Article

Not peer-reviewed version

SafeMD:Ownership-based Safe Memory Deallocation for C Programs

Xiaohua Yin , Zhiqiu Huang * , Shuanglong Kan , Guohua Shen

Posted Date: 2 October 2024

doi: 10.20944/preprints202409.2413.v1

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

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
[2] Department of Computer Science, TU Kaiserslautern
* 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 which enforces automatic deallocation of unused resources without using the garbage collector. In light of Rust's promise of safety, a natural question arises about the possible benefits of exploiting ownership to ensure memory safety of C programs. In our previous work, we developed a formal ownership checker to verify whether a C program follows ownership constraints. A C program that satisfies the ownership constraints is free of dangling pointers. In this paper, we further propose a static ownership-based safe memory deallocation approach, named SafeMD, to ensure memory-leak free in the C programs that satisfy ownership constraints defined in prior formal ownership checker. Benefitting from the C programs satisfying ownership constraints, SafeMD obviates alias and inter-procedural analysis during the finding of fixing patches. Also, the patches generated by SafeMD make the input C programs still satisfy ownership constraints. Usually, a C program that satisfies the ownership constraints is safer than its normal version. Our evaluation shows that SafeMD is effective in fixing memory leaks of C programs that satisfy ownership constraints.

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

1. Introduction

C is widely used for implementing system and embedded software which are usually safety-critical systems. However, its manual memory-management can easily produce dangling pointers and memory leaks (ML) in C programs. Dereferencing and freeing a dangling pointer can cause Use-After-Free (UAF) and Double-Free (DF) memory errors, respectively.

Recently, an emerging programming language designed for highly safe systems, i.e., Rust[1], has received an increasing amount of attention. Dubbed a safer C, Rust combines memory safety and low-level control. It introduces ownership system (OWS) to provide memory safety at compile time. The basic idea of OWS is exclusive ownership, i.e., at any time each resource (mainly for memory) has a unique owner. Ownership of a resource can be transferred from old owner to new owner. To maintain a unique owner, the old owner becomes invalid, and it is no longer used. The above ownership constraints are useful for memory management. On the one hand, 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 side effects like UAF and DF. On the other hand, the ownership constraints rule out aliasing entirely, and consequently avoid dangling pointers at compile time.

In light of Rust's promise of safety, a natural question arises about the possible benefits of exploiting ownership to ensure memory safety of C programs. Because OWS can prevent dangling pointers at compile time, therefore, in our previous work [1], as shown in Figure 1, we developed

[1] https://www.rust-lang.org/

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 October 2024 doi:10.20944/preprints202409.2413.v1

a formal ownership checker, named SafeOSL, to verify whether a C program satisfies ownership constraints. A C program that follows SafeOSL is a memory-safe program that satisfies the ownership constraints and is free of dangling pointers. For such a special C program, in this paper, we can implement an ownership-based memory deallocation, named SafeMD, to automatic drop resources. Like the automatic drop scheme in OWS, SafeMD is also safe and thus does not incur dangling pointers. Finally, the output of SafeMD is a memory-safe C program that is free of dangling pointers and memory leaks. Also, the patches generated by SafeMD make the input C program still satisfies ownership constraints. Usually, a C program that satisfies the ownership constraints is safer than its normal version.

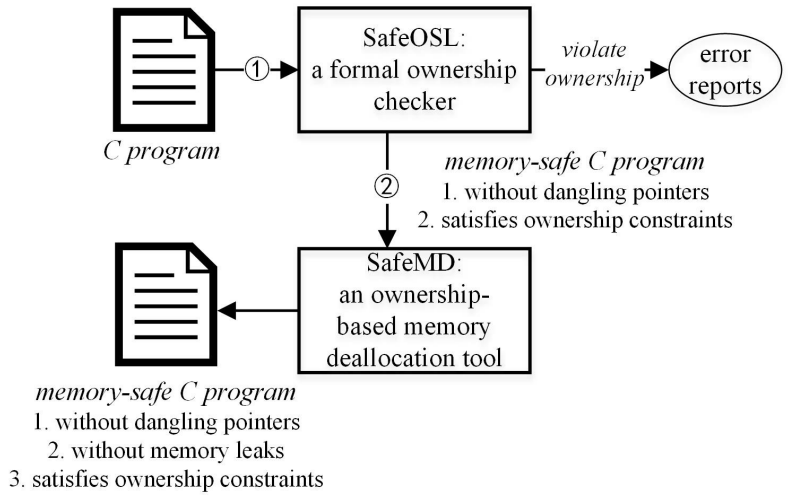

Figure 1. The ownership-based framework of ensuring memory safety of C programs.

Existing Techniques. Regarding the existing technologies for safe memory deallocation [2–6], they suffer from some drawbacks: (1) Because most technologies do not consider whether their input C programs satisfy ownership constraints, they are complex as they often rely on alias and inter-procedural analysis to find the location of memory deallocation. (2) For the input C programs that satisfy ownership constraints, some technologies achieve low repairability due to their limit fixing strategies. For example, consider the C code satisfying ownership constraints in Listing 1, MemFix [5] will insert `free(q)` and `free(p)` at line 19 and 20 to free memory objects, but it may introduce DF (for $o2$) at line 20 and remain a leaky object $o1$. In this case, to safely fix memory leaks, the correct patch `if(flag == 1) free(q);` is required at line 19. But MemFix fails to fix because it is unable to generate patches with conditional statements. Again, consider the simplified C code that satisfies ownership constraints in Listing 2, where the function `f` returns 0 with and without memory leaks. Saver [6] will insert a conditional patch at line 17, but this fix partially eliminates the memory leaks and some leaks still remain. (3) For the input C programs that satisfy ownership constraints, although some technologies can fix their leaks, the patches they generated can not make the input C programs satisfy ownership constraints. Usually, a C program that satisfies the ownership constraints is safer than its normal version.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 October 2024 doi:10.20944/preprints202409.2413.v1


```
1  int *foo1(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 main(){
14   int *p, *q;
15   int flag = 1;
16   p = malloc(1); //o2
17   q = foo1(p, flag);
18   // p is no longer used after here
19   // MemFix/SafeMD: free(q);
20   // MemFix: (double-free) free(p);
21 }
```

Listing 1: MemFix-generated patch

```
1  int f(void *q) {
2    if (...){
3      // SafeMD: free(q);
4      return 0; // memory leak
5    }
6    if (...){
7      // SafeMD: free(q);
8      return -1; // memory leak
9    }
10   free(q);
11   return 0; // no memory leak
12 }
13
14 int g(void *p) {
15   x = f(p);
16   // p is no longer used after here
17   // Saver: (memory leak) if(x==-1) free(p);
18 }
```

Listing 2: Saver-generated patch

Our Approach. In consideration of the above limitation, in this paper, we present SafeMD, an ownership-based safe memory deallocation designed for C programs that satisfy ownership constraints. Benefitting from the C programs satisfying ownership constraints, that is, each object has a unique pointer (owner) at any time, and ownership of object can be transferred in function calls and this designates which function is responsible for deallocating memory objects. This can make SafeMD obviate alias and inter-procedural analysis during the finding of fixing patches. Thus, SafeMD can fix more memory-leak pattern because it fixes memory-leaks only by inserting deallocation statements without combining other expressions such as conditional expressions. For example, for the code in Listing 1, SafeMD separately analyzes function `foo1` and `main`, and generates deallocation statement `free(m)` and `free(q)` at line 5 and 19, respectively. In true branch of `foo1`, SafeMD considers the ownership hold by code and concludes that `foo1` is responsible for freeing the objcet pointed to by `m`. On the contrary, MemFix tries to free this objcet in `main` function, but it fails since it can not generate conditional patches. Although Saver can fix leaks in this code with conditional patches, but the input C program repaired by Saver does not satisfy ownership constraints. Similarly, for the code in Listing 2, SafeMD does not perform inter-procedural analysis to free objects, so it does not depend on the return values of the function to generate patches. Instead, SafeMD will generate the patch `free(q)` at line 3 and 7 in function `f`. This deallocation is safe since `p` is no longer used after line 15 in function `g`. The reason why SafeMD and Saver generate different patches is that SafeMD believes that function `f`, not `g` is responsible for deallocating the object.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 October 2024 doi:10.20944/preprints202409.2413.v1


```
1  p = malloc(1); //o1
2  if(...){
3    q = malloc(1); //o2
4
5  }
6  else{
7    q = p; // p is no longer used after here
8  }
9  ... = *q;
```

Listing 3: buggy code with memory-leaks

MemFix [5] is an automated technique for fixing memory deallocation errors in C programs. It aims to fix ML by finding a set of free-statements that correctly deallocate all allocated objects without causing DF and UAF. In this paper, we present SafeMD, which modifies the algorithm of MemFix to serve only for C programs that satisfy ownership constraints.

Review of the Ideas of MemFix. Before explaining our algorithm, let us first explain the ideas of MemFix. Given a buggy program where all deallocation statements are removed before analysis, MemFix works with two steps: 1) using a static analysis that collects patch candidates for each allocated object, and 2) finding correct patches by solving an exact cover problem. Before these two steps, MemFix need to run standard alias analyses to get may-alias and must-alias information.

Consider the simple example in Listing 3, MemFix first analyzes the code to collect patch candidates based on control-flow graph of code, as presented in step 1 of Figure 2. MemFix uses a state of the form $\langle o, must, mustNot, patch, patchNot\rangle$ to maintain points-to and patch information for each allocated object, 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 o, and $patchNot$ is a set of potentially unsafe patches that may cause UAF or DF. Each patch in $patch$ and $patchNot$ is a pair (n, e) of a program location n and a pointer expression e, which denotes a deallocation statement `free(e)` can be inserted right after line n.

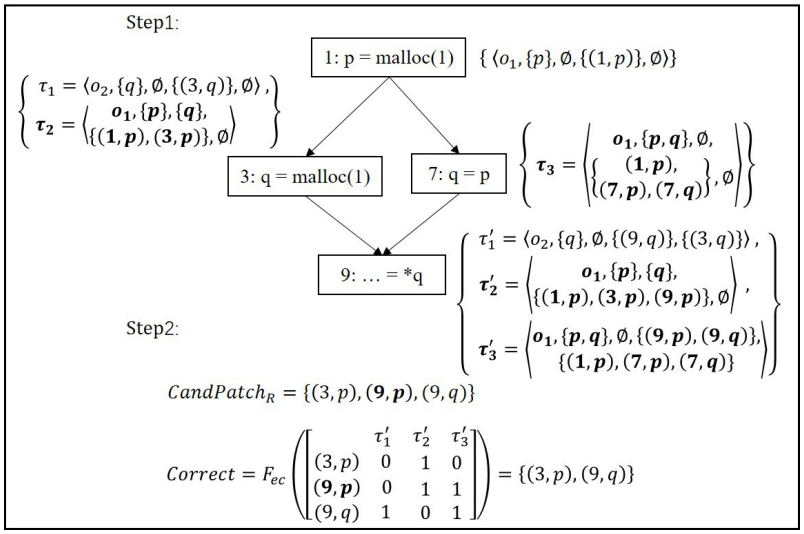

Figure 2. The analysis performed by MemFix

The analysis of MemFix is flow-sensitive, and thus it updates the states for each statement until reaching the exit of code. For example, the allocation statement at line 1 creates a new state, where object o_1 is pointed to by p, and it can be deallocated safely by inserting `free(p)` at line 2 (after line 1). Line 3 also creates a new state for o_2, which may change the state of o_1. Because q definitely no

 doi:10.20944/preprints202409.2413.v1

longer point to o_1, q is added to *mustNot* of o_1. Also, *patch* of o_1 includes $(3, p)$ and $(1, p)$ since it is safe to deallocate o_1 after line 1 or 3 via `free(p)`. At line 7, the state of o_1 is updated to τ_3. Due to the assignment `q = p`, both q and p point to o_1. The object can be safely deallocated with either `free(p)` or `free(q)` after line 7. The analysis of MemFix is disjunctive and maintains states separately for each different branch, therefore, for the joint point at line 9, the state τ_1, τ_2 and τ_3 are changed to τ_1', τ_2' and τ_3', respectively. Because q is used at line 9, the existing *patch* for the objects pointed to by q are no longer safe due to use-after-free errors. Thus, MemFix removes the patches from *patch* and instead adds them to *patchNot*. For example, the patch $(3, q)$ is removed from τ_1's *patch* to τ_1''s *patchNot*. Also, in τ_1' the new safe patch becomes $(9, q)$. However, the *patch* of τ_2 remains safe because q does not point to o_1 and therefore the corresponding object is not be used at line 9. Instead, a new safe patch $(9, p)$ is added to *patch*, yielding τ_2'.

After analysis in step 1, MemFix has collected all patch candidates for each allocated object. To find a set of patches from patch candidates that correctly deallocate all objects without introducing DF and UAF, MemFix solves an exact cover problem to find such correct patches, as shown in step 2 of Figure 2. From the states τ_1', τ_2' and τ_3', the analysis first collects candidate patches that are included in *patch* set but not in *patchNot*: $CandPatch_R = \{(3, p), (9, p), (9, q)\}$. For example, $(1, p)$ is in *patch* of τ_2' while in *patchNot* of τ_3', so it is excluded from $CandPatch_R$. The patches in $CandPatch_R$ can not cause UAF, but may cause DF because of aliases. Therefore, MemFix finds a subset of the candidate patches that does not introduce DF, which corresponds to solving an exact cover problem represented by the following incidence matrix:

	τ_1'	τ_2'	τ_3'
$(3, p)$	0	1	0
$(9, p)$	0	1	1
$(9, q)$	1	0	1

Each row r in matrix represents a patch in $CandPatch_R$ and each column τ represents a state. 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 $(9, q)$ in *patch*, so the entry in row $(9, q)$ and column τ_1' is 1. Solving an exact cover problem represented by the above incidence matrix is the selection of rows such that each column contains only single 1 among selected rows. In this example, the correct patches are computed as $\{(3, p), (9, q)\}$ which covers all states (i.e., no ML) and each state is covered by at most one patch (i.e., no DF).

Through the above analysis, MemFix will insert `free(p)` after line 3 and `free(q)` after line 9 to safely deallocate all allocated objects in code.

Our Proposal in This Paper. We modify the algorithm of MemFix to deallocate the objects in C programs that satisfy ownership constraints. SafeMD also contains the same two steps as MemFix. Its analysis results are presented in Figure 3, where the results different from MemFix are highlighted in bold. This code can be considered to satisfy ownership constraints, because p is no longer in use after line 7, which follows the constraint that old owners can not be accessed after ownership transferring and thus satisfies the exclusive ownership. Therefore, SafeMD does not need alias analyses before analysis. In step 1 of collecting patch candidates, because the code satisfies ownership constraints, the key idea for deallocation is that an object is deallocated only by its unique owner (rather than together with old owners), which is different with MemFix who will deallocate an object via all must-alias (may contain old owners). Based on the above idea, we use the state of the form $\langle o, newOwner, oldOwners, patch, patchNot\rangle$ to maintain owner and patch information for each allocated object, where *newOwner* is a pointer who is the unique owner of o and *oldOwners* is a set of pointers that are the old owners of o.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 October 2024 doi:10.20944/preprints202409.2413.v1

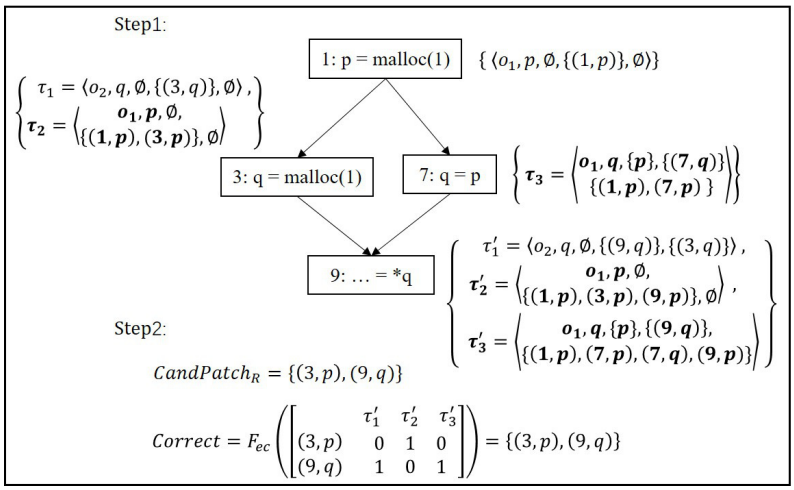

Figure 3. The analysis performed by SafeMD

SafeMD will first update the owner information and then patch information. The allocation at line 1 creates a new state where the ownership of object o_1 is bound to p, so the *newOwner* and *oldOwners* of o_1 is p and empty, respectively. The safe patch $(1, p)$ indicates that we can safely free o_1 via owner p after line 1. The pointer assignment at line 3 creates a new ownership binding for o_2, but it does not change the owner for o_1. The assignment at line 7 can transfer the ownership of o_1 from p to q, making q become new owner while p become old owner. The only safe patch for o_1 is $(7, q)$, all other patches via old owners, namely, $(1, p)$ and $(7, p)$ are considered to be unsafe patches. This patch update is different from MemFix who tries to free o_1 via all must-alias (p and q are must-alias). At line 9, the assignment uses the object pointed to by q , but it does not transfer ownership of any object, therefore, owner information of each state remains the same. For each state, a new safe patch $(9, t)$, where $t = newOwner$, is added to *patch*. For example, $(9, p)$ is generated in τ_2'. However, because q is used at line 9, we remove $(3, q)$ and $(7, q)$ from *patch* of τ_1 and τ_3, respectively, and declare them unsafe in τ_1' and τ_3'. More importantly, for the state τ_3' whose *oldOwners* is not empty, we generate a new unsafe patch $(9, p)$ for old owner p, because an object can not be deallocated by old owners.

In step 2 of finding correct patches that do not cause DF, the size of $CandPatch_R$ collected by SafeMD is smaller than that collected by MemFix. This is because SafeMD adds all patches related with old owners (aliases) to *patchNot* in step 1, which excludes a plenty of unsafe patches that may cause DF due to pointer aliasing. For example, $(9, p)$ will be added in *patchNot* at line 9 and thus excluded from $CandPatch_R$. This can reduce the search space for solving exact cover problem.

We implement SafeMD based on the ideas described above. It has distinct features as follows:

- SafeMD only works for C programs that satisfy ownership constraints.
- If C programs satisfy ownership constraints, SafeMD obviates alias and inter-procedural analysis compared with most existing techniques on memory deallocation.
- If C programs satisfy ownership constraints, SafeMD can safely deallocate objects only by inserting `free` statements instead of conditional patches (e.g., MemFix and Saver).
- If C programs satisfy ownership constraints, SafeMD deallocates objects only via its unique owner and thus generates a smaller search space for correct patches compared with existing techniques (e.g, MemFix).
- If C programs satisfy ownership constraints, the patches generated by SafeMD make the input C programs still satisfy ownership constraints. A C program that satisfies the ownership constraints is safer than its normal version.

This paper makes the following contributions:

 doi:10.20944/preprints202409.2413.v1

- We present SafeMD, an ownership-based safe memory deallocation technique for C programs that satisfy ownership constraints.
- We implement SafeMD and conduct experiments to demonstrate its effectiveness.
- We explore the benefit of Rust's novel ownership-based memory management in C.

The rest of this paper is organized as follows. Section 2 introduces the ownership system in Rust. Section 3 presents the algorithm of SafeMD. Section 4 reports experimental results. Section 5 discusses related work and Section 6 concludes the paper.

2. Rust Ownership System

Compared with C, the distinctive advantage of Rust is to introduce ownership system and consequently guarantee memory safety at compile time. 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 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 4, 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 $s2$, making $s1$ is no longer valid until it is re-assigned a value again. Therefore, Rust compiler will issue an error at Line 5. This is different with pointer assignments in C where both $s1$ and $s2$ is valid. The above kind of transferring is called *move*. Because $s2$ is the unique owner of o, when the owner $s2$ goes out of its scope, Rust compiler automatically inserts `drop` destructor to free o at line 6. Therefore, the code at line 7 is rejected as o has been already destroyed. Now, we take a closer look at line 8. When $s1$ goes out of its scope and tries to free o, Rust compiler does not insert `drop` since it finds the ownership of $s1$ has been moved. This can ensure memory deallocation does not introduce DF.

```
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.
```

Listing 4: Transferring ownership in assignments

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 5, the function call at Line 3 moves the ownership of the object o created at line 2 to `takes_ownership`, so Rust compiler will issue an error at line 4 where s becomes old owner and can not be accessed in `main` function. Rust compiler compiles each function individually, it will rely on ownership to determine whether to insert `drop` destructors to free objects. For example, Rust compiler first compiles `main` function. It finds that s is moved to `takes_ownership` and thus it does not insert `drop` to free o at line 5. Next, Rust compiler compiles `takes_ownership` function. Because ss is a String type which can move ownership, therefore, Rust compiler will automatically insert `drop` once ss goes out of its scope at line 9.

 doi:10.20944/preprints202409.2413.v1


```
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.
```

Listing 5: Transferring ownership via parameters

```
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.
```

Listing 6: Transferring ownership via return values

Besides parameter passing, return values can also transfer ownership. For example, in Listing 6, `gives_back` moves ownership out from `gives_back` via return value *ss* to its caller `main`. In this case, exclusive ownership still ensures automatic memory deallocation is safe. When Rust compiler compiles `main` function, the object *o* is dropped automatically once *s*1 goes out of scope at line 6 but nothing happens for *s* because *s* is moved. This avoids DF when *s* and *s*1 go out of their scope (line 6) and both try to free the object *o*. When compiling `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 *s*1 is used in `main` function.

3. Approach Details

This section presents the algorithm of SafeMD. Section 3.1 defines a core language. The two steps of SafeMD are presented in Sections 3.2 and 3.3, respectively. Note that SafeMD is modular, it performs step 1 and step 2 for each function in programs to generate corresponding correct patches.

3.1. 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:

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

A pointer expression p can be a variable x , a pointer dereference $*x$ or a NULL pointer. Allocation command $\text{alloc}(p)$ creates a new memory object pointed to by p. Assignment command $\text{assign}(p, e)$ assigns the expression e to p. The command $\text{funDef}(fname, par\_list,$

 doi:10.20944/preprints202409.2413.v1

par_t_list, ret_t) describes a function signature, where $fname$, par_list, par_t_list and ret_t denote 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 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.

Besides, the program P must also satisfy ownership constraints, and thus is free of dangling pointers.

3.2. Step 1: Collecting Patch Candidates by Ownership Tracking.

The first step of SafeMD is to statically analyze each function in programs to collect patch candidates. The key idea is to track owner of each object and deallocate object only via its unique owner.

3.2.1. Abstract Domain

The abstract domain of the analysis is defined as follows:

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

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 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 3.1, AP equals to the set of pointer expression p. We assume that a pointer expression p always points to heap objects rather than stack objects. The symbol $\bot$ denotes the invalid access paths. Each element in $patch$ and $patchNot$ is a pair $(c, p) \in \mathbb{C} \times AP$ which 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.

3.2.2. Abstract Semantics

SafeMD collects patch candidates for each function individually. For each function, the analysis is start 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:

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

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

If the function $fname$ has no parameter, S_0 is empty; otherwise, it is computed by $\gamma_c : Var \to State$ which generates a state for each parameter of $fname$. The function $isPointer(x)$ equals true if variable x points to a heap object. Because the ownership of objects can be transferred by parameter passing, a

 doi:10.20944/preprints202409.2413.v1

new state is created: the allocation-site of object is the program point c, the unique owner of object is par, and the safe patch is (c, par). For example, in Listing 1, 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 $\mathrm{lf}_{\mathrm{p}} F \in \mathbb{D}$ of the semantics function $F \in \mathbb{D} \to \mathbb{D}$:

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

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 is created, because the input program of SafeMD ensures the lack of dangling pointers and thus the returned pointer of callee always points to a valid heap memory. For example, in Listing 1, the call ″q = foo1(p, flag)″ will create the new state $\langle o_3, q, \emptyset, \{(17, q)\}, \emptyset \rangle$, where o_3 is the object that q points to.

The set S' is updated by two transfer functions ϕ_c and φ_c: $S' = \bigcup_{s \in S} (\varphi_c \circ \phi_c)(s)$. 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 unique owner. GO is the set of unsafe patches that are newly generated at c via old owners, because an object can not 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 the $oldOwners'$ to ensure that $newOwner'$ and $oldOwners'$ are disjoint.

 doi:10.20944/preprints202409.2413.v1

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 the $patchNot'$, it adds a set of new unsafe patches related with old owners, which contains *patch* and GO. The pointer assignment `q = p` 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 object.

For the *otherwise* case where o is not be used at c, we only merge new safe 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 to the owner of object o in state s, then the ownership of o is moved out from callee and p becomes old owner. We use the symbol $\perp$ to indicate 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 caller. We reset *patch* to $\emptyset$ to indicate that the callee can not free o since it has lost ownership of o. The responsibility for deallocating o falls on the caller. Consider the `foo1` in Listing 1, SafeMD will merge the states computed from if-else branch as follows:

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

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 to $\perp$ and q becomes old owner. Also, their *patch* is reset to $\emptyset$. For state τ_1, it remains unchanged. The update result is shown below.

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

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 October 2024 doi:10.20944/preprints202409.2413.v1

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

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

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

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

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

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

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

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

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

We discuss three cases. The first case is $newOwner = \bot$, which indicates the ownership of object is moved. For the $patchNot'$, it adds a set of new unsafe patches related with old owners, which contains $patch$ and GO. We reset the $patch$ to $\emptyset$ to denote that caller is not responsible for deallocating object since the ownership of object is moved into callee. For example, consider the `main` in Listing 1, line 16 creates a new state $\langle o_1, p, \emptyset, \{(16, p)\}, \emptyset \rangle$. The call `q = foo1(p,flag)` at line 17 updates the state to $\langle o_1, \bot, \{p\}, \emptyset, \{(16, p), (17, p)\} \rangle$, since this call transfers the ownership of o_1 to callee `foo1`.

The second case is "$newOwner \neq \bot$ but o is used at c", which indicates the ownership of object o is not be 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, is belongs to this case.

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

3.3. 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 proposed to find correct patches by solving an exact cover problem. We use this method but modify it because ownership is considered.

Let $R = (\mathrm{lf}_{\mathrm{p}} F)(c_x) \subseteq State$ 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 defines as follows:*

$$Safe = \bigcup \{patch \mid \langle \_, \_, \_, patch, \_ \rangle \in R\}.$$

$$UnSafe = \bigcup \{patchNot \mid \langle \_, \_, \_, \_, patchNot \rangle \in R\}.$$

$$CandPatch_R = Safe \setminus UnSafe.$$

 doi:10.20944/preprints202409.2413.v1

$Safe$ 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 get the set $CandPatch_R$ that are used to find correct patches.

The patches in $CandPatch_R$ can not cause UAF since these unsafe patches are all collected in $UnSafe$ and thus already excluded from $CandPatch_R$. Besides, the patches in $CandPatch_R$ also satisfy ownership constraints, that is, old owners can not be used to deallocate objects. After all, the C programs satisfying ownership are more safe than normal C programs. However, the patches in $CandPatch_R$ may cause DF. Thanks to ownership, a plenty of unsafe patches caused by pointer aliasing are excluded from $CandPatch_R$, since SafeMD collects the unsafe patches related with 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. For example, in the $CandPatch_R$ of Figure 2, we need to verify the combination of $(9, p)$ and $(9, q)$ to find the correct patches, but not for the $CandPatch_R$ of Figure 3, because $(9, p)$ is related with old owners and thus excluded from $CandPatch_R$. However, $CandPatch_R$ can not 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 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 kind of DF.

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

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

Next, we present the definition of 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 Correct covers all valid states, and*
- *$M(r_1) \bigcap M(r_2) = \emptyset$ for all r_1,$r_2 \in Correct$, which means the chosen subsets in $M(r)$ (where $r \in Correct$) are pairwise disjoint.*

M describes the incidence matrix in Figure 3. The first condition means 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 $CandPatch_R$.

4. Evaluation

We show that SafeMD is effective in deallocating memory objects of real-world C programs that satisfy ownership constraints. Our experiments are performed on a PC with Intel Core i7-7700 CPU (3.60GHZ) and 8GB RAM running 64-bit Ubuntu 18.04.3 LTS.

 doi:10.20944/preprints202409.2413.v1

4.1. Implementation

We implemented SafeMD as a stand-alone tool[2]. Specifically, we first make use of the open-source code analysis platform for C/C++ based on code property graphs, Joern[3], to extract CFGs for all functions in our benchmarks. Then all CFGs (Dot files) are loaded into NetworkX for graph traversal to calculate candidate correct patches and correct patches. 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.

In step 1, recall that the generation of initial states S_0 mentioned in Section 3.2.2, 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 `main` function and proceed according to call graph, and set 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 exact cover problem. This algorithm takes optimization strategies to improve the search speed for exact cover problem. Also, our algorithm of finding correct patches will not find all solutions, instead, it will return the first solution it finds.

4.2. Benchmark

We use three benchmarks to evaluate SafeMD. In Table 1, the first benchmark is relevant to memory leak (CWE-401) in Juliet Test Suite (JTS) for C, ″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 [5] who constructed them from 5 GitHub open-source C repositories. The last benchmark listed in Table 2 consists of 15 programs selected from GNU Coreutils-8.29. These programs use dynamic memory allocation except for `realloc`.

The column Program* stresses that all programs in these benchmarks are modified to satisfy ownership constraints. For the C programs that are difficult to modify have been removed from the benchmarks, leaving a total of 120 programs (76 in CWE-401, 26 in open-source C repositories and 15 in GNU Coreutils). Note that the modifications on benchmarks do not guarantee semantics preservation, since in this experiment we aim to provide the test programs that satisfy ownership constraints and how to rewrite C programs to satisfy ownership constraints 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 (except for `realloc`) and modify these pointers to make their use satisfy ownership constraints. For better understanding on experimental result, we briefly list the main code features of C programs that satisfy ownership constraints below.

- **Assignments.** For the pointer assignments, such as `q = p;`, if variable p points to a heap object, then p can not 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;`, then the struct itself p can not 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 caller, p can not be used after call-site. For the library functions (except for standard allocation and deallocation functions), such as `memcpy(p,...)`, because we can not 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 caller.

---

[2] https://bitbucket.org/yxhnuaa/safemd/src/master/
[3] https://joern.io/

 doi:10.20944/preprints202409.2413.v1

4.3. Results

Table 1 shows the results on JTS and open-source C repositories. #Loc represents the average number of lines of code. 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 the programs have. #LTime reports the maximum execution time performed by SafeMD. Fix/#Pgm represents the average success rate of SafeMD. For CWE-401, SafeMD succeeded to fix 95% (72/76) on average as these programs have relatively simple structures and data types. Four test programs including function pointers are not supported by SafeMD. For open-source C repositories, SafeMD can generate correct patches for 69% (18/26). These programs frequently use low-level C features and more complex data types to store memory objects, such as arrays and linked list. For such data types, it is common to traverse them using two pointers, but the assignment between two pointers makes only one pointer is valid, and this valid pointer is used to traverse the data. When the traversal is finished, SafeMD will deallocate the data with this valid pointer, which result in invalid-free errors, as discussed in Section 4.4. Besides, the maximum execution time for fixing these two benchmark is smaller than 1.0s.

Table 2 shows the results on GNU Coreutils. #Allocation reports the number of allocation-sites (malloc, calloc and strdup) in the programs. A program is fixed if all allocation-sites are safely deallocated by a SafeMD-generated patch. The results show that SafeMD can repair 7 out of 15 programs. These programs also have more complex structures and data types, and often use memory-allocators `strdup`. Because the current version of SafeMD cannot handle such features, the portion of successful fixing is relatively low in GNU Coreutils (see discussion in Section 4.4).

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

Benchmark	Program*	#Loc	#Function	Fix/#Pgm.	#LTime(sec.)
CWE-401	int_malloc	82	10	36/38	<1.0
	twoIntsStruct_malloc	92	9	36/38	<1.0
Open-Source C Repo.	Binutils	127	6	4/5	<1.0
	Git	150	5	2/5	<1.0
	OpenSSH	150	4	5/6	<1.0
	OpenSSL	134	3	3/4	<1.0
	Tmux	154	6	4/6	<1.0

Table 2. Evaluation results on GNU Coreutils

Benchmark	Program*	#Loc	#Function	#Allocation	#Fix*	#Time(sec.)
GNU Coreutils	yes	116	2	1	N	
	users	157	5	1	N	
	unexpand	215	3	1	Y	<1.0
	tee	221	3	1	Y	<1.0
	mktemp	293	6	3	N	
	nl	484	11	4	Y	<1.0
	ln	459	6	2	Y	<1.0
	printf	562	8	1	Y	<1.0
	stdbuf	286	7	3	N	
	wc	686	7	1	Y	1.2
	shred	945	17	5	N	
	cp	946	8	5	N	
	install	818	20	1	Y	3.0
	who	634	18	8	N	
	tr	1363	36	10	N	

* N means cannot fix, Y means can fix

 doi:10.20944/preprints202409.2413.v1

4.4. Discussion

The success rate of SafeMD is relatively low, such as in GNU Coreutils, primarily due to the ownership limitations of SafeMD. This means the strictness of ownership enforced in C makes it unsafe for SafeMD to release objects in some situations.

```c
void foo(int *t){
  printf("%d", *t);
  // SafeMD: free(t);
}

int main(){
  int i = 3;
  int *q = malloc(1);
  ...
  foo(&i);  // invalid free
  foo(q); // correct free
  return 0;
}
```

Listing 7: invalid free on calls

One situation, where SafeMD fails, is when an argument passed to a function does not always point to heap objects in different call context. Consider the Listing 7's code pattern found in who program, SafeMD will start from `main` function, then proceed to `foo` function after the analysis of `main` is finished. During the analysis of `main`, for the call `foo(&i)`, because &i points to a stack object, the flag with this call-site is marked with false, indicating the initial state S_0 of `foo` is empty. However, for the next call `foo(q)`, because q points to a heap object whose ownership is transferred to `foo(q)`, the flag with this call-site is set to true, indicating the initial state S_0 of `foo` is not empty. This different call context makes SafeMD can not generate the safe patch for `foo`, because the patch $(2, t)$ is safe for `foo(q)` but not safe for `foo(&i)`. We give a warning to indicate this case can not be fixed by SafeMD. This situation is absent in Rust, as in Rust the type of parameters can explicitly express whether its ownership is transferred. But C has no built-in ownership, so the ownership of pointers is implicit and determined by the objects to which the pointers point.

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

Listing 8: invalid free in pointer traversal

```c
bool f(char *s,...){
  s = xmalloc(...);
  ... // use s
  // SafeMD: free(s);
  // s is dangling
  return true;
}

void g(...){
  char *s;
  bool ok = f(s);
  v(s); // UAF error
}
```

Listing 9: UAF in output parameter

 doi:10.20944/preprints202409.2413.v1

Another situation is when enforce ownership constraints on operations related with array and linked list. Consider the Listing 8's code pattern found Git. Line 3 creates an object $o1$. Although this code is relevant to array traversal, it follows the ownership constraints since the old owner $msg2$ is no longer be 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 $o1$. Such C-style pointers are usually represented in Rust as slice references who are a type that can access a portion of an array without taking ownership of the array it references.

The last situation is caused by an output parameter (a parameter that can be accessed from outside the function). Consider the Listing 9's code pattern found in tr, users, and cp program. In `g()`, because s is not initialized by dynamic memory allocation functions, SafeMD considers that s is not be moved into `f()`and thus it can be used again in `v()`. In `f()`, an object is allocated for s and it does not be returned to caller. According to ownership constraints, SafeMD will generate `free(s)` at line 4, making s be a dangling pointer. However, since s is an output parameter, a UAF error is occurred at line 12. In Rust, output parameters can be represented as reference type that can use value without taking ownership of value. Therefore, s does not be dropped in `f()`, but rather returned to the caller. However, there is no reference type in C. One way of addressing this problem is to modify such code by returning output parameters explicitly, i.e., we rewrite the code `return true;` to `return s;`. This modified code can eliminate UAF as s is moved out from `f()` and thus SafeMD does not generate patch for s in `f()`.

5. Related Work

Memory-leak fixing techniques. We focus on the static analysis techniques for C programs. Because these work targets general C programs, they often rely on alias and inter-procedural analysis to aid in fixing memory leaks. LeakFix [2] can identify and safely fix memory leaks. It uses pointer analysis, which is an inter-procedural, flow-insensitive, context-sensitive, to get an SSA-based points-to graph for each procedure. Also, it performs an inter-procedural analysis to identify which procedure should fix the leaked objects. AutoFix [3] combines static analysis with runtime checking to prevent memory leaks. In its implementation of static analysis, Andersen's pointer analysis is used to build the value-flow graph (VFG) for the program. FootPatch [4] can fix memory leaks in large code bases but may introduce new errors such as DF as a side-effect. Memfix [5] can safely repair ML, DF and UAF in a unified fashion. It removes memory errors by collecting patch candidates and solving an exact cover problem. Before analysis, Memfix performs standard pointer and alias analyses. Saver [6] proposes object flow graphs (OFG), which captures the program's heap-related behavior, to safely fix memory errors such as ML, UAF and DF. Pointer analysis is used to construct OFG. A framework that is proposed in [7] presents practical solutions for developers to automate C program repair. Before generating patches, it uses program analysis, including static analysis and code instrumentation techniques, to extract patch conditions that can guide the generation of a correct patch.

For the C programs that satisfy exclusive ownership, the work mentioned above still performs alias and inter-procedural analysis to fix memory leaks, making the repair complex and inefficiency. However, the ownership constraints satisfied by input programs can ease memory-leak fixing. Compared with the work mentioned above, SafeMD obviates alias and inter-procedural analysis for safe memory deallocation. Besides, the patches generated by SafeMD satisfy ownership constraints. A C program that satisfies the ownership constraints is safer than its normal version.

Ownership Discussion. Ownership has been used in OO programming to enable controlled aliasing [8,9] and prevent data races [10,11]. Similarly, the concept of ownership has also been applied to analyse C/C++ programs. Heine et al. [12] presents an ownership type system to detect ML and DF. Their ownerships range over integer values $\{0,1\}$. However, their model adds optional ownership transfer in assignment, allows arbitrary aliases, and thus can not detect UAF errors. Kohei et al. [13] proposes 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 $x \in [0,1]$.

 doi:10.20944/preprints202409.2413.v1

Tatsuya et al. [14] extends the fractional-ownership type system in [13] to fix memory leaks. Their technique conducts type inference for the extended type system to detect where to insert deallocation statements. Compared with these work, the goal of SafeMD is to fix memory leaks, instead of detecting them. Besides, SafeMD takes advantage of Rust's ownership, called borrow-based ownership, not fractional-based ownership. Borrow-based ownership of Rust has been proven to be effective in preventing memory errors [15,16]. Inversely, fractional-based ownership is not implemented as a language feature like Rust's ownership; it's more of a design concept. Also, it allows multiple owners by assigning fractions of ownership and thus require more complex reasoning about ownership.

Rust. A majority of existing work towards Rust mainly focuses on formal verification of Rust programs [17–19], empirical research on effectiveness of Rust in fighting against memory bugs [16,20] and bug detection of Rust programs [21,22]. Recently, some work on (semi-)automatically translating C code to Rust are proposed [23–25]. Their idea is to first translate C programs to Rust programs using c2rust[4] (only grammatical transformation) and then remove unsafe features in translated Rust programs, e.g., translating unsafe pointers to safe Rust references. Although our work, i.e., SafeOSL and SafeMD, is not translating C to Rust directly, both of our work and their work leverage ownership mechanism of Rust to generate a safer C program. We believe our work can be used in conjunction with their approach to ensure memory safety in C programs.

6. Conclusions

We propose SafeMD, an ownership-based memory deallocation for C programs that satisfy ownership constraints. Benefitting from ownership, SafeMD obviates alias and inter-procedural analysis during collecting patch candidates (step 1), and reduces the search space for finding the correct patches (step 2). 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 memory safety of C language. However, the conducted experiment is limited due to ownership limitations of SafeMD. As future work, we plan to consider more memory-allocators, and relax ownership constraints according to semantics of C language (e.g., array traversal) to make SafeMD more practical.

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

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

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

References

1. X. H. Yin; Z. Q. Huang; S. L Kan; G. H. Shen; Y. Liu; F. Wang. SafeOSL: Ensuring memory safety of C via ownership-based intermediate language. *Softw. Pract. Exp.* **2022**, *52*, 1114–1142.
2. Q. Gao; Y. F. Xiong; Y. H. Mi; L. Zhang; W. K. Yang; A. P. Zhou; B. Xie and H. Mei. Safe Memory-Leak Fixing for C Programs. In Proceedings of 37th IEEE/ACM International Conference on Software Engineering, Florence, Italy, 16-24 May 2015; Volume 1.
3. H. Yan; Y. L Sui; S. P. Chen; J. L. Xue. Automated memory leak fixing on value-flow slices for C programs. In Proceedings of 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4-8 April 2016.
4. R. J. Tonder; C. L. Goues. Static automated program repair for heap properties. In Proceedings of 40th International Conference on Software Engineering, Gothenburg, Sweden, May 27 - June 03, 2018.
5. J. Lee; S. H. Hong; H. J. Oh. MemFix: Static analysis-based repair of memory deallocation errors for C. In Proceedings of 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.

[4] https://c2rust.com/manual/intro.html

 doi:10.20944/preprints202409.2413.v1

6. S. H. Hong; J. Lee; J. S. Lee; H. J. Oh. SAVER: scalable, precise, and safe memory-error repair. In Proceedings of 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020.
7. J. F. Xuan; Q. Xin; L. Q. Chen; X. G. Mao. Potential Solutions to Challenges in C Program Repair: A Practical Perspective. In Proceedings of 38th IEEE/ACM International Conference on Automated Software Engineering, Luxembourg, 11-15 September 2023.
8. D. G. Clarke; J. Potter; J. Noble. Ownership Types for Flexible Alias Protection. In Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Vancouver, British Columbia, Canada, 18-22 October 1998.
9. D. G. Clarke; J. Ostlund; I. Sergey; T. Wrigstad. SafeOSL: Ownership Types: A Survey. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification* **2013**, *7850*, 15–58.
10. C. Boyapati; R. Lee; M. C. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Seattle, Washington, USA, 4-8 November 2002.
11. J. Kloos; R. Majumdar; V. Vafeiadis. Asynchronous Liquid Separation Types. In Proceedings of 29th European Conference on Object-Oriented Programming, Prague, Czech Republic, 5-10 July 2015; Volume 37.
12. D. L. Heine; M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, California, USA, 9-11 June 2003.
13. K. Suenaga; N. Kobayashi. Fractional Ownerships for Safe Memory Deallocation. In Proceedings of 7th Asian Symposium on Programming Languages and Systems, Seoul, Korea, 14-16 December 2009; volume 5904.
14. T. Sonobe; K. Suenaga; A. Igarashi. Automatic Memory Management Based on Program Transformation Using Ownership. In Proceedings of 12th Asian Symposium on Programming Languages and Systems, Singapore, 17-19 November 2014; volume 8858.
15. R. Jung; J. H. Jourdan; R. Krebbers; D. Dreyer. RustBelt: Securing the foundations of the rust programming language. *ACM Trans. Softw. Eng. Methodol.* **2018**, *2*, 1–34.
16. H. Xu; Z. B. Chen; M. S. Sun; Y. F. Zhou; M. R. Lyu. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *Proc. ACM Program. Lang.* **2022**, *31*, 1–25.
17. J. Toman; S. Pernsteiner; E. Torlak. Crust: A Bounded Verifier for Rust (N). In Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering, Lincoln, NE, USA, 9-13 November 2015.
18. V. Astrauskas; A. Bílý; J. Fiala; Z. Grannan; M. Zhang; C Matheja; P. Müller; F. Poli; A. J. Summers. The Prusti Project: Formal Verification for Rust. In Proceedings of 14th International Symposium on NASA Formal Methods, Pasadena, CA, USA, 24-27 May 2022; volume 13260.
19. A. Lattuada; T. Hance; C. Cho; M. Brun; I. Subasinghe; Y. Zhou; J. Howell; B. Parno; C Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* **2023**, *7*, 286–315.
20. B. Q. Qin; Y. L. Chen; Z. M. Yu; L. H. Song; Y. Y. Zhang, Understanding memory and thread safety practices and issues in real-world Rust programs. In Proceedings of 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, London, UK, 15-20 June 2020.
21. Z. H. Li; J. C. Wang; M. S. Sun; J. S. Lui. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In Proceedings of ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, 15-19 November 2021.
22. M. Cui; C. J. Chen; H. Xu; Y. F. Zhou. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–21.
23. M. Emre; R. Schroeder; K. Dewey; B. Hardekopf.Translating C to safer Rust. *Proc. ACM Program. Lang.* **2021**, *5*, 1–29.
24. M. Emre; P. Boyland; A. Parekh; R. Schroeder; K. Dewey; B. Hardekopf. Aliasing Limits on Translating C to Safe Rust. *Proc. ACM Program. Lang.* **2023**, *7*, 551–579.
25. H. L. Zhang; C. David; Y. J. Yu; M Wang. Ownership Guided C to Rust Translation. In Proceedings of 35th International Conference on Computer Aided Verification, Paris, France, 17-22 July 2023; volume 13966.

 doi:10.20944/preprints202409.2413.v1

disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.