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 analyzer is used to detect program memory errors and then repair tools collect the essential information via static analysis. But since 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*

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 [3, 5] 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 has 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 [3] saves these heap-related behaviors by constructing 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 which 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 [4] 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 while 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 collect the relevant information using a lightweight static analysis, then repeatedly synthesize the patch and test it with fuzzer.
- We present HAMER [1], a memory error repair tool that implements the proposed approach.

II. Background

HAMER detects bugs using existing bug detection tools. In this section, we will introduce the static analyzer (Infer [2]) and fuzzer (LibFuzzer [3]) that HAMER uses, as well as the patch generation technique (component-based program synthesis [4] （CBPS）). We will also present the patch template we used to fix the error.

i. Infer

Infer 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).

ii. 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 [4] instrumentation.

LibFuzzer uses the information provided by the AddressSanitizer [5] to determine if an error has happened within the detected code coverage when detecting memory errors. When the

[1]https://github.com/QIANZECHANG/MyResearch(仮)
[2]https://fbinfer.com/
[3]https://llvm.org/docs/LibFuzzer.html
[4]https://clang.llvm.org/docs/SanitizerCoverage.html
[5]https://clang.llvm.org/docs/AddressSanitizer.html

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.

```
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. To improve, we run LibFuzzer several times to detect as many errors as possible.

iii. Patch Template

The purpose of this research is to fix temporal memory errors such as memory leak, 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 section IV.

iv. Component-based Program Synthesis

Since we only need to synthesize the conditional of deallocation, we use a simplified version of component-based program synthesis (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 table1.

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 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 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 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 1 does not have $(x = 6, output = False)$, for example, we might obtain $x < 7$.

III. 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, 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.

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:

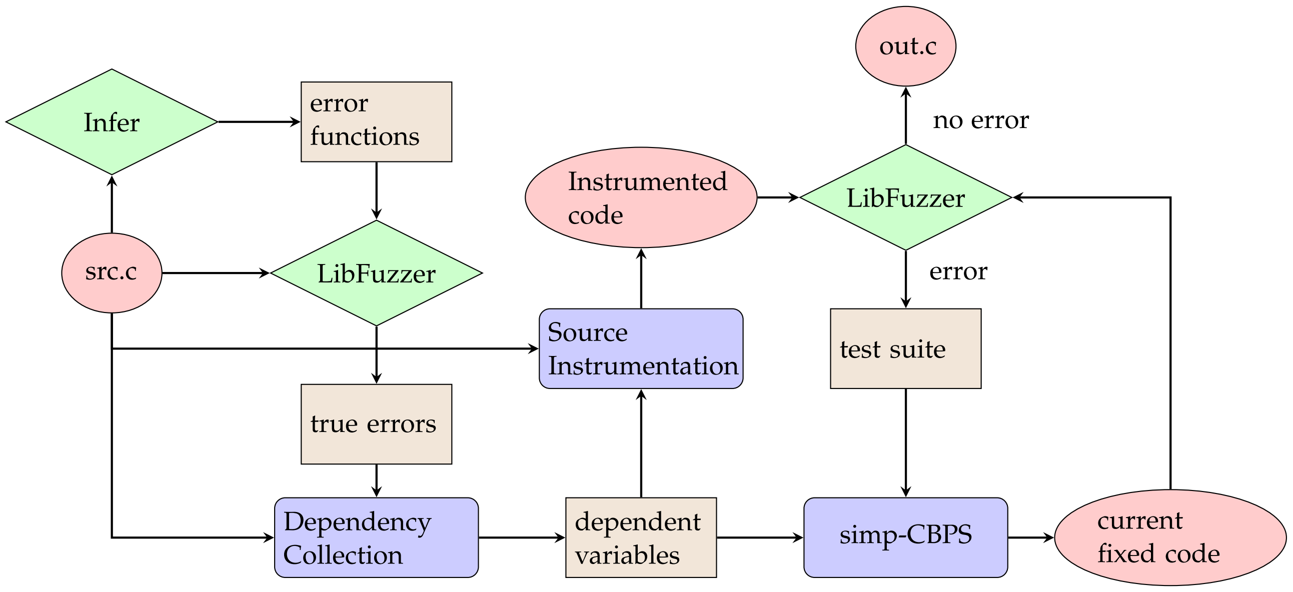

Figure 1: *HAMER pipeline*

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 2: *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 2 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 2 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(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

```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*

tests to synthesize a new patch. For example, we can also synthesize patches like *if(a<=3)free(x);* using the Table 2 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*.

ii. 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 [3] 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.

IV. 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 enough test suites to synthesize the correct patches. 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 subsection iv, we will go over our repair algorithm in detail. Until then, we will describe how HAMER gathers the data required to resolve the errors.

i. 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.

ii. 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.

$$EP = \{get\_path(e) | e \epsilon E\} \tag{1}$$

In Motivating Example 1 (Figure 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 [3, 5] 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 iv. 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.

$$Dep = \{(get\_dep(path), get\_ret(path), get\_ob(path)) | path \epsilon EP\} \tag{2}$$

iii. Source Instrumentation

iv. Patch Generation

Algorithm 1 Source Instrumentation Result Collection

```
Input: src, Dep
Output: SynInf
1:  $dep\_var \leftarrow \emptyset$
2:  $SynInf \leftarrow \emptyset$
3:  $err\_id \leftarrow 0$
4:  for $(Var, Ret, ob) \in Dep$ do
5:      $dep\_var \leftarrow dep\_var \cup Var$
6:      $SynInf[err\_id].add((Ret, ob))$
7:      $err\_id \leftarrow err\_id + 1$
8:  end for
9:  $inst\_code \leftarrow$ Instrument$(src, dep\_var)$
10: $i \leftarrow 0$
11: while $i < 10$ do
12:     $res \leftarrow$ Fuzz$(inst\_code)$
13:     for $err\_id = 0$ to $|Dep| - 1$ do
14:         for $var \in dep\_var$ do
15:             if memory leak happen then
16:                 $SynInf[err\_id][var].add((res[var], 1))$
17:             else
18:                 $SynInf[err\_id][var].add((res[var], 0))$
19:             end if
20:         end for
21:     end for
22:     $i \leftarrow i + 1$
23: end while
```

Algorithm 2 Repair Algorithm

```
Input: src, SynInf
Output: fixed_code
1:  $fixed\_code \leftarrow src$
2:  $queue \leftarrow \{0, 1, ..., |SynInf| - 1\}$
3:  $flag = 0$
4:  while $|queue| \neq 0$ and $flag \neq |queue|$ do
5:    $err\_id \leftarrow queue.pop$
6:    $err\_inf \leftarrow SynInf[err\_id]$
7:    repeat
8:      $cur\_patches \leftarrow$ Synthesizer($err\_inf$)
9:      if $cur\_patches$ same as last time then
10:       break
11:     end if
12:     $cur\_code \leftarrow$ Fix($cur\_patches, fixed\_code$)
13:     if not $fixed$ then
14:       $err\_inf \leftarrow$ Update($err\_inf$)
15:     end if
16:   until $timeout$ or $error\ fixed$
17:   if $current\ error\ fixed$ then
18:     $fixed\_code \leftarrow cur\_code$
19:     $flag \leftarrow 0$
20:     if $queue$ is not empty then
21:       $SynInf \leftarrow$ Clean($SynInf$)
22:       $SynInf \leftarrow$ Update($SynInf$)
23:     end if
24:   else
25:     $queue.add(err\_id)$
26:     $flag \leftarrow flag + 1$
27:   end if
28: end while
```

Algorithm 3 Function Fix

```
Input: cur_patches, code
Output: fixed_code
test_code ← code
for $(cur\_patch, Ret) \in cur\_patches$ do
  for $retloc \in Ret$ do
    $test\_code \leftarrow$ InsertPatch$((cur\_patch, retloc))$
    Fuzz$(test\_code)$
    if new error occurs then
      $test\_code \leftarrow code$
      continue
    else if same error occurs then
      continue
    else
      return $test\_code$
    end if
  end for
end for
return $\emptyset$
```

V. Evaluation

VI. Related Work

CPR [7], Angelix [6], Extractfix [2], Getafix [1], SAVER [3], Memfix [5]

VII. Limitation

VIII. Conclusion

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.