely fix memory leaks, but the patches that they generate cannot guarantee that the C programs satisfy the exclusive ownership. Usually, a C program that satisfies the exclusive ownership is safer than its normal version.

In this paper, we propose an ownership-based memory deallocation, named SafeMD, to fix memory leaks of C programs that satisfy exclusive ownership. SafeMD can generate a set of free statements to safely deallocate all allocated memory objects without introducing UAF and DF. SafeMD includes two steps: (1) using a static analysis that collects patch candidates for each allocated object. The main idea of collecting patch candidates is to track ownership of the object and free the object where the owner last used it. Benefiting from the input C programs satisfying exclusive ownership, this step obviates alias and inter-procedural analysis. Also, the patch candidates collected satisfy exclusive ownership. (2) Finding correct patches by solving an exact cover problem. Because ownership designates which function is responsible for deallocating memory objects, SafeMD simplifies inter-procedural analysis to intra-procedural analysis, and each analysis performs the above two steps.

The experimental results demonstrate that SafeMD is able to fix the memory leaks in C programs. We evaluated SafeMD with two different benchmark sets: Juliet Test Suite (JTS) for C [12] and 26 open-source C repositories [10]. We compared SafeMD with MemFix. When the input C programs satisfy exclusive ownership, SafeMD can fix more memory-leak patterns than MemFix.

We summarize the contributions of this paper as follows.

- We present SafeMD, an ownership-based safe memory deallocation technique for C programs that satisfy exclusive ownership. Compared with the existing techniques, SafeMD obviates alias and inter-procedural analysis, and the patches generated satisfy exclusive ownership.
- We implement SafeMD and compare it with MemFix.
- We explore the benefit of Rust's novel ownership-based memory management in C.

2. Related Work

2.1. Approaches for Memory-Leak Fixing

Some prominent techniques have been proposed to statically fix memory leaks. Leakfix [7] performs pointer analysis on the whole C program to identify and safely fix memory leaks. Each procedure is classified into three types: those that allocate, deallocate or use a given memory allocation. It first abstracts the program into an abstract control flow graph

(CFG) where each node is a procedure classified into above three types. With this graph, the task of finding correct patches is equivalent to finding edges in the graph that meet a set of conditions. AutoFix [8] combines static analysis with runtime checking to prevent memory leaks. In its static analysis, Andersen′s pointer analysis is used to build the value-flow graph (VFG) for the program. Based on the VFG, AutoFix performs a graph reachability analysis to identify leaky paths and then conducts a liveness analysis to locate the program points for inserting patches on identified leaky paths. FootPatch [9] can fix memory leaks by applying local reasoning based on separation logic. But, it may introduce new errors as it checks the patch correctness against the given error report only. Memfix [10] can safely repair ML, DF and UAF in a unified fashion. The key insight behind MemFix is that finding a correct patch for memory leaks corresponds to solving an exact cover problem. Before analysis, Memfix performs standard pointer [13] and alias analyses [14]. SAVER [11] can safely fix memory errors such as ML, UAF and DF. It performs pointer analysis to construct object flow graphs (OFGs) that capture the program's heap-related behavior. Based on the OFG, fixing memory errors can be formulated as a graph-labeling problem over the OFG.

For the C programs that satisfy exclusive ownership, most of the work mentioned above still performs alias and inter-procedural analysis to fix memory leaks, making the repair complex and inefficient. However, the exclusive ownership satisfied by input programs can ease memory-leak fixing since exclusive ownership entirely rules out aliases. Also, the patches generated by the existing work may violate exclusive ownership. Therefore, this paper proposes an approach that exploits the particularity of ownership to fix memory leaks for C programs that satisfy exclusive ownership.

Several dynamic-based techniques have been proposed [15–17]. DEF_LEAK [15] performs dynamic symbolic execution to expose memory leaks occurring in all execution paths. In their approach, the program to be analyzed is instrumented before execution. During the program execution, information about each allocated memory is updated when corresponding statements are executed. Based on this information, DEF_LEAK records the changes in variables pointing to each memory, detects memory leaks and fixes leaked memory. LeakPoint [16] is a dynamic analysis framework that performs taint propagation on pointers to detect memory leaks. It can identify last-use sites of the leaked objects and suggest the patches for fixing them. AddressWatcher [17] is a dynamic tool for fixing memory leaks. It allows the semantics of a memory object to be tracked on multiple execution paths. It accomplishes this by using a leak database that allows one to store and compare different execution paths of a leak over several test cases.

2.2. Ownership for Memory Safety

Ownership has been used in OO programming to enable controlled aliasing [18,19] and prevent data races [20,21]. Most of these works construct an ownership-type system for Java and require programmers to provide various annotations. A small number of the works have applied ownership to detect memory errors of C programs. Heine et al. [22] present an ownership-type system to detect ML and DF. Their ownerships range over integer values $\{0, 1\}$. In their model, every object is pointed to by one and only one owning pointer (i.e., ownership value equals 1), which holds the exclusive right and obligation to either delete the object or to transfer the right to another owning pointer. However, the rules of their ownership model are not very strict; for example, it adds an optional ownership transfer in assignment and thus allows for arbitrary aliases. Swamy et al. [23] develop a language Cyclone that introduces a simple concept similar to ownership to detect dangling pointers. Unlike C, their language requires programmers to provide various annotations (such as whether a pointer is aliased or not). Suenaga et al. [24] propose a fractional-ownership-type system to detect ML, DF and UAF in C. Their model augments a pointer type with a fractional ownership, which is a rational number $x \in [0, 1]$. In their ownership model, a non-zero ownership expresses a permission to dereference the pointer, and an ownership of 1 expresses a permission to update and deallocate the memory cell referenced by the pointer. Therefore, if one has a non-zero ownership less than 1, one has to eventually combine it with other ownerships to obtain an ownership of 1 in order to deallocate the pointer. Sonobe et al. [25] extend the fractional-ownership-type system in [24] to fix memory leaks. Their technique conducts type inference for the extended-type system to detect where to insert deallocation statements.

In recent years, ownership in Rust has received much attention. A majority of existing work toward Rust mainly focuses on formal verification (including ownership) of Rust programs [26–28] and empirical research on the effectiveness of Rust ownership in fighting against memory bugs [29,30]. Recently, some work on (semi-)automatically translating C code to Rust has been proposed [31–33]. Compared to the ownership proposed in the earlier literature (before Rust was released), the potential advantages of Rust ownership are: (1) it has more strict rules; (2) its implementation is simpler and more efficient than fractional-based ownership; (3) it has been proven to be effective in preventing memory errors [30,34]. Therefore, in our previous work [6], we exploit Rust ownership to check for memory errors of C programs, and, in this paper, we further exploit Rust ownership to fix memory leaks for C programs that satisfy exclusive ownership.

3. Ownership System in Rust

Rust guarantees memory safety at compile time by introducing an ownership system and consequently avoids many memory errors, such as dangling pointers and memory leaks. Ownership in Rust denotes a set of rules that govern how the Rust compiler manages memory. The idea of OwS is exclusive ownership, which means that each resource has a unique variable as its owner at any time. Ownership can be transferred among owners. When the owner of a resource goes out of its scope, the resource can be automatically dropped without using the garbage collector. Below, we introduce ownership transferring.

Ownership and Assignments. In Rust, ownership can be transferred in assignments. Consider the code in Listing 1, where line 2 creates a String object o on the heap, and let $s1$ be the owner of o. At line 4, the assignment transfers the ownership of o from $s1$ to $s2$. To maintain the unique owner, the assignment performs *move* semantics, which makes $s1$ become the old owner and no longer valid until it is re-assigned a value again. Therefore, the Rust compiler will issue an error at line 5. This is different with pointer assignments in C, where both $s1$ and $s2$ are valid and can be used. Because $s2$ is the unique owner of o, when the owner $s2$ goes out of its scope, the Rust compiler automatically inserts a `drop` destructor to free o at line 6, which can avoid memory leaks. Therefore, the code at line 7 is rejected as o has already been destroyed. Now, we take a closer look at line 8. When $s1$ goes out of its scope and tries to free o, the Rust compiler does not insert `drop` since it finds that the ownership of $s1$ has been moved. This means that an object cannot be freed by any of its old owners (like $s1$), which can ensure that memory deallocation does not introduce DF.

Listing 1. Transferring ownership in assignments.

```
1 fn main(){
2   let s1 = String::from("hello");
3   {
4    let s2 = s1; // ownership is moved to s2.
5    println!("{}", s1);  // s1 is no longer valid.
6   }  // s2 goes out of scope and 'drop' is called.
7   println!("{}", s2); //memory is freed via s2.
8 }  // s1 was moved, so nothing happens.
```

Ownership and Functions. Ownership can also be transferred in function calls. When ownership of an object is moved to a callee via parameters, this object is no longer available in the caller. For example, in Listing 2, the function call at line 3 moves the ownership of the object o created at line 2 to `takes_ownership`, so the Rust compiler will issue an error at line 4, where s becomes the old owner and cannot be accessed in the `main` function. The Rust compiler performs intra-procedural analysis to insert a `drop` destructor, which compiles each function individually. It relies on ownership to determine whether the current function has a responsibility to free objects. For example, the Rust compiler first compiles the `main` function. It finds that s is moved to `takes_ownership`; therefore, the `main` function has no responsibility to free the object o. The Rust compiler does not insert `drop` at line 5. Next, the Rust compiler compiles the `takes_ownership` function. Because ss is a String type that can move ownership, the Rust compiler will automatically insert `drop` once ss goes out of its scope at line 9.

Listing 2. Transferring ownership via parameters.

```
1  fn main() {
2      let s = String::from("hello");
3      takes_ownership(s); // s is moved
4      println!("{}", s); // s is invalid
5  }  // s was moved, so nothing happens.
6
7  fn takes_ownership(ss: String){
8      println!("{}", ss);
9  }  // ss goes out of scope and 'drop' is called.
```

Besides parameter passing, return values can also transfer ownership. For example, in Listing 3, `gives_back` moves ownership out from `gives_back` via return value ss to its caller `main`, which means that `gives_back` has no responsibility to free the object. Exclusive ownership can ensure that automatic memory deallocation is safe. When the Rust compiler compiles the `main` function, the object o is dropped automatically once $s1$ goes out of scope at line 6 but nothing happens for s because s is moved. This avoids DF when s and $s1$ go out of their scope (line 6) and both try to free the object o. When compiling the `gives_back` function, it fails to insert `drop` to free the object pointed to by ss since ss is moved out from `gives_back`. This can avoid UAF if ss is freed while $s1$ is used in the `main` function.

In conclusion, the main ideas of OwS are summarized as follows.

R1: Each resource has a unique owner at any time.

R2: When the owner of a resource goes out of its scope, it deallocates the resource that it owned. Any old owners of the resource cannot deallocate the resource (this can avoid DF and UAF).

In our previous work, SafeOSL ensures that a C program satisfies R1. For such a special C program, SafeMD proposed by this paper borrows the idea of R2 to fix its memory leaks.

Listing 3. Transferring ownership via return values.

```
1  fn main() {
2      let s = String::from("hello");
3      let s1 = gives_back(s);
4      println!("{}", s); // s is invalid
5      println!("{}", s1);
6  }  // s1 can free, s can not free as it is moved.
7
8  fn gives_back(ss: String) -> String {
9      ss  // return ss
10 }  // ss can not free as it is moved.
```

4. Approach Overview

We illustrate the algorithm of SafeMD using a simple example in Listing 4. This code satisfies the exclusive ownership. For example, `t = foo(p)` at line 24 moves ownership of $o2$ into `foo1` and, after here, p is no longer used. Before analysis, we remove all `free` statements from programs. SafeMD will generate the patches that can safely deallocate all allocated objects without introducing UAF and DF, as shown in Listing 5. In addition, the C programs fixed by SafeMD still satisfy the exclusive ownership.

SafeMD includes two steps: (1) collect patch candidates for each object by tracking ownership. The main idea of collecting patch candidates is to track ownership of the object and free the object where the owner last used it. (2) Find a correct patch from patch candidates by solving an exact cover problem over the allocated objects. SafeMD analyzes each function individually and each analysis contains the above two steps. Figures 2 and 3 show the analysis of the main and foo1 function, respectively. We only explain the analysis for the `main` function below.

Listing 4. code with memory-leaks.

```c
int *foo1(int *m){
  int *q;
  if(...) {
    q = malloc(1); // o1

  }
  else {
    q = m; // m is no longer used
  }
  ... = *q;
  return q;
}

void foo2(int *k){
  printf("%d", *k);

  return;
}

int main() {
  int *p, *t, *z;
  p = malloc(1); // o2
  if(...) {
    t = foo1(p); // p is no longer used
    ... = *t;

  }
  else {
    z = malloc(1); // o3
    free(p); // is removed befroe analysis
    foo2(z); // z is no longer used
  }
  return 0;
}
```

Listing 5. SafeMD-generated patches.

```c
int *foo1(int *m){
  int *q;
  if(...) {
    q = malloc(1); //o1
    free(m); //+
  }
  else {
    q = m;
  }
  ... = *q;
  return q;
}

void foo2(int *k){
  printf("%d", *k);
  free(k); //+
  return;
}

int main() {
  int *p, *t, *z;
  p = malloc(1); //o2
  if(...) {
    t = foo1(p);
    ... = *t;
    free(t); //+
  }
  else {
    z = malloc(1); //o3
    free(p); //+
    foo2(z);
  }
  return 0;
}
```

Step 1: Collecting Patch Candidates by Ownership Tracking. This analysis step is based on a control flow graph (CGF). The CFG of the main function and analysis results at each node is presented in Figure 2. This analysis maintains owner and patch information for each allocated object as a state of the following form:

$$< o, newOwner, oldOwners, patch, patchNot >$$

where o is a heap object represented by its allocation site, *newOwner* is a pointer who is the unique owner of o, *oldOwners* is a set of pointers that are the old owners of o, *patch* is a set of patches that can safely deallocate the object and *patchNot* is a set of unsafe patches that may introduce UAF and DF. Both *patch* and *patchNot* are denoted by a pair (n, e), which means that an object can be deallocated by inserting a deallocation statement $\texttt{free}(e)$ right after line n, where n is a program point and e is a pointer expression.

For the `main` function, the analysis of SafeMD starts with the function signature. At line 20, because the main function has no parameters, the initial state is marked as empty. The allocation statement at line 22 creates a new tuple $\{\langle o_1, p, \emptyset, \{(22, p)\}, \emptyset\rangle\}$: the allocation site is o_1, its owner is p and the safe patch is $(22, p)$, which indicates that we can safely free o_1 via owner p after line 22. Now, *oldOwners* and *patchNot* are empty.

We first consider the false branch at line 29. The analysis updates the states as follows:

$$\left\{ \begin{array}{l} \langle o_1, p, \emptyset, \{(22, p), (29, p)\}, \emptyset\rangle \\ \langle o_2, z, \emptyset, \{(29, z)\}, \emptyset\rangle \end{array} \right\} \tag{1}$$

A new tuple for the new object o_2 allocated at line 29 is created. For the state of o_1, a new safe patch $(29, p)$ is added into *patch*.

At line 31, the function call updates the states as follows:

$$\left\{ \begin{array}{l} \tau_1 = \langle o_1, p, \emptyset, \{(22, p), (29, p), (31, p)\}, \emptyset\rangle \\ \tau_2 = \langle o_2, \perp, \{z\}, \emptyset, \{(29, z)\}, \{(31, z)\}\rangle \end{array} \right\} \tag{2}$$

Because the object o_2 is used as an argument z in `foo2(z)`, to avoid UAF, in state τ_2, we remove the safe patch $(29, z)$ from the *patch* and add it into *patchNot*. The call `foo2(z)` moves ownership of o_2 into `foo2` via parameter passing; for this, we carry out three changes: (1) we mark *newOwner* with $\perp$ to indicate that the ownership of o_2 is moved. Thus, z becomes the old owner. (2) We reset *patch* to $\emptyset$ to denote that the `main` function has no responsibility to free o_2 since it has lost ownership of o_2. (3) Because the old owner cannot free o_2, a new unsafe patch $(31, z)$, where z is now an old owner of o_2, is generated.

Next, we consider the true branch at line 24, where the function call updates the state $\{\langle o_1, p, \emptyset, \{(22, p)\}, \emptyset\rangle\}$ as follows:

$$\left\{ \begin{array}{l} \langle o_1, \perp, \{p\}, \emptyset, \{(22, p), (24, p)\}\rangle \\ \langle o_3, t, \emptyset, \{(24, t)\}, \emptyset\rangle \end{array} \right\} \tag{3}$$

The ownership of object o_1 is moved into `foo1`, so the update for the state of o_1 is the same as the state of o_2 in (2). Note that `foo1` returns a pointer that points to a valid object, so we create a new state: the allocation site is o_3, its owner is the receiver t and the safe patch is $(24, t)$.

At line 25, the states are updated as follows:

$$\left\{ \begin{array}{l} \tau_3 = \langle o_1, \perp, \{p\}, \emptyset, \{(22, p), (24, p), (25, p)\}\rangle \\ \tau_4 = \langle o_3, t, \emptyset, \{(25, t)\}, \{(24, t)\}\rangle \end{array} \right\} \tag{4}$$

In τ_4, because o_3 is used by $*t$ at line 25, we remove the safe patch $(24, t)$ from the state and declare it as unsafe to avoid UAF. Now, the only safe patch for o_3 is $(25, t)$. In τ_3, a new unsafe patch $(25, p)$ is generated. This is because p is an old owner of o_1 and thus cannot free o_1 after line 25.

At the join point line 33, our analysis maintains each state separately for each different branch. With the states in (2) and (4) as input, the analysis produces the following states as output:

$$\left\{\begin{array}{l} \tau_1' = \langle o_1, p, \emptyset, \{(22,p),(29,p),(31,p)\}, \emptyset\rangle \\ \tau_2' = \langle o_2, \bot, \{z\}, \emptyset, \{(29,z)\}, \{(31,z)\}\rangle \\ \tau_3' = \langle o_1, \bot, \{p\}, \emptyset, \{(22,p),(24,p),(25,p)\}\rangle \\ \tau_4' = \langle o_3, t, \emptyset, \{(25,t)\}, \{(24,t)\}\rangle \end{array}\right\} \tag{5}$$

The return statement returns a value of 0 instead of moving any object's ownership, so the states $\tau_1' \sim \tau_4'$ are the same as $\tau_1 \sim \tau_4$. The analysis finishes with the states in (5).

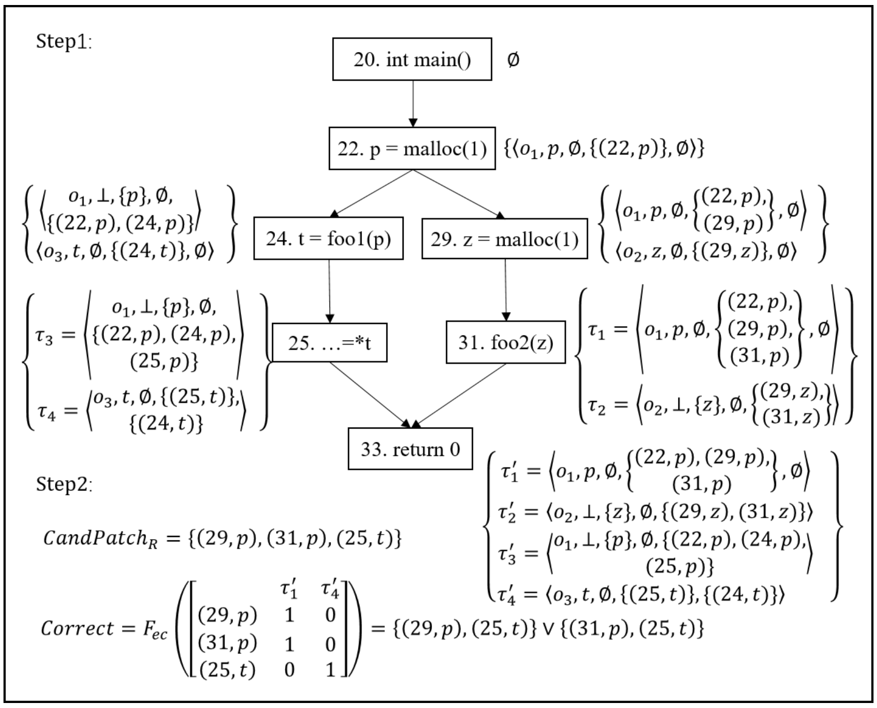
Step1:
20. int main()
∅
22. p = malloc(1)
{⟨o1, p, ∅, {(22, p)}, ∅⟩}
24. t = foo1(p)
29. z = malloc(1)
25. …=*t
31. foo2(z)
33. return 0
Step2:
CandPatch_R = {(29, p), (31, p), (25, t)}
Correct = F_ec(...) = {(29, p), (25, t)} ∨ {(31, p), (25, t)}

Figure 2. SafeMD for main function.

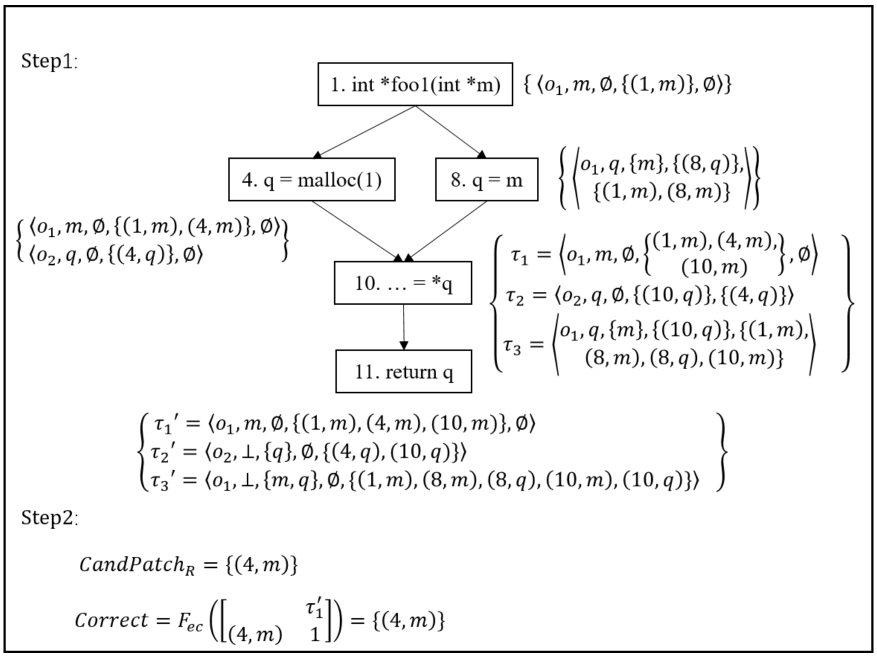

Figure 3. SafeMD for foo1 function.

Step 2: Finding Correct Patches by Solving Exact Cover Problem. After we collect patch candidates in step 1, step 2 is to find correct patches that safely deallocate all objects (i.e., no memory leaks) while not introducing UAF and DF. Finding a correct patch can be reduced to solve an exact cover problem.

First, due to ownership, we collect the valid states whose *new_owner* is not $\perp$. The valid states denote that the current procedure has responsibility to free the objects. From the owner information of states in (5), we obtain the valid states $ValidStates = \{\tau'_1, \tau'_4\}$.

Then, from the patch information of states in (5), we collect the safe patches from *patch* and unsafe patches from *patchNot* from all states:

$$\begin{aligned} Safe &= \{(22,p),(29,p),(31,p),(25,t)\} \\ UnSafe &= \{(29,z),(31,z),(22,p),(24,p),(25,p),(24,t)\}. \end{aligned}$$

Thus, candidate patches are those in $Safe$ but not in $UnSafe$:

$$CandPatch_R = Safe \setminus UnSafe = \{(29,p),(31,p),(25,t)\}$$

The patches in $CandPatch_R$ cannot incur UAF because the patches that may cause UAF are collected in $UnSafe$. However, using all patches in $CandPatch_R$ may cause DF. For example, using both $(29,p)$ and $(31,p)$ will incur double-free for o_1 in the false branch. So, we have to find a subset of the candidate patches that does not introduce DF while deallocating all memory objects. This can be solved by an exact cover problem over valid states, which is represented by the following incidence matrix:

	τ'_1	τ'_4
$(29,p)$	1	0
$(31,p)$	1	0
$(25,t)$	0	1

Each row r in matrix represents a patch in $CandPatch_R$ and each column τ represents a valid state in $ValidStates$. The entry in row r and column τ is 1 if patch r is included in $patch$ of state τ and 0 otherwise. For example, τ'_1 contains $(29,p)$ in $patch$, so the entry in row $(29,p)$ and column τ'_1 is 1. Solving an exact cover problem represented by the above incidence matrix is achieved by the selection of rows such that each column contains only a single 1 among selected rows. In this example, the correct patches are computed as $\{(29,p),(25,t)\} \vee \{(31,p),(25,t)\}$. Both the patches $\{(29,p),(25,t)\}$ and $\{(31,p),(25,t)\}$ cover all states (i.e., no memory leaks) and each state is covered by at most one patch (i.e., no DF). In addition, these two patches satisfy exclusive ownership constraints. Considering that the objects are deallocated as early as possible, we choose $\{(29,p),(25,t)\}$ to free the objects in the `main` function, as shown in Listing 5.

SafeMD will generate the correct patch $(4,m)$ for `foo1`. The four patches generated by SafeMD for the code in Listing 4 can safely fix the memory leaks across functions. MemFix fails to safely fix this code: it will generate `free(t)` and `free(p)` at line 26 in the `main` function, which may introduce DF. Other fixing techniques can safely fix this code, but the patches generated by them may violate exclusive ownership.

5. Approach Details

This section presents the algorithm of SafeMD, which modifies the algorithm of MemFix to serve only C programs that satisfy exclusive ownership constraints. Section 5.1 defines a core language. The two steps of SafeMD are presented in Sections 5.2 and 5.3, respectively.

5.1. Language

For simplicity, we formalize SafeMD on top of a simple pointer language. Let P be an input program for SafeMD. A program P is represented by a CFG $(\mathbb{C}, \hookrightarrow, c_e, c_x)$, where $\mathbb{C}$ denotes the set of program points, $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is the set of flow edges, $c_1 \hookrightarrow c_2$ indicates that there is a possible flow of execution from c_1 to c_2 and c_e and c_x are the unique entry and exit nodes of P. A program point $c \in \mathbb{C}$ is associated with a command, denoted $cmd(c)$, as defined by the following grammar:

$$\begin{aligned} cmd \rightarrow\ & \text{alloc}(p) \mid \text{assign}(p,e) \mid \text{return}(p) \mid \text{funDef}(fname, par\_list, par\_t\_list, ret\_t) \\ & \mid \text{call}(fname, arg\_list, arg\_t\_list, receiver) \\ p \rightarrow\ & x \mid *x \mid \text{null} \\ e, par\_list, arg\_list, receiver \rightarrow\ & p \mid \text{none} \\ par\_t\_list, ret\_t, arg\_t\_list \rightarrow\ & type \\ type \rightarrow\ & \text{void} \mid \text{int} \mid ... \mid \text{struct} \mid ... \mid \text{type}* \mid \text{type[ ]} \end{aligned}$$

A pointer expression p can be a variable x, a pointer dereference $*x$ or a NULL pointer. Allocation command alloc(p) creates a new memory object pointed to by p. Assignment command assign(p,e) assigns the expression e to p. The command funDef$(fname, par_list, par_t_list, ret_t)$ describes a function signature, where $fname$, par_list, par_t_list and ret_t denote the function name, parameter list, parameter type list and return type, respectively. The command call$(fname, arg_list, arg_t_list, receiver)$ describes the call on function $fname$, where arg_list, arg_t_list and $receiver$ represent the argument list, argument type list and a variable who receives the return value, respectively. “none” means empty; for example, if arg_list, arg_t_list and $receiver$ equal none, it represents a function call with no arguments and return values. The deallocation statements free(p) are ignored because we remove them from the program P before analysis. In addition, the program P must also satisfy exclusive ownership.

5.2. Step 1: Collecting Patch Candidates by Ownership Tracking

The first step of SafeMD is to statically analyze the procedure to collect patch candidates. The idea of this step is to track ownership of the object and free the object where the owner last used it.

5.2.1. Abstract Domain

The abstract domain of the analysis is defined as follows:

$$\begin{aligned} A\ & \in \mathbb{D} = \mathbb{C} \rightarrow \mathcal{P}(State) \\ s\ & \in State = AllocSite \times NewOwner \times OldOwner \times Patch \times PatchNot \\ o\ & \in AllocSite \subseteq \mathbb{C} \\ newOwner\ & \in NewOwner = AP \\ oldOwners\ & \in OldOwner = \mathcal{P}(AP) \\ patch\ & \in Patch = \mathcal{P}(\mathbb{C} \times AP) \\ patchNot\ & \in PatchNot = \mathcal{P}(\mathbb{C} \times AP) \\ p\ & \in AP = \{x, *x, \text{null} \mid x \in Var\} \cup \{\bot\} \end{aligned}$$

Var is the finite set of program variables in P. $AllocSite \subseteq \mathbb{C}$ is the finite set of allocation sites in P, i.e., the nodes whose associated commands are alloc(p). A domain element $A \in \mathbb{D}$ is a finite map that maps each program point to a set of reachable states. A state $s = \langle o, newOwner, oldOwners, patch, patchNot \rangle$ describes an abstract object with owner and patch information, where $o \in AllocSite$ is the allocation site of the object, $newOwner \in AP$ is an access path that points to the object via a unique owner, $oldOwners \subseteq AP$ is a set of access paths that points to the object via old owners, $patch \subseteq \mathbb{C} \times AP$ is a set of patches that can safely deallocate the object and $patchNot \subseteq \mathbb{C} \times AP$ is a set of unsafe patches. AP denotes the set of access paths that can be generated for the given program P. For the language in Section 5.1, AP equals the set of pointer expression p, i.e., AP can be a variable x, a pointer dereference $*x$, a NULL pointer or an invalid access path $\bot$. Each element in $patch$ and $patchNot$ is a pair $(c,p) \in \mathbb{C} \times AP$ that consists of a program point c and an access path p. A patch (c,p) represents a `free`(p) statement that can be inserted right after the program point c.

5.2.2. Abstract Semantics

SafeMD collects patch candidates for each function individually. For each function, the analysis starts from the program point c where the function is defined, i.e., the node whose command is funDef($fname, par\_list, par\_t\_list, ret\_t$), and computes an initial set S_0 that consists of initial states. S_0 is defined as follows:

$$S_0 = \begin{cases} \bigcup\limits_{par \in par\_list} \gamma_c(par) & \text{if } par\_list \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\gamma_c(par) = \begin{cases} \langle c, par, \emptyset, \{(c, par)\}, \emptyset \rangle & \text{if } isPointer(par) = true \\ \emptyset & \text{otherwise.} \end{cases}$$

If the function $fname$ has no parameter, S_0 is empty; otherwise, it is computed by $\gamma_c : Var \to State$ which generates an initial state for a parameter par of $fname$. The initial states for all parameters form S_0. For the computation of γ_c, if a parameter par points to a heap object (i.e., the function $isPointer(x)$ equals true), a new state is created since the ownership of objects can be transferred by parameter passing: the allocation site of the object is the program point c, the owner of the object is par and the safe patch is (c, par). For example, in Figure 3, S_0 for `foo1` definition at line 1 only contains a state, i.e., $< o_1, m, \emptyset, \{(1, m)\}, \emptyset >$.

Start with S_0: SafeMD updates the states at each node based on the command associated with that node until reaching the exit node. In other words, the analysis computes a least fixed point $\mathrm{lf_p} F \in \mathbb{D}$ 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 $f_c : \mathcal{P}(State) \to \mathcal{P}(State)$ is the transfer function at a program point c:

$$f_c(S) = \begin{cases} S' \cup \{s_{new}\} & \text{if } cmd(c) = \text{alloc}(p) \\ S' & \text{if } cmd(c) = \text{assign}(p, e) \\ S' & \text{if } cmd(c) = \text{return}(p) \\ S' \cup \{s_{new}\} & \text{if } cmd(c) = \text{call}(\_, \_, \_, p) \wedge isPointer(p) = true \\ S' & \text{if } cmd(c) = \text{call}(\_, \_, \_, \text{none}). \end{cases}$$

where $s_{new} = \langle c, p, \emptyset, \{(c, p)\}, \emptyset \rangle$. f_c updates the states in S according to different commands. For alloc(p), f_c not only updates the existing states in S to S' but also creates a new state s_{new}. Similarly, for the command call($\_, \_, \_, p$), where p is a pointer, a new state for the object pointed by p is created.

The set S' is updated by two transfer functions ϕ_c and φ_c: $S' = \bigcup\limits_{s \in S} (\varphi_c \circ \phi_c)(s)$. For a state s, we first update owner information by $\phi_c : State \to State$ and then patch information by $\varphi_c : State \to State$. Next, we define ϕ_c and φ_c for different commands.

Given a state s at the program point c,

$$s = \langle o, newOwner, oldOwners, patch, patchNot \rangle$$

(1) When $cmd(c)$ = alloc(p), ϕ_c makes $newOwner$ and $oldOwners$ unchanged:

$$\phi_c(s) = \langle o, newOwner', oldOwners', patch, patchNot \rangle$$
$$newOwner' = newOwner, oldOwners' = oldOwners.$$

Then, φ_c updates $patch$ and $patchNot$ as follows:

$$\varphi_c(s) = \langle o, newOwner, oldOwners, patch', patchNot' \rangle$$
$$patchNot' = patchNot \cup GO, patch' = (patch \cup GN) \setminus patchNot'.$$

where $GN = \{(c, q) \mid q = newOwner\}$, $GO = \{(c, q) \mid q \in oldOwners\}$. GN contains a safe patch that is newly generated at c via a unique owner. GO is the set of unsafe patches that are newly generated at c via old owners, because an object cannot be deallocated via old owners. Also, we exclude $patchNot'$ from $patch'$ to ensure that $patch'$ and $patchNot'$ are disjoint.

(2) When $cmd(c)$ = assign(p, e), ϕ_c updates *newOwner* and *oldOwners* as follows:

$$\phi_c(s) = \langle o, newOwner', oldOwners', patch, patchNot\rangle$$
$$newOwner' = \begin{cases} p & \text{if } e = newOwner \\ newOwner & otherwise. \end{cases}$$
$$oldOwners' = \begin{cases} oldOwners \cup \{e\} \setminus newOwner' & \text{if } e = newOwner \\ oldOwners & otherwise. \end{cases}$$

An assignment `p = e` can transfer the ownership of object o from e to p, making p and e become the new owner and old owner, respectively. We also exclude $newOwner'$ from $oldOwners'$ to ensure that $newOwner'$ and $oldOwners'$ are disjoint.

Then, φ_c updates *patch* and *patchNot* as follows:

$$\varphi_c(s) = \langle o, newOwner, oldOwners, patch', patchNot'\rangle$$
$$patchNot' = \begin{cases} patchNot \cup patch \cup GO & \text{if } o \text{ is used at } c \\ patchNot \cup GO & \text{otherwise.} \end{cases}$$
$$patch' = \begin{cases} GN \setminus patchNot' & \text{if } o \text{ is used at } c \\ (patch \cup GN) \setminus patchNot' & \text{otherwise.} \end{cases}$$

The condition "o is used at c" contains two cases. In the first case `p = e`, where o is used by expression e and transfers the ownership of o from e to p, the only safe patch is (c, p) generated by GN. For $patchNot'$, it adds a set of new unsafe patches related to old owners that contains *patch* and GO. The pointer assignment `q = m` at line 7 in Figure 3 is such an example.

Consider the second case `p = e`, where o is used by expression e but does not transfer the ownership of o: to prevent UAF, the safe patches are removed from *patch* and added to *patchNot*, and the only safe patch is generated by GN. Also, GO is included in $patchNot'$. For example, the assignment `... = *q` in Figure 3 uses the object pointed to by q via dereference expression $*q$, but it does not transfer the ownership of the object.

For the *otherwise* case, where o is not used at c, we only merge new safe patches generated by GN with *patch* and unsafe patches generated by GO with *patchNot*.

(3) When $cmd(c)$ = return(p), ϕ_c updates *newOwner* and *oldOwners* as follows:

$$\phi_c(s) = \langle o, newOwner', oldOwners', patch, patchNot\rangle$$
$$oldOwners' = \begin{cases} (oldOwners \cup \{p\}) \setminus newOwner' & \text{if } p = newOwner \\ oldOwners & otherwise. \end{cases}$$
$$newOwner' = \begin{cases} \perp & \text{if } p = newOwner \\ newOwner & \text{otherwise.} \end{cases}$$

The ownership of objects can be transferred by return values. If return value p equals the owner of object o in state s (i.e., $p = newOwner$), then the ownership of o is moved out from the callee and p becomes the old owner. We use the symbol $\perp$ to indicate that the ownership of o is moved.

Then, φ_c updates *patch* and *patchNot* as follows:

$$\varphi_c(s) = \langle o, newOwner, oldOwners, patch', patchNot'\rangle$$
$$patchNot' = \begin{cases} patchNot \cup patch & \text{if } newOwner = \perp \\ patchNot & \text{otherwise.} \end{cases} \quad patch' = \begin{cases} \emptyset & \text{if } newOwner = \perp \\ patch & \text{otherwise.} \end{cases}$$

If *newOwner* of object o is updated to $\perp$ by ϕ_c, the safe patches in *patch* become unsafe because o is returned to the caller. We reset *patch* to $\emptyset$ to indicate that the callee cannot free o since it has lost ownership of o. The responsibility for deallocating o falls on the caller. Consider the `foo1` in Figure 3: SafeMD will merge the states computed from the if-else branch as follows:

$$\left\{ \begin{array}{l} \tau_1 = \langle o_1, m, \emptyset, \{(1,m), (4,m), (10,m)\}, \emptyset\rangle \\ \tau_2 = \langle o_2, q, \emptyset, \{(10,q)\}, \{(4,q)\}\rangle \\ \tau_3 = \langle o_1, q, \{m\}, \{(10,q)\}, \{(1,m), (8,m), (8,q), (10,m)\}\rangle \end{array} \right\}$$

For the objects in state τ_2 and τ_3, because their ownership is moved out from `foo1` by return statement "`return q`", their *newOwner* equals $\perp$ and q becomes the old owner. Also, their *patch* is reset to $\emptyset$. For state τ_1, it remains unchanged. The update result is shown below.

$$\left\{ \begin{array}{l} \tau_1 = \langle o_1, m, \emptyset, \{(1,m), (4,m), (10,m)\}, \emptyset\rangle \\ \tau_2' = \langle o_2, \perp, \{q\}, \emptyset, \{(4,q), (10,q)\}\rangle \\ \tau_3' = \langle o_1, \perp, \{m,q\}, \emptyset, \{(8,q), (1,m), (8,m), (10,m), (10,q)\}\rangle \end{array} \right\}$$

(4) When *cmd*(c) = call($fname$,*arg_list*,*arg_t_list*, *receiver*), ϕ_c updates *newOwner* and *oldOwners* as follows:

$$\phi_c(s) = \langle o, newOwner', oldOwners', patch, patchNot\rangle$$

$$oldOwners' = \begin{cases} (oldOwners \cup \{newOwner\}) \setminus newOwner' & \text{if } newOwner \in arg\_list \\ oldOwners & \text{otherwise.} \end{cases}$$

$$newOwner' = \begin{cases} \perp & \text{if } newOwner \in arg\_list \\ newOwner & \text{otherwise.} \end{cases}$$

The ownership of objects can be transferred to the callee by parameter passing. If *newOwner* of o is passed to $fname$ as an argument, i.e., $newOwner \in arg\_list$, then *newOwner* becomes the old owner. We mark *newOwner* with $\perp$ to indicate that the ownership of o is moved.

Then, φ_c updates *patch* and *patchNot* as follows:

$$\varphi_c(s) = \langle o, newOwner, oldOwners, patch', patchNot'\rangle$$

$$patchNot' = \begin{cases} patchNot \cup patch \cup GO & \text{if } newOwner = \perp \\ patchNot \cup patch \cup GO & \text{if } newOwner \neq \perp \text{ but } o \text{ is used in } arg\_list \\ patchNot \cup GO & \text{otherwise.} \end{cases}$$

$$patch' = \begin{cases} \emptyset & \text{if } newOwner = \perp \\ GN \setminus patchNot' & \text{if } newOwner \neq \perp \text{ but } o \text{ is used in } arg\_list \\ \{patch \cup GN\} \setminus patchNot' & \text{otherwise.} \end{cases}$$

We discuss three cases. The first case is *newOwner* $=\perp$, which indicates that the ownership of object is moved. For the *patchNot*$'$, it adds a set of new unsafe patches related to old owners that contains *patch* and *GO*. We reset the *patch* to $\emptyset$ to denote that the caller is not responsible for deallocating the object since the ownership of the object is moved to the callee.

The second case is "*newOwner* $\neq\perp$ but o is used at c", which indicates that the ownership of object o is not transferred but o is used (e.g., by pointer dereference) in arguments. Because o is used, the safe patches in *patch* are added to *patchNot* to avoid UAF, and the only safe patch is generated by *GN*. For example, function call `foo(*p)`, where p is a pointer that points to a valid object, belongs to this case.

For the *otherwise* case, where the ownership of object o is not transferred and o is also not used, we merge new safe patches generated by *GN* with *patch* and unsafe patches generated by *GO* with *patchNot*.

5.3. Step 2: Finding Correct Patches by Solving Exact Cover Problem

Once the owner and patch information for each object are collected in step 1, the second step of SafeMD is to find the correct patches that can safely deallocate all allocated objects (no ML) while not introducing UAF and DF. MemFix found correct patches by solving an exact cover problem. We use this method but modify it because ownership is considered.

Let $R = (\text{lf}_\text{p}F)(c_x) \subseteq State$ be the set of reachable states computed by step 1 at the exit node of the program. We first give the definition of candidate correct patches and valid states from R.

Definition 1 (Candidate Correct Patches)**.** *The set of candidate correct patches collects the possible safe patches for each object, which are defined as follows:*

$$\begin{aligned} Safe &= \bigcup\{patch \mid \langle\_,\_,\_, patch, \_\rangle \in R\}. \\ UnSafe &= \bigcup\{patchNot \mid \langle\_,\_,\_,\_, patchNot\rangle \in R\}. \\ CandPatch_R &= Safe \setminus UnSafe. \end{aligned}$$

$Safe$ collects the patches that can safely deallocate an object, and $UnSafe$ collects the unsafe patches that can cause UAF and DF (caused by pointer aliasing). We exclude $UnSafe$ from $Safe$ to obtain the set $CandPatch_R$, which is used to find correct patches.

The patches in $CandPatch_R$ cannot cause UAF since these unsafe patches are all collected in $UnSafe$ and thus already excluded from $CandPatch_R$. In addition, the patches in $CandPatch_R$ also satisfy exclusive ownership constraints; that is, old owners cannot be used to deallocate objects. After all, the C programs satisfying exclusive ownership are more safe than normal C programs. However, the patches in $CandPatch_R$ may cause DF. Thanks to ownership, plenty of unsafe patches caused by pointer aliasing are excluded from $CandPatch_R$, since SafeMD collects the unsafe patches related to old owners (aliases) in $UnSafe$. This can noticeably reduce the search space for finding the correct patches as we avoid verifying patch combinations containing pointer aliasing. However, $CandPatch_R$ cannot exclude DF caused by freeing the memory multiple times using the same pointer. For example, if an object can be safely deallocated at line 4 and line 5, then $(3, p)$ and $(4, p)$ may be in $CandPatch_R$, and using both of them will incur DF. Therefore, this step aims to find the correct patches that do not cause such a kind of DF.

Definition 2 (Valid States)**.** *The valid states indicate that the ownership of objects has not been moved and thus the objects should be deallocated in the current function, which is defined as follows:*

$$ValidStates = \bigcup\{\langle\_, newOwner, \_,\_,\_\rangle \in R \mid newOwner \neq \perp\}.$$

Next, we present the definition of the problem for finding the correct patches, which can be reduced into solving an exact cover problem over valid states.

Definition 3 (The Problem of Finding Correct Patches)**.** *Let $M : CandPatch_R \to \mathcal{P}(ValidStates)$ be the function from candidate correct patches to the valid states that can be safely deallocated by the corresponding patches:*

$$M(r) = \{<\_,\_,\_, patch, \_> \in ValidStates \mid r \in patch\}.$$

From M, find a subset $Correct \subseteq CandPatch_R$ such that

- *$ValidStates = \bigcup_{r \in Correct} M(r)$, which means that Correct covers all valid states;*
- *$M(r_1) \bigcap M(r_2) = \emptyset$ for all $r_1, r_2 \in Correct$, which means that the chosen subsets in $M(r)$ (where $r \in Correct$) are pairwise disjoint.*

M describes the incidence matrix in Figures 2 and 3. The first condition means that all allocated objects must be deallocated, which guarantees the absence of memory leaks. The second condition means that every allocated object is deallocated no more than once, which guarantees the absence of DF. Recall that UAF is avoided in $CandPatch_R$.

6. Evaluation

We evaluated SafeMD with two different benchmark sets and compared it with MemFix, a static-based approach for fixing memory leaks in C/C++ programs. Our experiments were performed on a PC with Intel Core i7-7700 CPU (3.60 GHZ) and 8 GB RAM running 64-bit Ubuntu 18.04.3 LTS.

6.1. Implementation

We implemented SafeMD as a stand-alone tool [35]. The framework of SafeMD is shown in Figure 4. We first make use of the open-source code analysis platform for C/C++ based on code property graphs, Joern [36], to extract CFGs for all functions in our benchmarks. Then, all CFGs (dot files) are loaded into NetworkX for graph traversal to collect patch candidates (Section 5.2) and calculate correct patches (Section 5.3). Our implementation supports the C standard memory allocators `malloc` and `calloc` except for `realloc`, since `realloc` may be fixed safely by adding conditional statements, which is beyond the scope of the current algorithm of SafeMD.

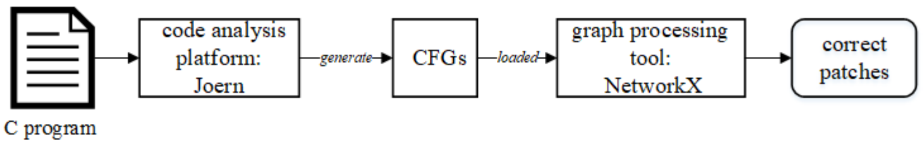
code analysis platform: Joern
generate
CFGs
loaded
graph processing tool: NetworkX
correct patches
C program

Figure 4. The framework of SafeMD.

In step 1, recall that, in the generation of initial states S_0 mentioned in Section 5.2.2, a new state is created if the parameter points to a heap object. Although SafeMD analyzes each function individually, to improve the accuracy of S_0, SafeMD starts from the `main` function and proceeds according to the call graph, and sets a flag to guide the generation of S_0 for each callee function. In step 2, the exact cover problem is NP-complete. Our implementation uses existing DFS-based search algorithm to solve the exact cover problem. This algorithm takes optimization strategies to improve the search speed for the exact cover problem. Also, our algorithm of finding correct patches will not find all solutions; instead, it will return the first solution that it finds.

6.2. Benchmark

We use two benchmarks to evaluate SafeMD. In Table 1, the first benchmark is relevant to memory leak (CWE-401) in Juliet Test Suite (JTS) for C, and “int_malloc” and “twoIntsStruct_malloc” mean a memory object pointed to by an integer pointer and struct pointer leaks, respectively. The second benchmark has 26 model programs with memory leaks. These programs are selected from 50 test programs that are provided by [10], who constructed them from five GitHub open-source C repositories.

Table 1. Evaluation results on CWE-401 and open-source C repositories.

Benchmark	Program	#Loc	#Function	SafeMD/MemFix/#Pgm.	#Time (s)
CWE-401	int_malloc	82	10	36/36/38	<1.0
	twoIntsStruct_malloc	92	9	36/36/38	<1.0
Total				72/72/76	
Open-Source C Repo.	Binutils	127	6	4/4/5	<1.0
	Git	150	5	2/3/5	<1.0
	OpenSSH	150	4	5/2/6	<1.0
	OpenSSL	134	3	3/1/4	<1.0
	Tmux	154	6	4/3/6	<1.0
Total				18/13/26	

To evaluate SafeMD, the programs in these benchmarks are first modified to satisfy exclusive ownership constraints. For the C programs that are difficult to modify, they have been removed from the benchmarks, leaving a total of 102 programs (76 in CWE-401 and 26 in open-source C repositories). Note that the modifications in benchmarks do not guarantee semantics preservation since, in this experiment, we aim to provide the test programs that satisfy exclusive ownership constraints, and how to rewrite C programs to satisfy exclusive ownership is beyond the scope of this paper.

In our modifications for C programs in these benchmarks, we only focus on pointers returned by dynamic memory allocation functions and modify these pointers to make their use satisfy ownership constraints. For a better understanding of the experimental results, we briefly list the main code features of C programs that satisfy exclusive ownership below.

- **Assignments.** For the pointer assignments, such as `q = p;`, if variable p points to a heap object, then p cannot be used after this assignment until it is re-assigned. Particularly, for the assignment of compound data type (e.g., struct), such as `q = p.f;`, the struct itself p cannot be used after assignment, but other members of struct p are still available.
- **Function calls.** For the calls that call user-defined functions, such as `foo(p)`, if variable p points to a heap object, then, in the caller, p cannot be used after the call site. For the library functions (except for standard allocation and deallocation functions), such as `memcpy(p,...)`, because we cannot modify the code of library functions, we assume that the ownership of p does not move into library functions, and thus p is still available in the caller.

6.3. Results

Table 1 shows the results on JTS and open-source C repositories. #Loc represents the average number of lines of code (after modification). SafeMD analyzes each function individually, so we count the number of user-defined functions in all test programs and use #Function to list the average number of functions that the programs have. SafeMD/MemFix/#Pgm represents the number of test cases that can be fixed by SafeMD and MemFix. #Time reports the maximum execution time performed by SafeMD or MemFix. For test cases in CWE-401, because these programs have relatively simple structures and data types, they can easily be modified to satisfy exclusive ownership while preserving semantics. Both SafeMD and MemFix can fix a total of 72 programs (out of 76 programs) with an accuracy of 95%. Four test programs including function pointers are not supported by SafeMD and MemFix. The maximum execution time for fixing these test cases is smaller than 1.0 s.

For open-source C repositories, SafeMD and MemFix can fix a total of 18 programs (out of 26 programs) with an accuracy of 69% and 13 programs (out of 26 programs) with an accuracy of 50%, respectively. We manually look at these programs to investigate the shortcomings of SafeMD. We consider the following four scenarios (S1–S4):

S1. Memory leak fixed by both SafeMD and Memfix. For these test cases (e.g., Binutils), we find that the allocated and leaked object is largely limited in scope within a procedure and has limited leaked paths. The code snippet from binutils in Listing 6 shows the leak pattern that can be fixed by both SafeMD and Memfix.

Listing 6. Memory leak pattern fixed by both SafeMD and Memfix.

```
1  char *p = malloc(10); //o1
2  if(argv == NULL) {
3      goto FAIL; //error path
4  }
5  else{
6     *p = 'a'; //Non-error path
7     free(p); //both SafeMD and Memfix can generate this patches
8     return 0;
9  FALL:
10    free(p); //both SafeMD and Memfix can generate this patches
11    exit(1);
12 }
```

In the above code snippet that satisfies exclusive ownership, the allocated and leaked object $o1$ is limited in the current procedure. An error path refers to a program path where an abrupt return happens under abnormal situations. The object $o1$ allocated at line 1 is leaked on both the error path (when goto FAIL is executed on line 3) and the non-error path, and both SafeMD and Memfix can safely fix such memory leaks by generating patches for the error path and non-error path. However, the patches generated by Memfix may violate exclusive ownership.

S2. Memory leak fixed by SafeMD only. For these test cases, we find that the leaked object is transferred to another procedure. The code snippet in Listing 7 shows the leak pattern that can be fixed only by SafeMD.

Listing 7. Memory leak pattern fixed by SafeMD only.

```
1   int *f(int *m, int flag){
2     int *q;
3     if(flag == 1) {
4       q = malloc(1); //o1
5       // SafeMD: free(m);
6     }
7     else {
8       q = m;
9     }
10    return q;
11  }
12
13  int g(){
14    int *p, *q;
15    int flag = 0;
16    p = malloc(1); //o2
17    q = f(p, flag); // p is no longer used after here
18    // MemFix/SafeMD: free(q);
19    // MemFix: (double-free) free(p);
20  }
```

In the above code snippet that satisfies exclusive ownership, MemFix will perform interprocedural analysis and insert `free(q)` and `free(p)` at line 18 and 19 to free memory objects, but it may introduce DF for object $o2$ (at line 19) when the else branch is true in the `f` function. In this case, to safely fix memory leaks, the correct patch `if(flag == 1) free(q);` is required at line 18. But, MemFix fails to fix because it is unable to generate patches with conditional statements. However, SafeMD can safely fix these memory leaks without combining conditional expressions, since it tracks the owner of the leaked object and frees the leaked object from its owner. Specifically, SafeMD analyzes function `f` and `g` individually, and generates `free(m)` and `free(q)` at line 5 and 18, respectively. The function call at line 17 transfers the ownership of $o2$ from p to m; therefore, SafeMD concludes that `f` is responsible for freeing the object $o2$ if the ownership of $o2$ is not returned by `f`; that is, SafeMD will generate `free(m)` in the true branch of `f` since this execution path does not return the ownership of $o2$ to `g`. Because this code satisfies exclusive ownership, where p is no longer used after line 17, the patch `free(m)` is safe. On the contrary, in `f`, the execution path where the false branch is taken returns the ownership of $o2$ to `g`; therefore, SafeMD concludes that `g` is responsible for freeing the object $o2$ and thus generates `free(q)` at line 18. Compared with SafeMD, MemFix tries to free $o2$ in the `g` function, but it fails since it cannot generate conditional patches.

S3. Memory leak fixed by Memfix only. These are cases that cannot be fixed by SafeMD primarily due to the ownership limitations of SafeMD, which means that the strictness of ownership enforced in C makes it unsafe for SafeMD to free objects in some situations.

One situation that can be fixed by Memfix but not by SafeMD is when ownership constraints are enforced on operations related to arrays and linked lists. Consider the Listing 8's code pattern found in Git. Line 3 creates an object $o1$. This code satisfies the ownership constraints since the old owner $msg2$ is no longer used after assignment `ptr2 = msg2`. SafeMD will generate `free(ptr2)` at line 10, leading to an invalid free because $ptr2$ does not point to the start address of $o1$. However, Memfix does not enforce that the patches satisfy the ownership constraints, so it will generate `free(msg2)` at line 10, which is a safe patch. One way of addressing this problem for SafeMD is to modify this code (which is beyond the scope of this paper); for example, we can modify the assignment `ptr2 = msg2` to `ptr2 = &msg2`, where the ownership is not transferred to $ptr2$; thus, SafeMD can generate the safe patch `free(msg2)` at line 10.

Listing 8. Memory leak pattern fixed by Memfix only.

```
size_t screen(...){
  char *msg2;
  msg2 = xmalloc(...);
  ptr2 = msg2; // ownership transfer
  while (...) {
  ptr2++;
  }
  *ptr2 = '\0';
  // SafeMD: free(ptr2); Memfix: free(msg2)
  // invalid free for SafeMD; safely fixing for Memfix
}
```

S4. Memory leak not fixed by both SafeMD and Memfix. For these test cases, they are related to reallocated memory and function pointers, and SafeMD and Memfix cannot deal with these features. For example, the test cases in Git often store memory objects using reallocation, which makes the portion of successful fixing relatively low.

To sum up, our comparison demonstrates that SafeMD can fix more memory-leak patterns than Memfix (see S2) when the input C programs satisfy exclusive ownership. Also, the patches generated by SafeMD make the input C programs still satisfy exclusive ownership (see S1). However, because of the exclusive ownership satisfied by C programs, SafeMD can lead to invalid frees on the array (see S3).

7. Conclusions

We propose SafeMD, an ownership-based memory deallocation for C programs that satisfy exclusive ownership. Benefiting from ownership, SafeMD obviates alias and interprocedural analysis during collecting patch candidates. Also, the patches generated by SafeMD make the input C programs still satisfy exclusive ownership. Our experiment shows the effectiveness of SafeMD in fixing memory leaks of real-world C programs. It also shows that the ownership system of Rust can be used to guarantee the memory safety of C language. As future work, we plan to consider more memory allocators and relax ownership constraints according to the semantics of C language (e.g., array traversal) to make SafeMD more practical (ease the problem of S3 mentioned in Section 6.3).

Author Contributions: Methodology, X.Y.; validation, Z.H. and G.S.; data curation, S.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original data presented in this study are openly available in https://bitbucket.org/yxhnuaa/safemd/src/master/ (accessed on 10 October 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Chen, Z.; Gu, Y.; Huang, Z.Q.; Zheng, J.; Liu, C.; Liu, Z.Y. Model checking aircraft controller software: A case study. *Softw. Pract. Exp.* **2015**, *45*, 989–1017. [CrossRef]
2. Wang, W.W. MLEE:Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels. In Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC, Vancouver, BC, Canada, 14–16 July 2021.
3. CVE-CVE-2022–27819. MITRE Corporation. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-27819 (accessed on 10 October 2024).
4. CVE-CVE-2017–1081. MITRE Corporation. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1081 (accessed on 10 October 2024).
5. Rust. Rust Team. Available online: https://www.rust-lang.org/ (accessed on 10 October 2024).
6. Yin, X.H.; Huang, Z.Q.; Kan, S.L.; Shen, G.H.; Liu, Y.; Wang, F. SafeOSL: Ensuring memory safety of C via ownership-based intermediate language. *Softw. Pract. Exp.* **2022**, *52*, 1114–1142. [CrossRef]
7. Gao, Q.; Xiong, Y.F.; Mi, Y.H.; Zhang, L.; Yang, W.K.; Zhou, A.P.; Xie, B.; Mei, H. Safe Memory-Leak Fixing for C Programs. In Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1.
8. Yan, H.; Sui, Y.L.; Chen, S.P.; Xue, J.L. Automated memory leak fixing on value-flow slices for C programs. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4–8 April 2016.

9. Tonder, R.J.; Goues, C.L. Static automated program repair for heap properties. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018.
10. Lee, J.; Hong, S.H.; Oh, H.J. MemFix: Static analysis-based repair of memory deallocation errors for C. In Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018.
11. Hong, S.H.; Lee, J.; Lee, J.S.; Oh, H.J. SAVER: Scalable, precise, and safe memory-error repair. In Proceedings of the 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June–19 July 2020.
12. Juliet Test Suite 1.2. NSA Center for Assured Software. Available online: https://samate.nist.gov/SRD/view.php?tsID=86 (accessed on 10 October 2024).
13. Smaragdakis, Y.; Kastrinis, G. Pointer Analysis. *Found. Trends Program. Lang.* **2015**, *2*, 1–69. [CrossRef]
14. Balatsouras, G.; Ferles, K.; Kastrinis, G.; Smaragdakis, Y. A Datalog model of must-alias analysis. In Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis, Barcelona, Spain, 18 June 2017.
15. Yu, B.; Tian, C.; Zhang, N.; Duan, Z.; Du, H. A dynamic approach to detecting, eliminating and fixing memory leaks. *J. Comb. Optim.* **2021**, *42*, 409–426. [CrossRef]
16. Clause, J.; Orso, A. LEAKPOINT: Pinpointing the causes of memory leaks. In Proceedings of the 32nd International Conference on Software Engineering, Cape Town, South Africa, 1–8 May 2010.
17. Murali, A.; Alfadel, M.; Nagappan, M.; Xu, M.; Sun, C.N. AddressWatcher: Sanitizer-Based Localization of Memory Leak Fixes. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2398–2411. [CrossRef]
18. Clarke, D.G.; Potter, J.; Noble, J. Ownership Types for Flexible Alias Protection. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Vancouver, BC, Canada, 18–22 October 1998.
19. Clarke, D.G.; Ostlund, J.; Sergey, I.; Wrigstad, T. Ownership types: A survey. Aliasing in object-oriented programming. *Types Anal. Verif.* **2013**, *7850*, 15–58.
20. Boyapati, C.; Lee, R.; Rinard, M.C. Ownership types for safe programming: Preventing data races and deadlocks. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Seattle, WA, USA, 4–8 November 2002.
21. Kloos, J.; Majumdar, R.; Vafeiadis, V. Asynchronous Liquid Separation Types. In Proceedings of the 29th European Conference on Object-Oriented Programming, Prague, Czech Republic, 5–10 July 2015; Volume 37.
22. Heine, D.L.; Lam, M.S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, USA, 9–11 June 2003.
23. Swamy, N.; Hicks, M.W.; Morrisett, G.; Grossman, D.; Jim, T. Safe manual memory management in Cyclone. *Sci. Comput. Program.* **2006**, *62*, 122–144. [CrossRef]
24. Suenaga, K.; Kobayashi, N. Fractional Ownerships for Safe Memory Deallocation. In Proceedings of the 7th Asian Symposium on Programming Languages and Systems, Seoul, Republic of Korea, 14–16 December 2009; Volume 5904.
25. Sonobe, T.; Suenaga, K.; Igarashi, A. Automatic Memory Management Based on Program Transformation Using Ownership. In Proceedings of the 12th Asian Symposium on Programming Languages and Systems, Singapore, 17–19 November 2014; Volume 8858.
26. Toman, J.; Pernsteiner, S.; Torlak, E. Crust: A Bounded Verifier for Rust (N). In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, Lincoln, NE, USA, 9–13 November 2015.
27. Astrauskas, V.; Bílý, A.; Fiala, J.; Grannan, Z.; Zhang, M.; Matheja, C.; Müller, P.; Poli, F.; Summers, A.J. The Prusti Project: Formal Verification for Rust. In Proceedings of the 14th International Symposium on NASA Formal Methods, Pasadena, CA, USA, 24–27 May 2022; Volume 13260.
28. Lattuada, A.; Hance, T.; Cho, C.; Brun, M.; Subasinghe, I.; Zhou, Y.; Howell, J.; Parno, B.; Hawblitzel, C. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* **2023**, *7*, 286–315. [CrossRef]
29. Qin, B.Q.; Chen, Y.L.; Yu, Z.M.; Song, L.H.; Zhang, Y.Y. Understanding memory and thread safety practices and issues in real-world Rust programs. In Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, London, UK, 15–20 June 2020.
30. Xu, H.; Chen, Z.B.; Sun, M.S.; Zhou, Y.F.; Lyu, M.R. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *Proc. ACM Program. Lang.* **2022**, *31*, 1–25. [CrossRef]
31. Emre, M.; Schroeder, R.; Dewey, K.; Hardekopf, B. Translating C to safer Rust. *Proc. ACM Program. Lang.* **2021**, *5*, 1–29. [CrossRef]
32. Emre, M.; Boyland, P.; Parekh, A.; Schroeder, R.; Dewey, K.; Hardekopf, B. Aliasing Limits on Translating C to Safe Rust. *Proc. ACM Program. Lang.* **2023**, *7*, 551–579. [CrossRef]
33. Zhang, H.L.; David, C.; Yu, Y.J.; Wang, M. Ownership Guided C to Rust Translation. In Proceedings of the 35th International Conference on Computer Aided Verification, Paris, France, 17–22 July 2023; Volume 13966.
34. Jung, R.; Jourdan, J.H.; Krebbers, R.; Dreyer, D. RustBelt: Securing the foundations of the rust programming language. *ACM Trans. Softw. Eng. Methodol.* **2018**, *2*, 1–34. [CrossRef]

35. SafeMD. Xiaohua Yin. Available online: https://bitbucket.org/yxhnuaa/safemd/src/master/ (accessed on 10 October 2024).
36. joern. ShiftLeft Corporation. Available online: https://joern.io/ (accessed on 10 October 2024).