Master's Thesis

Automated Memory Error Repair Based on Hybrid Program Analysis

ZECHANG QIAN

20M31355

Graduate Major in Computer Science
School of Computing
Tokyo Institute of Technology

Supervisor: Katsuhiko Gondow

January, 2022

Abstract

Automated program repair is a technology that aims to fix program errors and vulnerabilities automatically. In the field of memory error repair, with the development of bug detection tools, we can easily detect memory errors in programs. However, fixing those errors is time-consuming and error-prone. Because the program ' s heap-related behavior plays a critical role in memory error repair, the existing techniques are mainly based on static analysis, where the static analyzer is used to detect program memory errors and then repair tools collect the essential information via static analysis. But since a static analyzer may give wrong alarms which will affect the performance of the repair tools, and static analysis often requires high overhead.

We present **HAMER**, a hybrid automated memory error repair tool that aims to address those shortcomings by using hybrid analysis. HAMER first uses fuzzer to check the alarms given by the static analyzer and extracts the real errors from those alarms. Then it tries to fix those errors by using hybrid analysis. HAMER is the first automated memory error repair technique based on fuzzing results. HAMER does not waste time resolving alarms that are incorrect because the errors reported by fuzzer are real errors, and HAMER also utilizes fuzzer to test the generated patches to ensure they are correct. For the necessary information for synthesizing patches, HAMER uses lightweight static analysis techniques to collect it. evaluation and conclusion

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory errors, such as memory leaks, can have catastrophic effects, thus detecting and fixing them has always been a critical task for developers. Memory error detection performance is improving with the development of memory error detection technologies, however resolving these problems takes a lot of time and work for developers, and erroneous patches might lead to more significant effects.

Existing memory error repair techniques [5,7] are mainly based on static analysis. This is because resolving issues like memory leaks necessitates an understanding of heap-related behavior, such as error source and sink. Collecting heap-related behavior information needs a high time and space overhead. Because wrong patches or fixing locations might lead to more significant errors, and different heap objects can interact with each other, all of this must be fully considered when generating patches. The state-of-the-art automated memory error repair technology SAVER [5] saves these heap-related behaviors by constructing an object flow graph, which has high time and space complexity. However, because static analysis techniques are not good at dealing with problems such as indirect calls and alias, the repair tools based on static analysis might generate the wrong patches. While dynamic analysis can help with these issues, it does not provide enough information to generate patches.

In this paper, we present **HAMER**, a hybrid analysis-based memory error repair technique. We use a static analyzer to detect the program first, then use a fuzzer to detect the alarms and extract the real errors. Fuzzer triggers runtime errors, thus it will not give false-positive alarms, allowing HAMER to avoid fixing wrong alarms. After that, we collect program variables that can be utilized to synthesize patches by variable dependency analysis. During this procedure, we also collect the return locations of each function as the candidate fix locations. With this lightweight static analysis, HAMER is able to obtain enough information to fix the buggy program. We then gather the test cases generated by the fuzzer which trigger or do not trigger the errors and use the component-based program synthesis [6] to try to generate patches from these variables and test cases. Finally, we utilize the fuzzer to detect the current fixed program, and if the repair is erroneous, we collect the test case that triggers the errors and repeat our repair method until the error is resolved or timeout. This strategy ensures that the patches generated by HAMER can repair current errors without introducing new ones. evaluation and conclusion

Contributions. This paper makes the following contributions:

- We present a new technique for repairing memory errors based on hybrid analysis. Our approach collects the relevant information using a lightweight static analysis, then repeatedly synthesizes the patch and tests it with fuzzer.
- We present HAMER [1], a memory error repair tool that implements the proposed

[1] https://github.com/QIANZECHANG/MyResearch(仮)

approach.

Chapter 2

Background

HAMER detects bugs using existing bug detection tools. In this section, we will introduce the static analyzer (Infer [1]) and fuzzer (LibFuzzer [2]) that HAMER uses.

2.1 Infer

Infer [2] is a static program analyzer for Java, C, and Objective-C, written in OCaml, developed by Facebook. At present Infer is tracking problems caused by null pointer dereferences and resource and memory leaks, which cause some of the more important problems. Infer is a high-performance static analyzer with high scalability and efficiency, and it is widely used by programmers and researchers. However, like other static analyzers, Infer is hard to handle some issues, such as indirect call and alias, which causes Infer to provide false-negative and false-positive alarms. As a result of this shortcoming, automated repair tools may waste time on the wrong alarms (false-positive) and have no way to fix the errors that are not discovered (false-negative).

2.2 LibFuzzer

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine. LibFuzzer is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entry point (target function); the fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage. The code coverage information for libFuzzer is provided by LLVM's SanitizerCoverage [3] instrumentation.

LibFuzzer uses the information provided by the AddressSanitizer [11] [4] to determine if an error has happened within the detected code coverage when detecting memory errors. When the program allocates memory, the allocated space is marked, and the behavior is verified for legality each time the space is accessed. These memory spaces are checked at the end of the program to see if they have been freed.

Because LibFuzzer can only fuzz test one function argument at a time, we must test each argument independently for functions with multiple arguments. LibFuzzer may fail to mutate the inputs that go into all paths for some functions with complex input. Furthermore, LibFuzzer will stop and report errors anytime it triggers the error, thus if a function has several errors, LibFuzzer cannot ensure that all of them will be detected at the same time. The code below, for example, has two memory leaks.

[1] https://fbinfer.com/
[2] https://llvm.org/docs/LibFuzzer.html
[3] https://clang.llvm.org/docs/SanitizerCoverage.html
[4] https://clang.llvm.org/docs/AddressSanitizer.html

p1 = malloc(1);
if (a == 5) p2 = malloc(1);

The ideal situation would be for LibFuzzer to input *a=5* and trigger both errors. But in most cases, LibFuzzer inputs other values only triggers the memory leak of *p1* and then exits. As a result, LibFuzzer may not be able to detect all of the errors in a function at the same time (although it has the ability to do).

2.3 SAVER

Chapter 3

Overview

We illustrate key features of HAMER and how it works. Figure 3.1 depicts a high-level overview of the HAMER pipeline.

First, we check the program with the static analyzer Infer, and then we manually collect the functions that contain Infer alarms. Following that, these functions are fuzzed by LibFuzzer to detect true errors. For those functions on the error path, we collect the variables that are dependent on the function arguments and stub them. Dependent variables will be used to synthesize the conditional of the patch. After that, we instrument all of the dependent variables and run the program via LibFuzzer to collect the tests that do or do not trigger the error. Finally, we use a simplified component-based program synthesis (simp-CBPS) to generate patches. Because the quality of the test suite affects the quality of the patch, we use LibFuzzer to check the current fixed code again, and if it has not been fixed or if a new error occurs, we collect the tests that cause the error or insert the patch in a different location. This strategy ensures that the patches generated by HAMER are correct and do not introduce new errors. In the following paragraphs of this section, we will use two motivating examples to demonstrate the workflow and characteristics of HAMER.

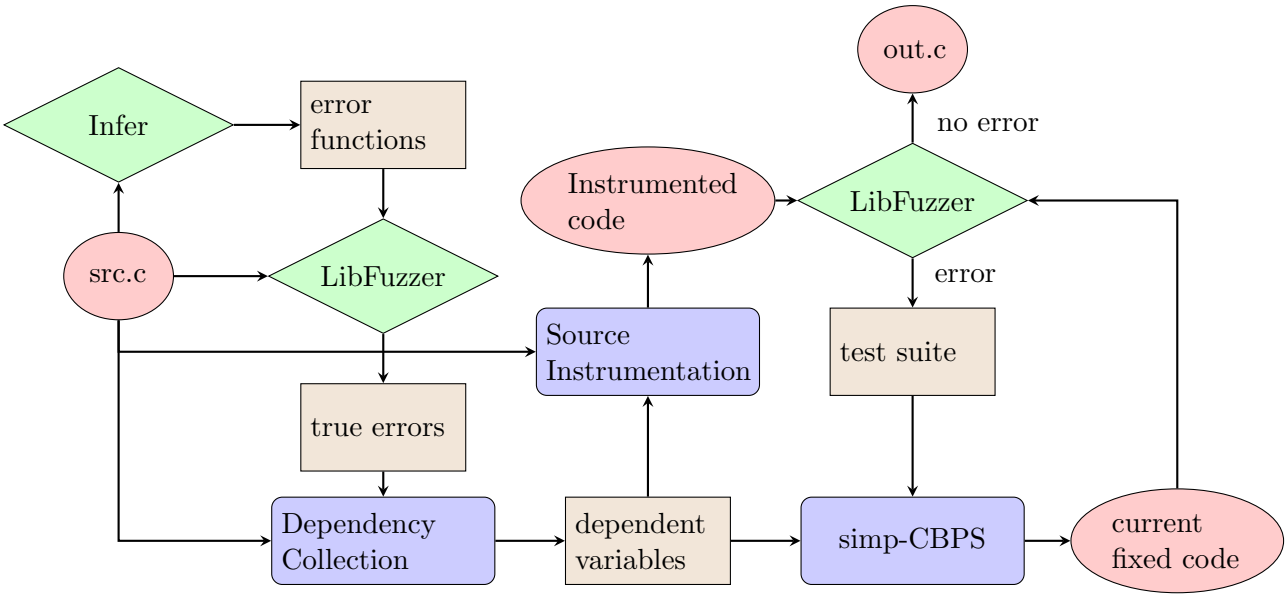

Figure 3.1: HAMER pipeline

```c
typedef struct N{
  int v;
}node;

node *new_node1(int a){
  node *n=(node*)malloc(sizeof(node));
  n->v=a;
  return n;
}

node *new_node2(int a){
  node *n=(node*)malloc(sizeof(node));
  n->v=a*a;
  return n;
}

int func(int a){
  node* (*p[])()={new_node1, new_node2};
  node *x;
  node *y=(node*)malloc(sizeof(node));//o2
  x=(*p[0])(a);//o0
  if(a<5){
    x=(*p[1])(a);//o1
  }
  x->v=10;
  return 0;
}
```

(a) o0, o1, o2 occur memory leak

```c
int func(int a){
  node* tmp_o0;
  node* tmp_o2;
  int tmp_a = a;
  node* tmp_o1;
  node* (*p[])()={new_node1, new_node2};
  node *x;
  node *y=(node*)malloc(sizeof(node));
  tmp_o2 = y;
  x=(*p[0])(a);
  tmp_o0 = x;
  if(a<5){
    x=(*p[1])(a);
    tmp_o1 = x;
  }
  x->v=10;
  if(tmp_a<=4)free(tmp_o1);
  free(tmp_o2);
  free(tmp_o0);
  return 0;
}
```

(b) HAMER-generated patch

Figure 3.2: Motivating Example 1: Infer false-negative alarm

```c
typedef struct N{
  int* p1;
  int* p2;
}node;

int func(int a){
  node x;
  x.p1=(int*)malloc(4);//o0
  x.p2=(int*)malloc(4);//o1
  free(*(&x.p1+1));
  free(x.p1);

  return 1;
}
```

(a) No memory leak

```c
typedef struct N{
  int* p1;
  int* p2;
}node;

int func(int a){
  node x;
  x.p1=(int*)malloc(4);//o0
  x.p2=(int*)malloc(4);//o1
  free(*(&x.p1+1));
  free(x.p1);
  free(x.p2);
  return 1;
}
```

(b) SAVER-generated patch

Figure 3.3: Motivating Example 2: Infer false-positive alarm

3.1 Motivating Example 1

This buggy code has three error heap objects, denoted by *o0*, *o1*, and *o2*, as shown in Figure 3.2a. First, we use Infer to detect this code, obtaining the following result:

Object allocated at line 20 is unreachable at line 20.

Because static analyzers like Infer have a difficult time resolving issues like indirect calls, they can only detect the memory leak of *o2*. After we received the Infer results, the error function was detected again by LibFuzzer, and all errors were successfully detected. For example, for *o1*, we can get the fuzzing result shown below:

in malloc ../a.out
in new_node2 ../src.c:12:18
in func ../src.c:23:7

We can get all of the functions on the error path using LibFuzzer. Obviously, the correct fix location could be in any of the functions, so we collect the variables that are dependent on the function argument, and we also collect both the heap object information and the return location of each function during this static analysis.

Table 3.1: Instrumentation result of o1

func	new_node2		error
a	a	n->v	
0	0	0	1
5	5	25	0
6	6	36	0
8	8	64	0

Following that, we instrument all of the dependent variables and run the source instrumented code through Libfuzzer to collect dynamic values for each dependent variable. Table 3.1 displays the results of o1's collection. The *error* column indicates whether or not memory leak occurred at the current value, with 1 indicating that it did and 0 indicating that it did not. Table 3.1 shows that *o1* leaks when the variable *a* in the function *func* is less than or equal to 4. We can synthesize the ideal patch using simp-CBPS, which is:

```
if(a<=4)free(o1);
```

Finally, we use LibFuzzer to detect the patched code, and if it fixes the current bug, we keep the patch and fix other bugs until all bugs are fixed or time out. If the current patch does not fix the bug or causes a new bug, we try to insert the patch into another location or collect new tests to synthesize a new patch. For example, we can also synthesize patches like *if(a¡=3)free(o1);* using the Table 3.1 results. With Libfuzzer, we can see that when *a=4*, *o1* still occurs memory leak. We will start by trying alternative fix locations, however, because the function *func* only has one return place, we can only collect new tests to synthesize a new patch. We add *(a:4, error:1)* to the test suite and then use the synthesizer to generate a new patch. It's self-evident that the improved test suite enabled us to obtain the correct patch. We will utilize temporary variables to save the heap object and variables in the conditional when we apply the patch. This step is necessary to prevent these variables from changing between the allocation location and patch insertion location.

Although it is possible to fix the *o1* memory leak by inserting a patch into function *new_node2*, the use-after-free problem will occur if the memory is freed too early due to the use of *x-¿v* at line 25 of the function *func*.

3.2 Motivating Example 2

We briefly discussed how HAMER uses fuzzing (LibFuzzer) to detect and fix errors that are not noticed by the static analyzer (Infer) in Motivating Example 1. Similarly, HAMER can avoid attempting to resolve false alarms. In Motivating Example 2, line 10 frees *o1* in some other way, but Infer misses it, so it assumes *o1* occurs memory leak. SAVER [5] also ignores the fact that this is a false alarm and generates the incorrect patch, resulting in *double-free*. HAMER utilizes LibFuzzer to dynamically detect code, and LibFuzzer does not report issues for Motivating Example 2, therefore HAMER saves time trying to generate a fix for the wrong alarm. However, depending on the complexity of the function argument and the fuzzer's mutation strategy, the fuzzer may fail to visit the error path.

Chapter 4

Approach

In this section, we describe our approach in detail, explaining what technical issues arise and how we address them. There are three major issues to consider:

- It is difficult for LibFuzzer to directly collect the high-quality test suite. How can HAMER synthesize the correct conditional?
- How does HAMER choose the correct fix location(s) when a function has multiple return places?
- How does HAMER deal with several memory errors in a single function?

HAMER will solve these issues by using LibFuzzer to constantly check the patched code. In section 4.5, we will go over our repair algorithm in detail. Until then, we will describe how HAMER gathers the data required to fix the errors.

4.1 Patch Template

The purpose of this research is to fix temporal memory errors such as memory leaks, not spatial memory errors such as buffer overflow. The most common solution for temporal memory errors is to free the memory at the correct location. As a result, the following patch template can fix the most of temporal memory errors.

$if(cond)free(ob)$;

We can free the specified heap object under the specific condition by inserting a conditional deallocator. Hence, in order to generate the correct patch, we should know the following three details:

(1) conditional *cond*

(2) error heap object *ob*

(3) fix location

We will present in detail how we generated the correct patch in the following subsection.

4.2 Error Detection

HAMER uses a static analyzer to find functions that may have memory errors and then uses fuzzer to detect the real errors in those functions. The purpose of applying the static analyzer is to improve HAMER's scalability, as the fuzzer always takes a long time to detect errors. Using the static analyzer to pick out candidate functions prevents fuzzer

from wasting time in locations where it is not necessary. But obviously, HAMER will not be able to fix errors in functions that are not provided by the static analyzer. So for shortcode, we can just simply use fuzzer to detect errors. After error detection, we can obtain a set E that contains the details of each error reported by fuzzer, similar to the report shown in Motivating Example 1.

4.3 Dependency Collection

We collect the error paths and error heap objects for each element in the set E separately after obtaining the error report E. We can organize the information of each error from the error report to produce the set EP using the error paths and coordinates provided by LibFuzzer.

$$EP = \{get\_path(e) | \, e \epsilon E\} \tag{4.1}$$

In Motivating Example 1 (Figure 3.2), LibFuzzer gives the error report of $o1$, and we can obtain the error path of $o1$:

{*funcname: func, coord: src.c:23:7, next:*
{*funcname: new_node2, coord: src.c:12:18, next:*
{*funcname: malloc*}}}

It is worthy to note that the *coord* of each function is the error location inside the function, not the function's coordinate. All error paths, obviously, will end with an allocation function, and the coordinates of the allocation function will be kept in the previous node's *coord*, which we will use to localize the error heap object.

After organizing the error report, we collect the dependent variables in each function on the error path, which are utilized to synthesize the conditional. Fuzzer tries to trigger the function's error by inputting different data, so all the variables in the function that are dependent on the argument could be utilized to synthesize the patch's conditional. These dependent variables are collected using *def-use* chain, and their names, types, and coordinates are saved. During this static analysis, we also gather the return location of each function, as well as the name and type of the error heap object.

As we present in the Background, fixing the temporal memory error requires inserting the deallocation at the correct location. A memory leak will occur if allocated memory space is not freed at the end of the program, hence it is critical to free allocated memory before it is unreachable. Most of the existing memory error repair techniques [5, 7] are based on static analysis that collects heap-related behavior, so they can insert the patch at the right location. However, the tradeoff is that it requires a high overhead, and static analysis is tough to deal with some problems such as indirect call, which may make repair tools fail to generate a patch or generate a wrong patch. HAMER aims to collect the essential information for fixing errors via lightweight static analysis. There are three major reasons why allocated memory cannot be accessed: (1) it is freed, (2) no pointer points to it, and (3) the current function exits. Memory leaks will not occur if it is properly freed. For the second reason, we will use a temporary variable to save the pointer to the error memory when it is allocated, as we will explain in detail in subsection 4.5. Hence, we simply need to think about the third reason. A function may exit in two ways: by *return* or by exiting automatically at the end of the function. As a result, we just need to collect all of each function's *return* locations, as well as the location of the function tail, and use them as the candidate fixing location.

We extract the name and type of error heap objects at the location detected by the fuzzer. Its type is mainly determined by three factors: (1) declaration type, (2) casting

type, and (3) type in the *sizeof*. For instance, the following code demonstrates how developers use *malloc* for memory allocation:

int p = (int*) malloc (sizeof (int))* ;

We only need one of these to determine the type of the heap object, but there are indeed cases where all three are not written, such as when type is defined in a structure and then cast and sizeof are not used:

x->v = malloc (4) ;

In such cases, HAMER is unable to obtain their type, however, this is not a weakness of our approach. HAMER employs lightweight static analysis that can handle most cases. If we want to achieve higher performance, we can employ a more precise type analysis, but the tradeoff is higher overhead.

Finally, we formalized the dependency information *Dep* as follows:

$$Dep = \{(get\_dep(path),\ get\_ret(path),\ get\_ob(path))|\ path \epsilon EP\} \tag{4.2}$$

4.4 Source Instrumentation

We apply source instrumentation on the dependent variables after obtaining them in order to collect their dynamic values and build a test suite. A common test suite consists of several $input - output$ pairs; however, because the goal of our technique is to fix memory leaks, each *input* corresponds to an *output* that indicates whether or not the *input* will trigger the error; if it does, the output is 1, otherwise, it is 0. For example, our instrumented code for the variable *a* of the $func$ function in Figure 3.2a is as below:

fprintf(stderr,"instrument: (line: 17) a : %d\n",a);

We record the locations of the variables to identify their values because the same variables will have different values at different locations. Because the results of LibFuzzer are sent to *stderr*, we also send the results of the instrumentation to *stderr* and combine them with the results of LibFuzzer.

Our approach's overall concept of source instrumentation and dynamic values collection is shown in Algorithm 1. We create a new set $SynInf$ to contain the information about fixing each error. We first define a set *dep_var* to store all the dependent variables in *Dep* and for different errors, we save the dependent variables Var, return location Ret, and error heap object information *ob* to $SynInf$. To get enough data, we use LibFuzzer to run the instrumented code 10 times and collect data for different errors. If the current execution results trigger an error, we set the *output* of the triggered error's dependent variable to 1, otherwise, we set it to 0.

4.5 Patch Generation

In this section, we will present our repair algorithm and demonstrate how our repair algorithm solves the major issues we mentioned before.

4.5.1 Component-based Program Synthesis

Since we only need to synthesize the conditional of deallocation, we use a simplified version of component-based program synthesis [6] (simp-CBPS). In simp-CBPS, a component is a variable, a constant, or an operator. simp-CBPS uses these user-given components to generate code that satisfies the test suite.

For example, we use the following components to synthesize code that satisfies the test suite in Table 4.1.

Algorithm 1 Source Instrumentation Result Collection

Input: src, Dep
Output: SynInf

```
dep_var ← ∅
SynInf ← ∅
err_id ← 0
for (Var, Ret, ob) ∈ Dep do
    dep_var ← dep_var ∪ Var
    SynInf[err_id].add((Var, Ret, ob))
    err_id ← err_id + 1
end for
inst_code ← Instrument(src, dep_var)
i ← 0
while i < 10 do
    res ← Fuzz(inst_code)
    for err_id = 0 to |Dep| − 1 do
        for var ∈ dep_var do
            if memory leak happen then
                SynInf[err_id][var].add((res[var], 1))
            else
                SynInf[err_id][var].add((res[var], 0))
            end if
        end for
    end for
    i ← i + 1
end while
```

variable: x
constant: c
operator: $*_1 < *_2$

Since $<$ is a binary operator, we can construct the expression $x < c$ and $c < x$ using the variable x and the constant c. Then we assign the value from Table 4.1 to get the logical formula below:

$x < c : (4 < c) \wedge (5 < c) \wedge \neg(6 < c) \wedge \neg(7 < c)$
$c < x : (c < 4) \wedge (c < 5) \wedge \neg(c < 6) \wedge \neg(c < 7)$

We have turned the program synthesis problem into *Satisfiability Modulo Theories* (SMT) by doing the above action. We solve the logical formula via the SMT solver. If a logical formula is unsatisfiable, it means that the present synthesized expression does not pass the test suite, indicating that it is not the expected expression. For example, the second logical formula is unsatisfiable, hence $c < x$ is not the correct expression. The first logical formula is satisfiable and the result is $c = 6$, so we can get the expected expression $x < 6$.

Table 4.1: Test suite

x	output
4	True
5	True
6	False
7	False

The quality of the test suite is the most critical part of using CBPS to synthesize expressions. CBPS will synthesize incorrect expressions if the test suite provided by the user is of poor quality. If Table 4.1 does not have $(x = 6, output = False)$, for example, we might obtain $x < 7$.

4.5.2 Repair Algorithm

The input of our repair algorithm (Algorithm 2) is the source code and the collected information $SynInf$, and the output is the fixed code generated by HAMER. We use a *queue* to keep the number of each error and repair them one by one. When repairing, we first pop an error number from the *queue* and then get the corresponding information from $SynInf$. After that, we synthesize the conditional of the patch using $simp-CBPS$. Because all the functions on the error path have the potential to become the fix location, $simp - CBPS$ synthesizes all satisfiable patches, and the fix location and heap object information corresponding to the patch are gathered and saved to *cur_patches*.

After that, we synthesize the patch and use the function Fix (Algorithm 3) to insert it in the right location. If the currently generated patch fails to fix the error, we gather the test that triggers the error and append it to the test suite. For example, when HAMER fixes the memory leak of $o1$ in Figure 3.2a of Motivating Example1, it's difficult to collect enough tests directly, thus the next two incorrect patches could be synthesized:

(1) $if(a <= 3)free(o1);$
(2) $if(a <= 5)free(o1);$

If $simp-CBPS$ does not get the test $(a : 4, error : 1)$, it will synthesize the conditional of the first wrong patch, and if it does not get the test $(a : 5, error : 0)$, it will synthesize the conditional of the second wrong patch. For the first patch, we can get that memory leak will happen when $a = 4$ via LibFuzzer, so we add $(a : 4, error : 1)$ to the test suite. For the second patch, when $a = 5$, since $o1$ is not defined yet, free it is undefined behavior, so we add $(a : 5, error : 0)$ to the test suite. Similarly, if a test triggers a double free, we will add this test and $(error : 0)$ to the test suite, indicating that this heap object has already been deallocated at this test.

We keep the patch of the current error if this error is fixed, and we clear the test suite in $SynInf$ if there still have errors to be fixed. The goal of this step is to save space and speed up the SMT solver computation. $SynInf$ takes up a lot of space because the test suite is constantly updated. The quality of the gathered test suite will also be affected if a function has several errors. We can recollect the test suite when an error is fixed to receive a higher-quality test suite and save space. If the current error is not fixed, we add the error number to the queue and try to fix it later.

To ensure that our algorithm terminates at the right time, we use two methods. The first occurs in line 9. If $simp-CBPS$ synthesizes the same patch as last time, indicating that this error is difficult to fix in the current situation. In most cases, we are unable to collect a high-quality test suite due to multiple errors in the function, so we give up trying to fix this error for the time being and return to it after fixing other, much simpler errors. The second is that we create an $unfixederror$ variable to keep track of errors that we try to fix but can not. Because there may sometimes be several errors that HAMER cannot solve, we stop HAMER if none of the remaining errors are fixed after HAMER tries to fix them.

4.5.3 Function Fix

It is critical to insert the patch in the correct location in order to fix temporal memory errors. Function Fix to determine if the fix location is correct. We test the patch and the fix location of each function on the error path one by one. Fuzzer checks a patch when it is

Algorithm 2 Repair Algorithm

Input: src, SynInf
Output: fixed_code

```
1: fixed_code ← src
2: queue ← {0, 1, ..., |SynInf| − 1}
3: unfixederror = 0
4: while |queue| ≠ 0 and unfixederror ≠ |queue| do
5:    err_id ← queue.pop
6:    err_inf ← SynInf[err_id]
7:    repeat
8:       cur_patches ← simp-CBPS(err_inf)
9:       if cur_patches same as last time then
10:         break
11:      end if
12:      cur_code ← Fix(cur_patches, fixed_code)
13:      if not fixed then
14:         err_inf ← Update(err_inf)
15:      end if
16:   until timeout or error fixed
17:   if current error fixed then
18:      fixed_code ← cur_code
19:      unfixederror ← 0
20:      if queue is not empty then
21:         SynInf ← Clean(SynInf)
22:         SynInf ← Update(SynInf)
23:      end if
24:   else
25:      queue.add(err_id)
26:      unfixederror ← unfixederror + 1
27:   end if
28: end while
```

inserted into a fix location. If the patch introduces a new error in the existing fix location, it is clear that the fix location is incorrect, and we should try a different fix location. If no new errors are raised but the existing error remains unfixed, it is possible that the current fix location only partially fixed the error; in that case, we save the patch inserted at the current fix location and try other fix locations. The following code, for example, has two returns, and we must apply the patch to both of them to fix the error.

```
p=malloc(1);
if(c){
  use(p);
  return 0;
}else{
  use(p);
  return 1;
}
```

The memory leak was not fixed when we inserted the patch before the first *return*, so we kept the current patch and inserted the patch before the second *return*, and then the error was correctly fixed.

Algorithm 3 Function Fix

Input: cur_patches, code
Output: fixed_code

```
1: test_code ← code
2: for (cur_patch, Ret) ∈ cur_patches do
3:    for retloc ∈ Ret do
4:       test_code ← InsertPatch((cur_patch, retloc))
5:       Fuzz(test_code)
6:       if new error occurs then
7:          test_code ← code
8:          continue
9:       else if same error occurs then
10:         continue
11:      else
12:         return test_code
13:      end if
14:   end for
15: end for
16: return ∅
```

4.5.4 Temporary Variable

We use temporary variables to store the variables used in the patch conditional and the error heap object while inserting patches. Because the values of these significant variables may modify between the error source and the fix location, analyzing these changes requires complex static analysis. To address this issue without increasing the complexity of our algorithm, we use temporary variables to store the values of these variables, so that even if their values change, the patches generated by HAMER are still correct (e.g., Figure 3.2b).

Chapter 5

Evaluation

We evaluate the effectiveness and efficiency of HAMER and answer the following research questions.

RQ1 Compared with state-of-the-art automated memory error repair tool SAVER, what is the overall effectiveness of HAMER?

RQ2 Can HAMER address the three major issues we mentioned in Chapter 4?

RQ3 How efficient is HAMER in using lightweight static analysis?

5.1 Implementation

We have implemented our approach in a tool named ExtractFix, whose pipeline is shown in Figure 3.1. HAMER detects the memory leak in the program using existing vulnerability detection tools. To detect the real errors, HAMER first uses a static analyzer to detect the program, then uses a fuzzer to detect static analyzer alarms. The main role of the static analyzer is to filter out the problematic functions in the program so that the fuzzer does not waste time on other parts. We directly use a fuzzer to verify the code in our experiments because we utilize short code examples to test the efficiency of HAMER. We use LibFuzzer to perform the fuzz testing. Since LibFuzzer may not detect all of the errors, we fuzz code 10 times each time. If LibFuzzer does not detect an error within 5 seconds for one execution, the code is judged no error (or fixed). The input data format is set to an integer of two digits or less. HAMER is written in 1000 lines of python3 code. For syntactic analysis of c, we use the pycparser module [1]. In our implementation of simp-CBPS, we use the Z3 [3] python module [2] as the SMT solver and implemented six operators ($!=, ==, <, <=, >, >=$) as components.

5.2 Experimental Setup

To analyze HAMER, we synthesized 10 codes. Because HAMER's scalability is currently limited, and memory leaks in real-world projects can not properly reflect HAMER's properties, we gave up utilizing real-world projects to evaluate HAMER. These codes were synthesized from six different directions, as shown in Table 5.1. To fuzz the code, LibFuzzer requires an entry point, which we manually added to the code as shown below.

[1] https://github.com/eliben/pycparser
[2] https://github.com/Z3Prover/z3

```
int LLVMFuzzerTestOneInput(char *data, int size) {
  ConvertChartoInt(data);
  EntryPoint(data);
  return 0;
}
```

HAMER's static analysis and fuzzing time were also observed during the experiment. Since the repair process of HAMER was not absolutely the same every time (depending on the test generated by LibFuzzer), we run HAMER ten times for each code and then calculate the average repair time. We installed SAVER directly from their github [3] (Infer version: v0.15.0-821a8db). Because SAVER relies on Infer's static analysis results, we must consider Infer's execution time when calculating SAVER's execution time. Table 5.2 shows the final experimental results.

Table 5.1: Characteristics of synthesizing code. FN and FP denote that the code contains Infer's false-negative and false-positive alarms. EP denotes that the code contains the error with a long error path. ME denotes that the code contains multiple error heap objects. MR denotes that the code has multiple returns. CF denotes that the code has a complicated control flow.

Syn	LoC	Feature
test1	12	CF
test2	9	MR
test3	23	EP, FN
test4	11	FP, FN
test5	15	EP, ME, FN
test6	24	EP, ME, MR, FN
test7	9	MR, CF, FN
test8	26	EP, ME, FN
test9	24	EP, ME, FN
test10	15	ME, CF

5.3 Experimental Results

5.3.1 Compared with state-of-the-art automated memory error repair tool SAVER, what is the overall effectiveness of HAMER?

There are 22 memory leaks in the 10 test codes we synthesized. Infer found 7 memory leaks successfully, missed 15 memory leaks, and reported a false alarm. Because SAVER relies on infer's alarms to repair the code, it has no opportunity to fix the 15 unreported errors (false-negative). SAVER generated 4 correct patches and two wrong patches for the seven alarms correctly reported by Infer, and it failed to identify Infer's false alarm and generated a wrong patch.

All 22 memory leaks were detected via LibFuzzer. For test8, LibFuzzer may not be able to detect all of the errors, also making it difficult to verify whether the patch generated by HAMER is correct or not. In **RQ2**, we will go over the details of test8. HAMER can generate all correct patches for other test codes and does not provide any erroneous patches. HAMER's average repair time is 9.22 seconds, although it is clear that the majority of that time is spent by LibFuzzer checking whether the patches are correct. We also set the fuzzing timeout to 5 seconds, which implies that any properly repaired test

[3] https://github.com/kupl/SAVER_public

Table 5.2: Evaluation result of HAMER and SAVER. ML denotes the number of memory leaks. T, FP, and FN denote the number of true, false-positive, and false-negative alarms detected by Infer. Fuzzer denotes the number of memory leaks detected by LibFuzzer. ✓ and × report the correct and wrong patches generated by HAMER and SAVER. SA, Fuzz, and Total report the static analysis time, fuzzing time, and the total fix time.

		Infer			SAVER			HAMER			
Syn	ML	T	FP/FN	sec	✓	×	sec	Fuzzer	✓	×	SA/Fuzz/Total
test1	1	1	0/0	0.68	0	0	0.05	1	1	0	0.01/7.01/7.94
test2	2	2	0/0	0.13	2	0	0.07	2	2	0	0.01/7.27/8.42
test3	2	0	0/2	0.36	-	-	-	2	2	0	0.04/7.71/9.14
test4	1	0	1/1	0.15	0	1	0.03	1	1	0	0.01/7.06/7.90
test5	2	0	0/2	0.14	-	-	-	2	2	0	0.03/7.71/9.10
test6	3	0	0/3	0.28	-	-	-	3	3	0	0.03/8.03/9.60
test7	2	1	0/1	0.14	0	1	0.05	2	2	0	0.01/7.43/8.61
test8	4	0	0/4	0.49	-	-	-	3-4	0-3	0-1	0.05/4.17/7.17
								4	4	0	0.05/9.94/13.17
test9	3	1	0/2	0.42	1	0	0.05	3	3	0	0.04/8.91/11.29
test10	2	2	0/0	0.42	1	1	0.07	2	2	0	0.02/7.67/9.04
Total/Ave	22	7	1/15	0.32	4	3	0.05	21-22	18-22	0-1	0.03/7.54/9.22

codes will take 5 seconds to confirm whether or not the patch is correct via LibFuzzer. We have already discussed the issues of indirect call and Infer's false-positive alarm in Chapter 3, so we will not go over it again here.

Infer failed to detect two memory leaks in test5 (shown in Figure 5.1) because it cannot effectively deal with the alias. It made SAVER have no chance to fix errors (although this is easy for SAVER). Similar problems appear in test3,4,6,8,9 when memory is dynamically allocated at another function and then returned to the current function's pointer. LibFuzzer, on the other hand, tracks the memory situation via AddressSanitizer, which instruments all dynamic memory allocation, allowing it to quickly detect any memory leaks in codes with simple control flow. Because *o1* uses *o0*, if *o0* is freed before *o1*, use-after-free will occur. Because HAMER stores error heap objects in temporary variables, we don't need to worry about the order of deallocation.

```
1  typedef struct N{
2    struct N *next;
3    int v;
4  }node;
5
6  node *new_node(int a){
7    node *n=(node*)malloc(sizeof(node));
8    n->next=NULL;
9    n->v=a;
10   return n;
11 }
12
13 int func(int a){
14   node *x=new_node(a);//o0
15   x->next=new_node(a+1);//o1
16   return 0;
17 }
```

(a) o0, o1 occur memory leak

```
13 int func(int a){
14   node* tmp_o0;
15   node* tmp_o1;
16   node *x=new_node(a);
17   tmp_o0 = x;
18   x->next=new_node(a+1);
19   tmp_o1 = x->next;
20   free(tmp_o1);
21   free(tmp_o0);
22   return 0;
23 }
```

(b) HAMER-generated patch

Figure 5.1: test5

```
1  typedef struct N{
2    int* p1;
3    int* p2;
4  }node;
5
6  int func(int a){
7    node x;
8    x.p1=(int*)malloc(4);//o0
9    int* n;
10   if(a<5){
11     n=x.p1;
12   }else{
13     n=malloc(4);//o1
14   }
15   return 1;
16 }
```

(a) o0, o1 occur memory leak

```
6  int func(int a){
7    node x;
8    x.p1=(int*)malloc(4);
9    int* n;
10   if(a<5){
11     n=x.p1;
12   }else{
13     n=malloc(4);
14   }
15   free(x.p1);
16   free(n);
17   return 1;
18 }
```

(b) SAVER-generated patch

```
6  int func(int a){
7    int* tmp_o0;
8    int tmp_a = a;
9    int* tmp_o1;
10   node x;
11   x.p1=(int*)malloc(4);
12   tmp_o0 = x.p1;
13   int* n;
14   if(a<5){
15     n=x.p1;
16   }else{
17     n=malloc(4);
18     tmp_o1 = n;
19   }
20   if(tmp_a>=5)free(tmp_o1);
21   free(tmp_o0);
22   return 1;
23 }
```

(c) HAMER-generated patch

Figure 5.2: test10

In test10 (shown in Figure 5.2), although Infer detected two memory leaks, SAVER generated a correct patch and a wrong patch. SAVER can not fix multiple errors at the same time. If only fix *o0* without considering *o1*, the patch generated by SAVER at line10 is correct, but if the patch of *o1* at line11 is added, double free will occur. HAMER can fix multiple errors at the same time and check whether the generated patches will cause new errors, so it can fix test10 correctly

HAMER has a higher repairability than SAVER. HAMER has more opportunities to repair more memory leaks, as well as better handling of issues like multiple errors, indirect calls, and alias, and is less likely to generate erroneous patches.

5.3.2 Can HAMER address the three major issues we mentioned in Chapter 4?

To synthesize correct conditional, HAMER continuously updates the test suite until the correct patch is generated or timeout. For *o1* in test10, sometimes LibFuzzer can not directly provide a significant test (e.g.,$(a : 5, error : True)$), causing HAMER to generate

the wrong conditional $a >= 4$. However, HAMER checked the patch via LibFuzzer and found that the memory leak of *o1* still existed when $a = 5$, so HAMER added this test to the test suite and generated the correct patch at last.

HAMER tries to insert the patch to each candidate fix location and verifies the patch via LibFuzzer. Test2, 6, 7 all have multiple returns, and HAMER successfully fixed them all, so HAMER can deal with the case of multiple returns in a function.

HAMER will repair each error one by one via HAMER's repair algorithm shown in Chapter 4. HAMER is able to reach 100% repairability for the function which has two error heap objects (e.g., tests 5, 6, 9, 10). Due to the feature of LibFuzzer, LibFuzzer stops and outputs results when it detects the error on the current explored path, so if multiple error heap objects exist on multiple different paths, LibFuzzer will be unable to detect all the errors at the same time, causing HAMER failed to generate patches. However, we can solve this problem by improving our repair algorithm, which we will discuss in the Future Work in Chapter 6.

Figure 5.3 shows the correct and incorrect patches generated by HAMER for test8. Test8 has four error heap objects (*o0, o1, o2, o3*). In most cases, LibFuzzer can detect all of the errors and give the necessary tests, enabling HAMER to successfully fix test8 as shown in Figure 5.3b. However, due to a large number of errors, LibFuzzer may not be able to detect all of them. For example, test8 has two paths($a < 5, a >= 5$), as shown in Figure 5.3c, and executing either path will trigger the memory leaks of *o0, o1, o3*. Although we initially set LibFuzzer to run 10 times in order to detect all errors, there were sometimes that the produced tests were all $>= 5$, so that the path of $a < 5$ has never been entered and the memory leak of *o2* has never been triggered. Although HAMER could generate the correct patch for *o0, o1, o3*, when checked by LibFuzzer, each patch will cause the memory leak of *o2* and HAMER will determine that the current patch is not correct since it will cause a new error.

5.3.3 How efficient is HAMER in using lightweight static analysis?

```c
typedef struct N{
  struct M *m;
  int v;
}node;

typedef struct M{
  struct M *next;
  int k;
}field;

node *new_node(int a){
  node *n=(node*)malloc(sizeof(node));
  n->m=(field*)malloc(sizeof(field));
  n->m->next=NULL;
  return n;
}

int func(int a){
  node *x=new_node(a);//o0, o2
  node *y=new_node(a);//o1, o3
  x->v=a;
  x->m->k=a+1;
  y->v=a;
  y->m->k=a+1;
  if(a<5){
    free(x);
  }
  return 0;
}
```

(a) o0, o1, o2, o3 occur memory leak

```c
int func(int a){
  field* tmp_o0;
  node* tmp_o1;
  int tmp_a = a;
  node* tmp_o2;
  field* tmp_o3;
  node *x=new_node(a);
  tmp_o0 = x->m;tmp_o2 = x;
  node *y=new_node(a);
  tmp_o1 = y;tmp_o3 = y->m;
  x->v=a;
  x->m->k=a+1;
  y->v=a;
  y->m->k=a+1;
  if(a<5){
    free(x);
  }
  free(tmp_o3);
  if(tmp_a>=5)free(tmp_o2);
  free(tmp_o1);
  free(tmp_o0);
  return 0;
}
```

(b) HAMER-generated correct patch

```c
int func(int a){
  node *x=new_node(a);
  node *y=new_node(a);
  x->v=a;
  x->m->k=a+1;
  y->v=a;
  y->m->k=a+1;
  if(a<5){
    free(x);
  }
  return 0;
}
```

(c) LibFuzzer only detect 3 errors at first, so HAMER failed to generate patch

```c
int func(int a){
  field* tmp_o0;
  node* tmp_o1;
  int tmp_a = a;
  node* tmp_o2;
  node *x=new_node(a);
  tmp_o0 = x->m;tmp_o2 = x;
  node *y=new_node(a);
  tmp_o1 = y;
  x->v=a;
  x->m->k=a+1;
  y->v=a;
  y->m->k=a+1;
  if(a<5){
    free(x);
  }
  if(tmp_a>=4)free(tmp_o2);
  free(tmp_o1);
  free(tmp_o0);
  return 0;
}
```

(d) LibuFuzzer failed to verify the wrong patch, make HAMER generate wrong patch and failed to generate the patch of o3

Figure 5.3: test8

Chapter 6

Conclusion

6.1 Limitations

6.2 Future Work

6.3 Conclusion

Chapter 7

Related Work

CPR [12], SemFix [10], DirectFix [8], Angelix [9], Extractfix [4], Getafix [1], SAVER [5], Memfix [7]

References

[1] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[2] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 459–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Trans. Softw. Eng. Methodol.*, 30(2), feb 2021.

[5] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. Saver: Scalable, precise, and safe memory-error repair. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 271–283, 2020.

[6] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 215–224, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Memfix: Static analysis-based repair of memory deallocation errors for c. ESEC/FSE 2018, 2018.

[8] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458, 2015.

[9] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[10] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.

[11] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical*

Conference (USENIX ATC 12), pages 309–318, Boston, MA, June 2012. USENIX Association.

[12] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 390–405, 2021.