Automated Memory Error Repair Based on Hybrid Program Analysis

ZECHANG QIAN
(Supervisor: Katsuhiko Gondow)

Abstract

Automated program repair is a technology which 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 bug-finder is used to detect program memory errors and then repair tools collect the essential information via static analysis. But since static bug-finder 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 automatic 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 bug-finder and extracts the real errors from those alarms. Then it tries to fix those errors by using hybrid analysis.*

I. 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[1, 3] 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. On the other hand, static analysis techniques might have a high time and space overhead, and they aren't always particularly good at dealing with problems such as indirect calls and alias. 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 bug-finder to detect the program first, then use a fuzzer to detect the alarms and extract the real bugs. After that, we collect program variables that can be utilized to synthesize patches by variable dependency analysis. We also gather the test cases generated by the fuzzer which trigger or do not trigger the errors and then use the component-based program synthesis technique to try to generate patches from these variables and test cases. Finally, We utilize a 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 while without introducing new ones.

Contributions. This paper makes the following contributions:

- We present a new technique for repairing memory errors based on hybrid analysis.
- We present HAMER [1], a memory error

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

repair tool that implements the proposed approach.

II. Overview

We illustrate key features of HAMER and how it works. Figure 1 depicts a high-level overview of the HAMER pipeline. First, we check the program with the static analyzer Infer [2], and then we manually collect the functions that contain Infer alerts. Following that, these functions are fuzzed by LibFuzzer [3] 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, and then we collect the tests that do or do not trigger the error by LibFuzzer. Finally, we use component-based program synthesis 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 hasn't 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.

i. Motivating Example 1

This buggy code has three memory leaks, denoted by *o0*, *o1*, and *o2*, as shown in Figure 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+0x52204d)
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 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 1 displays the results of o1's collection. The *error* column indicates whether or not an error occurred at the current value, with 1 indicating that it did and 0 indicating that it did not. Table 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 component-based program synthesis[2], which is:

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

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, by using the results of table1 we also can synthesize patches like a=4. With Libfuzzer we can get that when a is 4, o1 still occurs memory leak. First we will try other fix

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

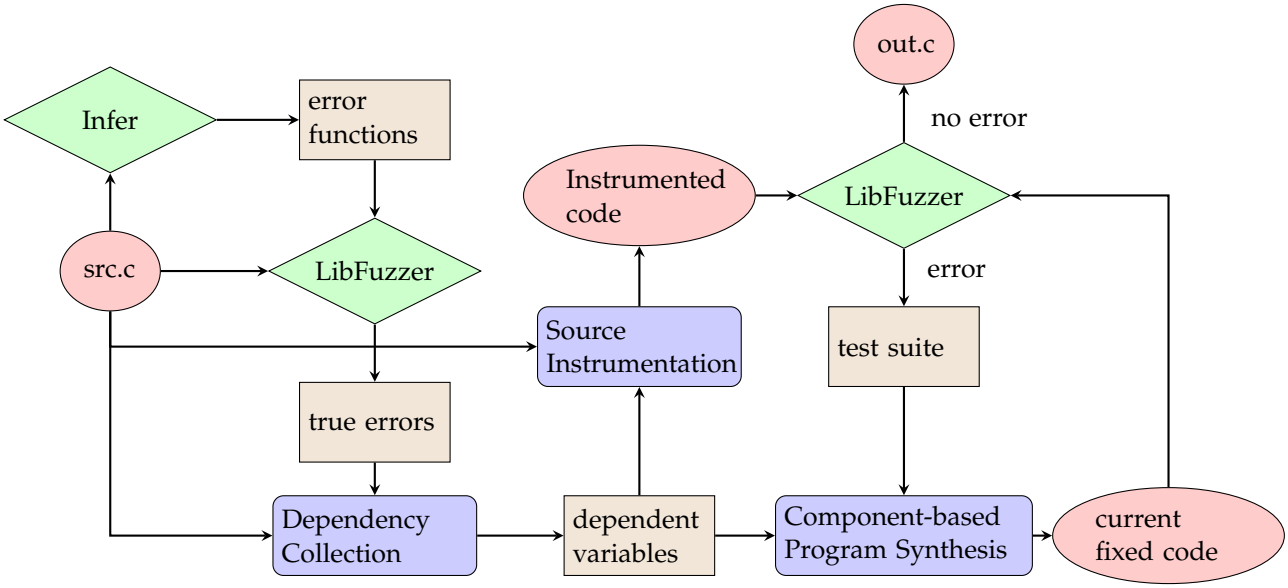

Figure 1: *HAMER pipeline*

locations, but since there is only one return in the func, we can only collect new tests to synthesize a new patch. We append a=4 to test suite and then generate a new patch via the synthesizer. It is obvious that the new test suite let us get the correct patch.

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*.

ii. Motivating Example 2

III. Approach

IV. Evaluation

V. Related Work

VI. Conclusion

References

[1] 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.

[2] 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.

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

```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 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: *Motivating Example 2: Infer false-positive alarm*