**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. The program's heap-related behaviors, such as heap object, error source and sink, play a critical role in memory error repair, so how to detect the errors and collect the relevant information are the main tasks. 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 static analyzer may give wrong alarms which will affect the performance of the repair tools, and heap-related behavior analyzing 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 has high repairability and can be easily integrated with a fuzzer. 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 an 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 verify the generated patches to ensure they are correct. For the necessary information for synthesizing patches, HAMER uses existing lightweight static analysis techniques to collect it. Our preliminary experiment suggests that HAMER can fix the complicated memory errors effectively, which the existing static memory error repair technique can not repair.

# 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 leak detection techniques can report the locations where the memory error occurs. But fixing these errors needs not only the error heap objects but also the suitable fix location and the heap-related behaviors. Furthermore, the allocation location and the utilized location are usually passed between functions and it is often difficult for the developer to find the suitable points to deallocate the memory. If developers insert the wrong patch or insert the patch at the wrong point, it may cause more significant errors.

Existing memory error repair techniques [11, 16] are mainly based on static analysis. This is because fixing errors 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 [11] stores 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 detect memory errors efficiently by instrumenting the critical program points to observe the legality of program behavior during execution, it does not provide enough information to generate patches.

In this paper, we present **HAMER**, a hybrid analysis-based memory error repair technique. HAMER has high repairability and can be easily integrated with a fuzzer, which can detect memory error and verify the patches generated by HAMER. 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 trying to fix 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 [12] to try to generate patches from these variables and test cases. Finally, we utilize the fuzzer to verify 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 fixed or timeout. This strategy ensures that the patches generated by HAMER can repair current errors without introducing new ones. Our preliminary experiment suggests that HAMER can fix the complicated memory errors effectively, which the existing static memory error repair technique cannot repair. It also shows that our lightweight static analysis can collect enough information in a short time to assist HAMER to generate the correct patches.

**Contributions.** This paper makes the following contributions:

- We present a new technique for repairing memory errors based on hybrid analysis. We present an efficient repair algorithm that only needs lightweight static analysis to collect relevant information and makes flexible use of the fuzzer's result to generate and verify patches. Also, our technique can generate and insert conditional deallocation patches correctly and can fix multiple memory leaks at the same time.
- We present HAMER [1], a memory error repair tool that implements the proposed approach. HAMER has high repairability and can be easily integrated with a fuzzer.

[1] https://github.com/QIANZECHANG/HAMER

# Chapter 2

# Background

HAMER detects bugs using existing bug detection tools. In this section, we will introduce the static analyzer (Infer [3] [1]) and fuzzer (LibFuzzer [2]) that HAMER uses. We also introduce the component-based program synthesis (CBPS) [12] which we use to synthesize the patches.

## 2.1 Infer

Infer is a static program analyzer for Java, C, and Objective-C, written in OCaml, developed by Facebook. Infer is a high-performance static analyzer with high scalability and efficiency, and it is widely used by programmers and researchers. Infer automatically discovers specifications for the functions that can be proven to be memory safe using the theoretical notion of bi-abductive inference [4]. The bi-abductive inference can generate the specification of a function's heap-related behavior and can be directly used to analyze the heap situation whenever the function is called.

For example, in Figure 2.1, the $foo$ function dynamically allocates memory through the *alloc* function. Infer uses bi-abductive inference to approximate the procedures' heap behavior, so Infer does not determine that *alloc* occurs memory leak because *alloc* returns $p$, meaning that $p$ is not unreachable and the responsibility for freeing $p$ is transferred to $foo$. Infer will construct the specification of *alloc* using these heap features and use it to analyze other procedures when *alloc* is called. Therefore, in $foo$, $p$ gets the return of *alloc* and has not been freed until last, Infer determines that $foo$ occurs memory leak.

```
1 int* alloc(){
2   int* p=malloc(sizeof(int));
3   return p;
4 }
5
6 void foo(){
7   int* p=alloc();
8 }
```

Figure 2.1: Example of Infer

However, 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 opportunity to fix the errors that are not discovered (false-negative).

[1] https://fbinfer.com/
[2] https://llvm.org/docs/LibFuzzer.html

## 2.2 LibFuzzer

LibFuzzer can be described as an evolutionary fuzzing engine and is both in-process and coverage-guided. LibFuzzer relies on an entry point to feed a fuzzy input to the function, which is often referred to as the target function. LibFuzzer can then track to a certain area and aim to explore more areas, generating the variant form of the input data mentioned above. The advantage of this is that it can obtain maximum code coverage. To check the code coverage, the information provided by SanitizerCoverage [3] is used by LibFuzzer.

LibFuzzer uses the information provided by the AddressSanitizer [23] [4] and the LeakSanitizer [5] to determine if an error has happened within the detected code coverage when detecting memory errors. AddressSanitizer uses a shadow memory to encode the allocated space (address and size) and verifies the legality each time the space is accessed. AddressSanitizer instruments every heap-related behavior (such as allocation, deallocation, read, and write) during compile-time and checks the legality of the behavior that the memory is being read/written, by checking the shadow memory's data during program execution. While LeakSanitizer stores the allocation thread ID and the allocation stack itself each time memory is allocated and records the situation when memory is deallocated. At the very end of program execution, LeakSanitizer will go over the heap to find out such objects that have not been freed.

LibFuzzer mutates the data based on a corpus of input samples of the tested codes. In general, a good corpus is seeded with different sets of valid and invalid inputs of the detection code. The fuzzer generates random mutations based on the input samples from the corpus in use at the moment. If this random mutation triggers an error in the detection code and the source of this erroneous path is a previously undiscovered path, then this random mutation is saved to the corpus we have set. The advantage of LibFuzzer is that it can work without any initial seed. The weakness on the other hand is that if the library being examined requires complex and structured input, then LibFuzzer's efficiency is subsequently somewhat reduced.

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 take a long time or fail to mutate the inputs that can go into each path for the functions that need complex inputs. Furthermore, LibFuzzer will stop and report errors anytime it triggers the error, thus if a function has several errors in different paths, LibFuzzer cannot ensure that all of them will be detected at the same time. The code below, for example, has two memory leaks.

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

The ideal situation would be for LibFuzzer to input the value smaller than 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). Although LibFuzzer can detect memory errors efficiently, it cannot provide enough information for repair tools to generate patches. For example, even though LibFuzzer inputs `a=3` and triggers both errors, we can not figure out what is the correct conditionals of the patches and where to insert the patches.

[3] https://clang.llvm.org/docs/SanitizerCoverage.html
[4] https://clang.llvm.org/docs/AddressSanitizer.html
[5] https://clang.llvm.org/docs/LeakSanitizer.html

## 2.3 Component-Based Program Synthesis

Component-based program synthesis can synthesize the program that satisfies the test suite using given components, which means, if given an input-output pair$(a, b)$, the synthesized program should output b when input a. In component-based program synthesis, we use a set of basic components (constant, operators, program variables) to synthesize the expected expression. For example, to synthesize an expression using components $(constant, -, +)$ and program variables $(x, y)$, we can synthesize the following expressions.

```
x+y, x-y, y-x, x+c, x-c, c-x, y+c, y-c, c-y
```

We use the components to synthesize the candidate expression and construct the constraint using the test suite. The constraint is in first-order logic and solved by an SMT solver. If it is satisfiable, the expression can be constructed.

$$\theta = \bigwedge_{i,o \in T} \varphi(\psi(e), i, o) \tag{2.1}$$

Formula 2.1 denotes the formal definition of the CBPS constraint. $\psi(e)$ denotes that the synthesized expression $e$ should be a well-formed expression that satisfied the specification of the operator and $i, o$ is an input-output pair of test suite $T$. $\varphi(\psi(e), i, o)$ denotes that when $e$ inputed $i$, the output must be $o$. Finally, the constraint $\theta$ represents that $e$ must be a well-formed expression and given test suite $T$, $e$ should satisfy all input-output pairs in it.

We can use the above components to synthesize an expression that satisfies the test suite shown in Table 2.1. We assign the test suite to the synthesized expressions and use SMT-solver to check whether the logical formula is satisfiable. If it is the case, we output the result; otherwise, we continue to try other expressions. The test suite is satisfied when $c$ of expression `x+c` is 1, hence we can obtain the result expression `x+1`.

Obviously, the synthesized result of CBPS depends on the quality of the test suite. If the test suite does not contain the important test, CBPS may synthesize the overfitting expression, which means that although this expression satisfies the test suite, it is not the result we want. Furthermore, if the given components do not contain the necessary operators and variables, CBPS is unable to synthesize the correct expression.

Table 2.1: Test suite

| x | y | output |
|---|---|---|
| 4 | 1 | 5 |
| 5 | 10 | 6 |
| 6 | 5 | 7 |
| 7 | 12 | 8 |

# 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.

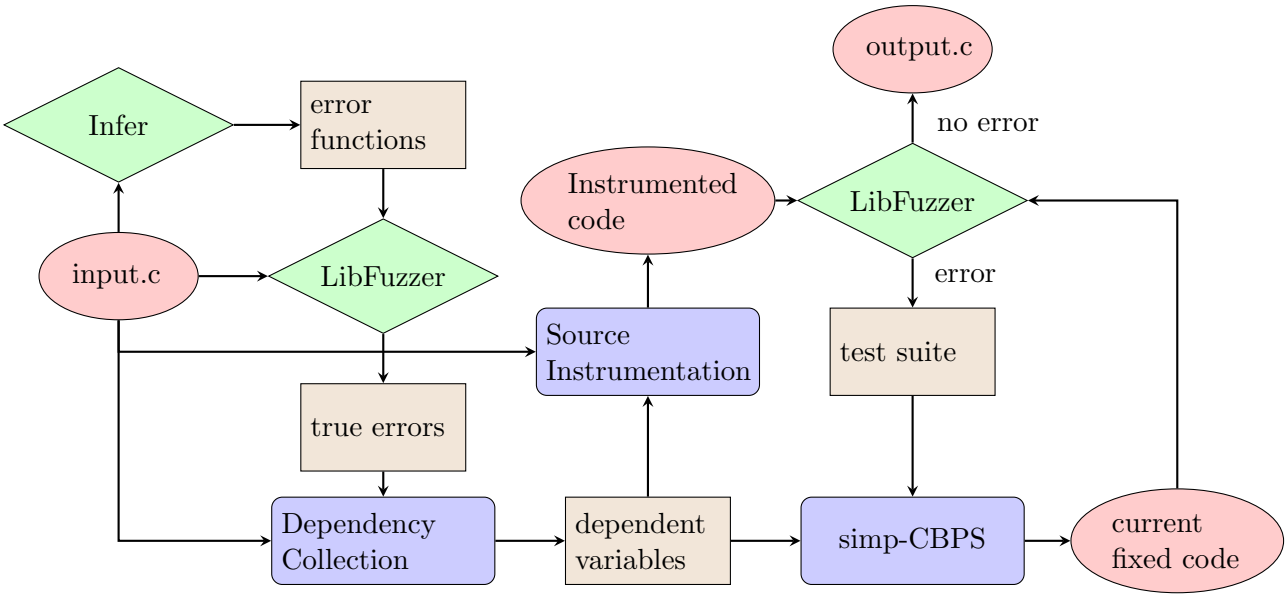


Figure 3.1: HAMER pipeline

```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 [11] 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 most of the 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.

Since HAMER relies on the existing fuzzer to detect errors, its performance is also affected by fuzzer. The purpose of this research is to propose a memory leak repair technique that can be easily integrated with a fuzzer and has high repairability. We use LibFuzzer, which has high scalability and can detect memory leaks, to make HAMER implementation easier. In the future, we will find or develop a fuzzer that is more compatible and suitable for HAMER.

## 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 [11, 16] 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.

**Algorithm 1** Source Instrumentation Result Collection

**Input:** src, Dep
**Output:** SynInf

$dep\_var \leftarrow \emptyset$
$SynInf \leftarrow \emptyset$
$err\_id \leftarrow 0$
**for** $(Var, Ret, ob) \in Dep$ **do**
  $dep\_var \leftarrow dep\_var \cup Var$
  $SynInf[err\_id].add((Var, Ret, ob))$
  $err\_id \leftarrow err\_id + 1$
**end for**
$inst\_code \leftarrow$ **Instrument**$(src, dep\_var)$
$i \leftarrow 0$
**while** $i < 10$ **do**
  $res \leftarrow$ **Fuzz**$(inst\_code)$
  **for** $err\_id = 0$ to $|Dep| - 1$ **do**
    **for** $var \in dep\_var$ **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**
  $i \leftarrow i + 1$
**end while**

### 4.5.1 simp-CBPS

Since we only need to synthesize the conditional of deallocation, we use a simplified version of component-based program synthesis [12] (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.

*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`.

The quality of the test suite is the most critical part of using CBPS to synthesize

Table 4.1: Test suite

| x | output |
|---|---|
| 4 | True |
| 5 | True |
| 6 | False |
| 7 | False |

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`. Obviously, CBPS can only synthesize numeric expressions.

### 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 Example 1, 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 are 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

**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
```

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 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 changed, 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 [6] python module [2] as the SMT solver and implemented six operators (`!=`, `==`, `<`, `<=`, `>`, `>=`) as components.

## 5.2 Experimental Setup

To analyze HAMER, we synthesized 11 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. Since the goal of making synthetic codes is to reproduce complex error patterns, we refer to memory leaks in real-world programs and also assume some more complicated cases. Of course, we can not guarantee that all cases are taken into account, so the goal of this experiment is to

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

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 |
| test11 | 15 | EP, CF, FP |

evaluate the repairability of the repair tool in the six directions of the error patterns shown in Table 5.1, and do not evaluate its scalability.

To fuzz the code, LibFuzzer requires an entry point, which we manually added to the code as shown below.

```
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. We manually checked the patch to see if it fixed the leak without introducing the new error. Table 5.2 shows the final experimental results.

## 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 23 memory leaks in the 11 test codes we synthesized. Infer found 7 memory leaks successfully, missed 16 memory leaks, and reported 2 false alarms. Because SAVER relies on infer's alarms to repair the code, it has no opportunity to fix the 16 unreported errors (false-negative). SAVER generated 4 correct patches and 2 wrong patches for the 7 alarms correctly reported by Infer, and it failed to identify Infer's false alarm and generated 2 wrong patches.

All 23 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

[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 |
| test11 | 1 | 0 | 1/0 | 0.14 | 0 | 1 | 0.05 | 1 | 1 | 0 | 0.02/7.02/7.89 |
| Total/Ave | 23 | 7 | 2/16 | 0.32 | 4 | 4 | 0.05 | 22-23 | 19-23 | 0-1 | 0.03/7.49/9.10 |

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 9.10 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 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 alias 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. HAMER stores error heap objects via temporary variables, so we do not need to worry about the order of deallocation.

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

In test11 (shown in Figure 5.3), *o0* allocated, deallocated, and leaked at 3 different functions. The error heap object was identified by Infer, but the leak location was incorrectly analyzed, and Infer reported the following alarm.

*Object allocated at line 12 is unreachable at line 13.*

Since SAVER is unable to follow heap-related behavior across multiple functions, it generated the erroneous patch, as shown in Figure 5.3b. HAMER quickly detected the error

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

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

int func(int a){
  node *x=new_node(a);//o0
  x->next=new_node(a+1);//o1
  return 0;
}
```

(a) o0, o1 occur memory leak

```c
int func(int a){
  node* tmp_o0;
  node* tmp_o1;
  node *x=new_node(a);
  tmp_o0 = x;
  x->next=new_node(a+1);
  tmp_o1 = x->next;
  free(tmp_o1);
  free(tmp_o0);
  return 0;
}
```

(b) HAMER-generated patch

Figure 5.1: test5

heap object using LibFuzzer, collected the key tests, and successfully generated the conditional deallocation statement patch (Figure 5.3c).

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 7.

Figure 5.4 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 5.4b. 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 5.4c, and executing either path will trigger the memory leaks of *o0, o1, o3*.

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

int func(int a){
  node x;
  x.p1=(int*)malloc(4);//o0
  int* n;
  if(a<5){
    n=x.p1;
  }else{
    n=malloc(4);//o1
  }
  return 1;
}
```

(a) o0, o1 occur memory leak

```c
int func(int a){
  node x;
  x.p1=(int*)malloc(4);
  int* n;
  if(a<5){
    n=x.p1;
  }else{
    n=malloc(4);
  }
  free(x.p1);
  free(n);
  return 1;
}
```

(b) SAVER-generated patch

```c
int func(int a){
  int* tmp_o0;
  int tmp_a = a;
  int* tmp_o1;
  node x;
  x.p1=(int*)malloc(4);
  tmp_o0 = x.p1;
  int* n;
  if(a<5){
    n=x.p1;
  }else{
    n=malloc(4);
    tmp_o1 = n;
  }
  if(tmp_a>=5)free(tmp_o1);
  free(tmp_o0);
  return 1;
}
```

(c) HAMER-generated patch

Figure 5.2: test10

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.

The instance of HAMER generating an error patch is also shown in Figure 5.4d. In beginning, HAMER generated the right patches of *o0* and *o1*. HAMER synthesized conditional `a>=2` with the current test suite when trying to fix *o2*. Then, using LibFuzzer, we can determine that the current patch was erroneous, and for example, we can collect the test of $(a : 4, error : True)$, and synthesize conditional `a>=4`. However, because *o3* has not been repaired and whatever was input memory leak for *o3* will be triggered, LibFuzzer failed to input the critical test `a=5` to trigger *o2*, causing HAMER to incorrectly judge this patch as the correct one and keep it, even LibFuzzer executed 10 times. When fixing *o3*, since *o3* was fixed correctly, LibFuzzer explored all paths and triggered the memory leak of *o2*, causing HAMER to incorrectly judge the patch of *o3* as the wrong patch. Finally,

```
1  int* f(){
2    int *a=(int*)malloc(sizeof(int));
3    return a;
4  }
5
6  void g(int*p){
7    *p=1;
8    free(p);
9  }
10
11 int func(int a){
12   int *p=f();//o0
13   if(a>5){
14     g(p);
15   }
16   return 0;
17 }
```

(a) o0 occurs memory leak

```
11 int func(int a){
12   int *p=f();
13   free(p);
14   if(a>5){
15     g(p);
16   }
17   return 0;
18 }
```

(b) SAVER-generated patch

```
11 int func(int a){
12   int tmp_a = a;
13   int* tmp_o0;
14   int *p=f();
15   tmp_o0 = p;
16   if(a>5){
17     g(p);
18   }
19   if(tmp_a<=5)free(tmp_o0);
20   return 0;
21 }
```

(c) HAMER-generated patch

Figure 5.3: test11

HAMER generated an incorrect patch and deleted a correct patch.

Although we can address a part of the issue shown in test8 (Figure 5.4) by improving HAMER's repair algorithm (discussed in Chapter 7.3), the basic issue is that the fuzzer may fail to detect errors, which gets more common as the type of input becomes more complex. If the error is not detected by the fuzzer, HAMER will be unable to fix it, since the performance of HAMER also depends on the detection tool it uses. Experiments have shown that HAMER can handle these three problems well when the fuzzer detects the error, thus we can say that HAMER can handle these three problems well. In the future, we will find or develop a fuzzer with better performance and more compatibility with HAMER.

| HAMER can handle multiple returns, multiple errors, and collect critical tests when LibFuzzer detects the errors successfully. |
|---|

### 5.3.3 How efficient is HAMER in using lightweight static analysis?

The repair times for HAMER and SAVER are shown in Table 5.2. HAMER is a memory error repair tool based on hybrid analysis, which collects information for repair using lightweight static analysis. SAVER is a static memory error repair tool that uses static analysis from detection to repair. SAVER's repair time is quite short, but it can not guarantee the correctness of the patches it generated, and its repairability is not as good as HAMER's. HAMER's repair time is longer, but it is obvious that most of the time

is spent on fuzzing, and due to our settings, HAMER spends 5 seconds at the end of the repair checking the correctness of the patches using LibFuzzer. The lightweight static analysis utilized by HAMER can collect the essential information to assist HAMER in generating the correct patches in a very short time. The HAMER repair algorithm does not require time to collect a large amount of heap-related behavior through static analysis, and experimental results also show that HAMER has not failed to repair errors due to our lightweight static analysis missing essential information.

| |
|---|
| Our lightweight static analysis can collect enough information in a short time to assist HAMER to generate the correct patches. |

```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) LibFuzzer failed to verify the wrong patch, make HAMER generate wrong patch and failed to generate the patch of o3

Figure 5.4: test8

# Chapter 6

# Related Work

Existing repair techniques can be categorized as general-purpose or special-purpose, depending on whether they are intended to fix all types of errors or only specific types of errors. HAMER is a special-purpose technique that focuses on fixing temporal memory errors such as memory leaks. Other specialized techniques focus on buffer overflows [25], null dereferences [1,7], etc.. We compared the features of several related techniques shown in Table 6.1.

There are many studies focused on fixing memory errors [2,5,8,10,11,22,27–30,32,33]. Among this, the most directly related to our research is MemFix [16] and SAVER [11]. MemFix starts by removing all deallocation statements from the program, then uses static analysis to trace and collect all heap objects and their candidate fix locations, and finally finds a set of patches that can free all dynamically allocated memory without introducing memory errors. However, MemFix cannot generate conditional deallocation statements and has limited scalability and repairability. SAVER is the state-of-the-art memory error repair technique, with high scalability and repairability. SAVER first uses Infer to detect the memory error and then collects the heap-related behavior of the program through static analysis and constructs the object flow graph. SAVER's repair algorithm converts the memory error repair problem into the relabeling problem of the object flow graph and can avoid generating wrong patches by considering the relationship between heap objects. However, due to the limitations of static analysis, SAVER cannot understand the dynamic information of the program, which makes it fail to fix some errors or generate erroneous patches. HAMER uses fuzzer to detect programs, allowing HAMER has more opportunities to fix errors (static analyzer's false-negative) and do not spend time on wrong alarms (static analyzer's false-positive). HAMER also uses fuzzer to check generated patches, preventing HAMER from generating wrong patches. HAMER's extensive repair strategy allows it to handle various kinds of memory errors. DEF_LEAK [33] is a dynamic approach to detecting, eliminating, and fixing memory leaks. DEF_LEAK first instruments the program for collecting memory information and then exposes memory leaks through dynamic symbolic execution. DEF_LEAK can generate the deallocation statement patch to eliminate the leaks at run time. However, DEF_LEAK's repair strategy is very simple (just add deallocation statement at suitable location) and can not fix complicated memory errors, so it does not have high repairability. HAMER uses a conditional deallocation statement to fix the error, so it can fix a leak in a typical path. HAMER can also deallocate a heap object at multiple leak locations.

General-purpose program repair techniques can be classified as semantics-based techniques [9, 14, 19–21, 24, 31] and search-based techniques [13, 15, 17, 18, 26]. Search-based techniques search for the correct patch in a certain search space, so although it has high scalability, the quality of the generated patches is often poor and the overhead is high. Getafix [1] fixes errors by learning from developers' past fixing history. Getafix has high

Table 6.1: Related works

| Target | Technique | Fix location | Test suite | Patch | Verification |
|---|---|---|---|---|---|
| General purpose | Angelix | statistical | necessary | static | manual |
| | CPR | user input | unnecessary | dynamic | manual |
| | ExtractFix | static | crash input | static | manual |
| | Getafix | static | training data | static | manual |
| Memory error | SAVER | static | unnecessary | static | static |
| | DEF_LEAK | dynamic | unnecessary | static | manual |
| | HAMER | hybrid | unnecessary | hybrid | dynamic |

scalability and repairability for fixing errors with simple repair patterns such as null dereference. Although fixing memory errors often only requires inserting a deallocation statement, the fix location is usually too far away from the error source, which Getafix can not handle, and Getafix is a static technique, so it cannot guarantee the correctness of the generated patch. Semantics-based techniques collect path constraints through symbolic execution and then synthesize patches through CBPS. Angelix [20] has high scalability and can fix multi-location bugs, but like other semantics-based techniques, Angelix is prone to generating overfitting patches, therefore its repairability entirely depends on the quality of the test suite. HAMER can collect the tests triggered errors generated by the fuzzer, which are essential to synthesize the correct patches. CPR [24] collects essential tests for patch synthesis by exploring input and patch space through concolic execution. ExtractFix [9] improves the performance of CPBS by collecting the sanitizer's constraint and the tests triggered sanitizer. ExtractFix also uses the crash location as the starting point and performs backward control and data-dependency analysis along with crashing path to collect candidate fix locations, so it has the ability to fix the error kinds which have the different crash and fix locations. The main reason these techniques do not perform well in fixing memory errors is that they cannot identify the correct fix location and it is difficult for them to guarantee the correctness of the patches they generated. HAMER collects essential tests via fuzzer, so it does not need the test suite at the beginning. HAMER collects the candidate fix location via lightweight static analysis and uses a fuzzer to find the correct fix location and verify the generated patches. HAMER is a memory error repair technique that can be fully automated among the processes of detection, repair, and verification.

# Chapter 7

# Conclusion

## 7.1 Conclusion

Fixing memory errors is difficult because new and more dangerous errors are easily introduced. In this research, we presented a new automated memory error repair technique based on hybrid analysis. HAMER dynamically detects memory errors and verifies generated patches via fuzzer and collects essential information for repair via lightweight static analysis. Experiments show that HAMER can fix the complicated memory errors effectively, which the existing static memory error repair technique can not repair. In the future, HAMER is expected to get better scalability and repairability.

## 7.2 Limitations

Since we use CBPS to synthesize the conditional of the patch and CBPS can only synthesize the numeric conditionals, HAMER can not synthesize the conditional that includes other types of elements, such as strings. HAMER is also unable to repair memory errors for parallelism. Finally, HAMER's current repair algorithm cannot handle the loop. If a variable is in a loop, for example, it will yield multiple values for a fuzzer input, making it impossible for HAMER to build an expected test suite. Similarly, if the program does dynamic memory allocation in a loop, HAMER can only fix the memory leaks that occur in each loop one by one, but the correct patch may just require deallocation in the loop.

## 7.3 Future Work

HAMER is an automated memory error repair tool. Although HAMER can only fix memory leaks at the present, we can simply extend our repair algorithm to fix use-after-free and double-free errors. We can fix use-after-free by first removing the free function and then treating it as a memory leak. Similarly, deleting the redundant free function for double free may either fix the error or introduce a memory leak, which we can also fix using the current approach.

We can also improve our repair algorithm to prevent patches from failing to generate patches or generating wrong patches due to fuzzer faults, such as test8. To avoid this issue, HAMER's current strategy is to run fuzzer multiple times to decrease the probability of an error being missed. We have two ways of improving HAMER's performance.

- If a new dynamic memory leak is detected during the repair process, we compare it to the previously detected errors and add it to our repair list if it is a new error.
- If we trigger a previously repaired error when repairing the current error, we add that error back to the repair list.

At this time, our implementation of simp-CBPS can only synthesize the conditional of a few templates, thus we need to expand the types of conditional by adding more conditional template specifications. We also need to increase HAMER's scalability and figure out how to deal with the loop.

# 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] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, page 109–126, New York, NY, USA, 2008. Association for Computing Machinery.

[3] 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.

[4] Cristiano Calcagno, Dino Distefano, Peter W. O ' Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), dec 2011.

[5] James Clause and Alessandro Orso. Leakpoint: pinpointing the causes of memory leaks. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 515–524, 2010.

[6] 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.

[7] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 349–358, 2017.

[8] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 459–470, 2015.

[9] 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.

[10] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

[11] 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.

[12] 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.

[13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 802–811. IEEE Press, 2013.

[14] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. Jfix: Semantics-based repair of java programs via symbolic pathfinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 376–379, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 593–604, New York, NY, USA, 2017. Association for Computing Machinery.

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

[17] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 166–178, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1):298–312, jan 2016.

[19] 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.

[20] 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.

[21] 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.

[22] Derek Rayside and Lucy Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 194–203, New York, NY, USA, 2007. Association for Computing Machinery.

[23] 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.

[24] 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.

[25] Alex Shaw, Dusten Doggett, and Munawar Hafiz. Automatically fixing c buffer overflows using program transformations. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 124–135, 2014.

[26] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 43–54, New York, NY, USA, 2015. Association for Computing Machinery.

[27] Tatsuya Sonobe, Kohei Suenaga, and Atsushi Igarashi. Automatic memory management based on program transformation using ownership. In *APLAS*, 2014.

[28] Kohei Suenaga, Ryota Fukuda, and Atsushi Igarashi. Type-based safe resource deallocation for shared-memory concurrency. *SIGPLAN Not.*, 47(10):1–20, oct 2012.

[29] Kohei Suenaga and Naoki Kobayashi. Fractional ownerships for safe memory deallocation. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 128–143, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[30] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 151–162, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426, 2017.

[32] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. *SIGPLAN Not.*, 46(6):270–282, jun 2011.

[33] Bin Yu, Cong Tian, Nan Zhang, Zhenhua Duan, and Hongwei Du. A dynamic approach to detecting, eliminating and fixing memory leaks. *Journal of Combinatorial Optimization*, pages 1–18, 2021.