Vulnerability Repair via Concolic Execution and Code Mutations

RIDWAN SHARIFFDEEN, National University of Singapore, Singapore
CHRISTOPHER S. TIMPERLEY, Carnegie Mellon University, USA
YANNIC NOLLER, Ruhr University Bochum, Germany
CLAIRE LE GOUES, Carnegie Mellon University, USA
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Security vulnerabilities detected via techniques like greybox fuzzing are often fixed with a significant time lag. This increases the exposure of the software to vulnerabilities. Automated fixing of vulnerabilities where a tool can generate fix suggestions is thus of value. In this work, we present such a tool, called CrashRepair, to automatically generate fix suggestions using concolic execution, specification inference, and search techniques. Our approach avoids generating fix suggestions merely at the crash location because such fixes often disable the manifestation of the error instead of fixing the error. Instead, based on sanitizer-guided concolic execution, we infer desired constraints at specific program locations and then opportunistically search for code mutations that help respect those constraints. Our technique only requires a single detected vulnerability or exploit as input; it does not require any user-provided properties. Evaluation results on a wide variety of CVEs in the VulnLoc benchmark, show CrashRepair achieves greater efficacy than state-of-the-art vulnerability repair tools like Senx. The repairs suggested come in the form of a ranked set of patches at different locations, and we show that on most occasions, the desired fix is among the top-3 fixes reported by CrashRepair.

ACM Reference Format:
Ridwan Shariffdeen, Christopher S. Timperley, Yannic Noller, Claire Le Goues, and Abhik Roychoudhury. 2024. Vulnerability Repair via Concolic Execution and Code Mutations. 1, 1 (November 2024), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The reliance on open-source software makes our infrastructures prone to the security vulnerabilities of such software. Today, there exist significant challenges in finding and fixing vulnerabilities. First of all, the software typically needs to undergo a campaign of greybox fuzzing to find inputs witnessing the vulnerabilities. Subsequently, even when the vulnerabilities are reported and constructed as CVEs, they may remain unpatched for long [12, 20]. This leads to significant exposure of the software to vulnerabilities. In this work, we take a step towards reducing the lag between *detection* and *repair* of security vulnerabilities. In principle, this could be achieved by merging the fixing process as part of a fuzzing campaign. However, naively attaching an automated fixing process as part of the fuzzing campaign would insert fixes based on a set of tests, which can introduce errors visible in other (unavailable) tests. This corresponds to the well-known problem of producing *overfitting*

Authors' addresses: Ridwan Shariffdeen, National University of Singapore, Singapore, ridwan@comp.nus.edu.sg; Christopher S. Timperley, Carnegie Mellon University, USA, ctimperley@cmu.edu; Yannic Noller, Ruhr University Bochum, Germany, yannic.noller@acm.org; Claire Le Goues, Carnegie Mellon University, USA, clegoues@cs.cmu.edu; Abhik Roychoudhury, National University of Singapore, Singapore, abhik@comp.nus.edu.sg.

XXXX-XXXX/2024/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

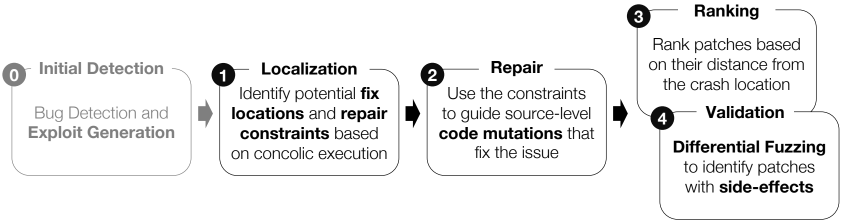

0 Initial Detection
Bug Detection and Exploit Generation
1 Localization
Identify potential fix locations and repair constraints based on concolic execution
2 Repair
Use the constraints to guide source-level code mutations that fix the issue
3 Ranking
Rank patches based on their distance from the crash location
4 Validation
Differential Fuzzing to identify patches with side-effects

Fig. 1. High-level Overview of CrashRepair.

patches in program repair [42]. Thus, any attempt to automatically fix detected vulnerabilities should be able to *generalize* based on the observed vulnerability or exploit.

Automated Program Repair (APR) [24] is an emerging technology that seeks to rectify errors or vulnerabilities automatically. Most automated repair techniques are test-driven, requiring a test suite in the form of input-output examples to rectify errors. The goal is often to generate a (minimal) change in the source code so that the patched code meets the input-output examples. While test-driven program repair is promising, it keeps the error fixing as post-processing to error-finding. Since a large number of errors are reported daily, and CVEs are created regularly, there is always a process of prioritization of "which error to fix". In addition, the lack of workforce for fixing known vulnerabilities leads to a time lag between finding and fixing a vulnerability. Recent works on security vulnerability repair, such as Senx [21], ExtractFix [17], and CPR [39], ease the task of fixing by relaxing the dependence on tests (i.e. requiring a single failing test witnessing the vulnerability). However, Senx depends on user-given properties, and its repair space only considers the first fix location that helps to avoid the identified property violation and adds a corresponding check. CPR [39] also depends on user-provided constraints and does not handle the issue of fault localization. ExtractFix [17] performs an expensive weakest precondition of the violated security constraint and patches the program by adding checks.

This work focuses on *greater security automation*, specifically for security vulnerability repair in programs. We reduce the dependence on tests and require only one exploit, which is a common form of bug reporting for software vulnerabilities. We do not require any user-provided constraint on desired behavior - and instead use concolic execution to generalize the single test, which in turn helps produce repair constraints for the desired patches. Finally, our patch generation is not limited to conditionals, instead patches meeting the repair constraint are flexibly generated by searching over code mutations. Overall our approach provides a balance between the search-based and the semantic approaches of program repair. Our approach has four main steps and takes a buggy program and a single exploit as input (see Figure 1). In the first step, we identify a set of fix locations and the corresponding repair constraints. We replay the failing input with concolic execution, equipped with sanitizers to extract security-relevant property violations. Based on the collected symbolic information and data dependencies, we determine which input parts are relevant for the property violation and subsequently compute potential fix locations and their repair constraints. In the second step, the collected repair ingredients from the semantic analysis are passed to a search-based repair component, which uses source-level code mutations to generate patches to satisfy the repair constraints. One can prioritize code mutations leading to smaller patches, thereby achieving smaller changes to the buggy program. Step 3 ranks the generated patches based on the distance from the crash location. Finally, in step 4, we validate the patches in order of their ranking; in particular, it checks that (a) the patched code meets the repair constraints inferred (and hence repairs the exploit input), and (b) the patched code does not introduce crashes. The validation uses a differential fuzzing campaign to generate inputs that explore the neighborhood of the original exploit.

We evaluate our approach by comparing our implementation CrashRepair with the state-of-art vulnerability repair techniques Senx [21], ExtractFix [17], VulnFix [49] using two recent security-related benchmarks proposed by VulnLoc [41] and ExtractFix [17]. Our evaluation results demonstrate CrashRepair's ability to locate and repair more observed faults than the state of the art. The experimental evaluation results show that CrashRepair outperforms existing state-of-the-art vulnerability repair approaches.

In summary, we make the following technical contributions:

- a novel combination of symbolic analysis and search-based patch generation for security vulnerability repair,
- the efficient usage of concolic execution to extract fix locations and corresponding repair constraints, and the usage of code mutations to efficiently select patches,
- the implementation of our approach CrashRepair for C/C++ programs and its evaluation on real-world security vulnerabilities.

Our approach has a key advantage: Our fix localization supports the generation of patch candidates that not only disable errors or crashes at the crash location but do fix faults that may have been introduced earlier in the execution.

2 MOTIVATIONAL EXAMPLE

To illustrate our approach, we want to start with a vulnerability that was reported for the LibTiff [3] library and assigned the CVE number *CVE-2016-10092*. LibTiff is a library that provides utilities for the Tag Image File Format (TIFF), a widely used format for storing image data. The bug was found in the function `readContigStripsIntoBuffer()` by the greybox fuzzer AFL [1] and reported as a *heap buffer overflow* [2] error. Listing 1 illustrates a simplified variant of the buggy code.

The specific buggy function handles the reading of bytes from an input image into a buffer for further processing. The crashing input generated by AFL causes a heap-based buffer overflow in the `_TIFFmemcpy` function in `tif_unix.c` in multiple versions of LibTIFF including 4.0.7 which allows remote attackers to have unspecified impact via a crafted image. In the scenario, the `bytes_read` gets assigned to a negative number, which later, in line 12, causes a buffer overflow triggered in a different program location when accessing the pointer `bufp`.

```
 1 static int readContigStripsIntoBuffer(TIFF* in, uint8* buf) {
 2   uint8* bufp = buf;
 3   int32  bytes_read = 0;
 4   uint32 stripsize = TIFFStripSize(in);
 5
 6   for(strip = 0; strip < nstrips; strip++) {
 7     bytes_read = TIFFReadEncodedStrip(in, strip, bufp, -1);
 8     rows = bytes_read / scanline_size;
 9     if ((strip < (nstrips - 1)) &&  (bytes_read != (int32)stripsize))
10       TIFFError(...);
11
12 - bufp += bytes_read;
13 + bufp += stripsize;
14
15   }
16   return 1;
17 } /* end readContigStripsIntoBuffer */
```

Listing 1. Snippet of the buggy code in our illustrative example based on LibTiff program (CVE-2016-10092) and the developer commit 9657bbe. Note the code is simplified to include only the relevant context for brevity.

Let us first consider the scenario where we use general test-based program repair techniques to find a repair for our illustrative example. For this purpose, we use F1X [30] and Darjeeling [43] (which is an optimized implementation of GenProg [23]). Figures 2a and 2b depict the patches generated by F1X and Darjeeling, respectively. Since these two APR techniques require a test suite, we created a test suite inclusive of the crashing test case generated by AFL and a passing test case by randomly mutating the crashing test.

```
static int readContigStripsIntoBuffer
(TIFF* in, uint8* buf)
{
  uint8* bufp = buf;
  int32  bytes_read = 0;
  uint32 stripsize = TIFFStripSize(in);

- for(strip = 0; strip < nstrips; strip++) {
+ for(strip = 0; strip == nstrips; strip++) {
```

(a) Patch generated by F1X

```
static int readContigStripsIntoBuffer
(TIFF* in, uint8* buf)
{
  uint8* bufp = buf;
  int32  bytes_read = 0;
  uint32 stripsize = TIFFStripSize(in);

- for(strip = 0; strip < nstrips; strip++) {
+ for(strip = 0; strip > nstrips; strip++) {
```

(b) Patch generated by Darjeeling

Fig. 2. Patches generated using general Automated Program Repair (APR) techniques.

F1X and Darjeeling generate patches that modify the terminating condition of the for-loop statement to avoid the failing test case. However, the fixes in Figures 2a and 2b do not generalize for test cases beyond the given test suite since they only satisfy the two given test cases. Both APR-generated patches fail to fix the buffer overflow vulnerability and instead generate patches that simply pass the failing test case. F1X and Darjeeling are search-based techniques that enumerate the search space of candidate patches in an attempt to satisfy the test suite. The generated patch would likely be more general if sufficient test cases were provided. This example highlights one of the limitations of current test-based APR techniques, generally known as the overfitting problem [42]. In a security context, we are usually left with a small number of test cases, and hence, unable to use such APR techniques effectively. Furthermore, fixing security vulnerabilities cannot tolerate inaccurate patches, which could lead to undesirable effects by believing that the vulnerability has been fixed when, in fact, it has not. Therefore, there is a need for a vulnerability repair tool that can generate fixes that generalize beyond a single failing test case. In the following text, we explain step by step our proposed solution to generate the correct fix addressing the underlying buffer overflow vulnerability.

First, we execute the buggy program with the crashing input generated by AFL, using concolic execution [37]. Concolic execution is a lightweight form of symbolic execution [6], which uses a concrete input to guide the program execution. Using concolic execution, one can obtain the symbolic constraints and other symbolic information while concretely executing one specific program path. For our purposes, we use concolic execution to reproduce the vulnerability to retrieve the failing constraint from a sanitizer (i.e., `AddressSanitizer` and `UndefinedBehaviorSanitizer`). Note that our concolic execution engine is an extension of the symbolic engine KLEE [6].

In our motivational example, the program crashes at function `_TIFFmemcpy` located in source file `libtiff/tif_unix.c` at line 340. The bug is detected by the security properties checked by KLEE during concolic execution. KLEE [6] is equipped with many security properties to detect bugs that can be detected (i.e., if violated) while exploring the input space of the program under test. Users can also extend the detection capabilities by encoding additional properties. In our example, KLEE detects the violation of the following security property:

$$\texttt{((base @var(pointer, d)) <= (@var(pointer, d)))} \tag{1}$$

The above security property captures a memory safety property for the pointer variable `d`, that the memory address accessed by the pointer should be within the bounds of the memory allocation. In this case, the violation is on the lower bound, which is the base address of the memory region. Variable `d` is a pointer used by the crashing function `_TIFFmemcpy` located in the source file `libtiff/tif_unix.c`. The right-hand side of the constraint `( @var(pointer, d))` depicts the current address captured by the pointer `d`. The left-hand side of the constraint is `(base @var(pointer, d))` depicts the base address for the pointer captured by the program variable `d`. The base address is the starting address for the allocated memory region accessed by the pointer `d`.

The above constraint, which captures the violated security property, can be inferred as the specification the repair must satisfy. We name this specification hereafter as the crash-free constraint (`CFC`). Using the collected symbolic information, we can infer that the variable `d`, appearing in the `CFC` at the crash location, is influenced by the variable `bufp` in function `readContigStripsIntoBuffer`. In our concolic execution, we mark all memory allocations as symbolic memory, which allows us to find the relationship between the pointer `d` at the crash location and the pointer `bufp`.

The pointer `bufp` is propagated via the function call `TIFFReadEncodedStrip` at line 7 in Listing 1. Using dependency analysis, we can further detect which statements in the program have a data dependency to the pointer `bufp`. In our example, the base pointer of the allocated buffer is referenced in `bufp`, and the last memory address recorded by the pointer `d` at the crash location is mapped to the expression `bufp + C` (where `C` denotes a constant). From these data-dependent relations, our fix localization identifies 35 program locations in the execution trace. For each program location l in the execution trace, which has a data dependency, we attempt to translate the `CFC` extracted from the crash location to the program location l. Using the symbolic relations between the observed program variables at program location l in the execution trace and the variables appearing in the `CFC` at the crash location, we can translate and localize the `CFC` to program location l. Thus, obtaining a new constraint at program location l, with variables appearing at location l.

For our example, line 12 highlighted in Listing 1 is listed as a potential fix location from our dependency analysis. For the identified fix location, we compute the constraint using symbolic expressions captured during concolic execution of the crashing input. By translating the Constraint 1 to the local variables at line 12 in Listing 1, we obtain:

```
((@var(pointer, bufp)) <= (@var(pointer, bufp)) + (@var(integer, bytes_read)))
```

Simplifying the above constraint, we obtain the Constraint 2 as the repair specification.

```
(0 <= (@var(integer, bytes_read)))                    (2)
```

Using additional information inferred from the Abstract Syntax Tree (AST) of the program and the symbolic expressions, we can further determine that the right-hand side of the computation at line 12 in Listing 1 can be enforced with the Constraint 3.

```
(0 <= (@result(integer)))                    (3)
```

The difference between Constraint 2 and Constraint 3 is that instead of expressing the `CFC` in terms of the program variable, we specify the constraint to the result of an operation. In our example, the arithmetic operation `+=` at line 12 updates the pointer value for `bufp` with an offset `bytes_read`. Our localization translates the `CFC` to the offset value used in the `+=` operation rather than on the program variable `bytes_read` used. Such fine-granular localization helps the repair process to identify which expression to mutate precisely.

To further elaborate on the difference for this example, we list two fix localization information in Table 1. For each location, we show the line, the constraint, and the values for the available variables. For illustration purposes, we show only a subset of the variable values. Next, we apply constraint-guided source-level mutations to fix the observed issue. For this example, we illustrate

Table 1. An extract of the localization results for our motivational example.

Line	Column	Repair Constraint	State Values
12	17	`(0 <=(@var(int,bytes_read)))`	(`bytes_read`, -1), (`rows`, -1), (`stripsize`, 2048)
12	25	`(0 <= (@result(int)))`	(`bytes_read`, -1), (`rows`, -1), (`stripsize`, 2048)

three different operators of CrashRepair. The most common fix would be to insert a conditional check and exit at line 12, e.g.:

```
if (!((0 <= bytes_read)))  exit(1);
```

According to the test case in our example, the expected exit code is zero; therefore, this patch is not plausible and can be removed using differential testing. Another template would be to add a guard statement for the computation at line 12 using the obtained constraint:

```
if ((0 <= (bytes_read)))  bufp += bytes_read;
```

This kind of expression guarding statements or strengthening existing conditions is a common fix for security vulnerabilities. However, it also can lead to skipping essential program features. In our example, this patch candidate can be removed by fuzzing-based differential testing. Finally, the observed heap buffer overflow issue can be avoided by enforcing the Constraint 3. The constraint specifies that the right-hand side of the += expression should be a positive value. The term `@result(integer)` in Constraint 3 can be filled with many possible program variables like {`nstrips`, `stripsize`, `rows`}. This kind of expression mutation is guided by the provided constraint, which ensures that the replacing variable satisfies the repair constraint. In this example, both `bytes_read` and `rows` do not satisfy the constraint, and only `stripsize` is applied.

```
bufp += bytes_read -> bufp += stripsize;
```

Lastly, we rank the patches based on the distance to the crash location. In the example, the correct fix location has a distance value of 177 from the crash location. The distance is computed as the number of unique lines in the trace of the exploit between the crash location and the fix location. CrashRepair can place this location in the top-5 ranking, i.e., within an acceptable range for developers to inspect [35]. Further, CrashRepair's generated patch is also identical to the developer patch. Interestingly, the same patch also mitigates another vulnerability *CVE-2016-10272*, which implies the developer patch is a generalized fix and CrashRepair can correctly identify such a generalized patch that is distant from the observed crash location.

3 APPROACH

Our approach assumes that a *vulnerability* has already been detected and that an error-triggering *input* is available. These assumptions match the output of a successful fuzzing campaign. Starting with these artifacts, we localize potential fix locations with corresponding repair constraints, generate patches at these fix locations, and use differential testing to validate and rank the generated patches. Figure 3 summarizes the workflow. In the following sections, we explain each of these steps in detail.

CrashRepair is built on top of a concolic execution engine extended from KLEE and is equipped with additional features to extract information from the execution trace $\mathcal{T}$. For each executed instruction, the concolic execution engine would record the debug information for the instruction (i.e., mapping to the source location), the concrete values, the corresponding symbolic expression that maintains the relationship between the symbolic variables, and the changes to the symbolic memory. Using the debug information CrashRepair maps the symbolic expressions of each instruction to program-level expressions. Recording the changes to the symbolic memory allows to

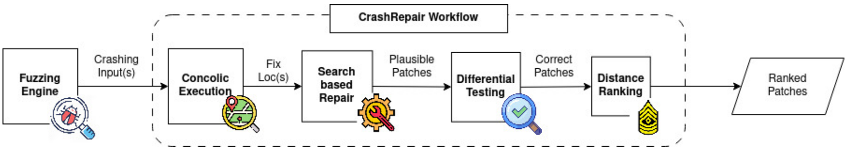


Fig. 3. Workflow of CrashRepair. Takes as input a program under test (PUT) with a crashing test case and generates ranked patches that fixes the underlying root cause of the crash.

query additional information related to memory pointers, which would be beneficial in the later stages of our repair (i.e., constraint generation). To detect the vulnerability exploited via the failing test case t_F, KLEE must be equipped with the necessary sanitizers. Although KLEE does not support `AddressSanitizer`, its built-in memory error detection can identify most memory-related vulnerabilities. For other types of vulnerability detection, such as null pointer dereferences, divide by zero, bad casting, and data type overflows, `UndefinedBehaviorSanitizer` can be equipped with the program. Our concolic version of KLEE generates the necessary semantic information to generate a crash-free constraint `CFC` at the crash location. The necessary semantic information includes the crash instruction, precisely mapped source location, stack trace, memory allocation/deallocations, and pointer aliases.

The high-level localization procedure is shown in Algorithm 1. The procedure ConcolicExec (line 2) takes as input a program $\mathcal{P}$ and a test case t_F and executes the program concretely while capturing symbolic relations. CrashRepair concolically executes the program $\mathcal{P}$ with the given failing test case t_F, i.e., the execution follows the path for t_F but also collects the symbolic information for each instruction. All user inputs to the program and memory locations of the program are marked as symbolic, and the program is executed concretely using the failing test case t_F. While executing each instruction i, KLEE will internally check for security property violations. Upon detecting such a violation, KLEE would terminate the concolic execution and generate the execution trace $\mathcal{T}$.

3.1 Constraint Generation

The procedure CrashAnalysis (line 3 in Algorithm 1) takes as input a program $\mathcal{P}$ and the execution trace $\mathcal{T}$ generated by previous concolic execution. By analyzing the crashing instruction and parsing the Abstract Syntax Tree (AST) of the program $\mathcal{P}$, CrashAnalysis generates a constraint that captures the security property violated in terms of program variables at the program crash location. Therefore, this procedure will generate the crash-free constraint CFC and the set of symbolic variables ($\mathcal{S}$) tainted with the crash-free constraint.

First, the type of crash is determined based on the program stack trace, the executed crashing statement/expression, taint values, and the error message. Once the crash type is determined a template will be instantiated using the corresponding variables appearing in the crashing instruction. Table 2 summarizes the different vulnerability types with their corresponding CFC templates.

```
338 void _TIFFmemcpy(void* d, const void* s, tmsize_t c)
339 {
340         memcpy(d, s, (size_t) c);
341 } /* end _TIFFmemcpy */
```

Listing 2. Crash function in our motivational example based on LibTiff program (CVE-2016-10092).

To illustrate this step, see Listing 2 that depicts the crashing function for our motivational example. The program crashes at program location `libtiff/tif_unix.c:340`, which calls the C library function `memcpy`. The pointer variable `d` passed as a parameter to the function `memcpy`

Table 2. Templates used to generate the Crash Free Constraint (CFC)

Error Type	Example Expressions	Constraint Template
Division by Zero	`a / b, a % b`	b != 0
Arithmetic Overflow	`a + b, a++, a x b`	MIN < a op| b < MAX
Memory Overflow	`*p`	p+sizeof(*p) $\leq base(p) + size(p)$
Shift Overflow	`a « b`	MIN < b < MAX
Type Cast Overflow	`(long) a`	MIN < (long) a < MAX
Error in memcpy	`memcpy(d,s,n)`	d + n $\leq s \vee s + n \leq d$
Error in memmove	`memmove(d,s,n)`	|d - s| < n
Error in memset	`memset(p,s,n)`	p + n < base(p) + size(p)
Assertion Error	`assert(C)`	C
NULL Pointer Dereference	`*p`	p != 0

accesses an out-of-bound memory location, which leads to the program crash. The violated security property is captured by the constraint `((base @var(pointer, d)) <= ( @var(pointer, d)))` . KLEE's symbolic analysis maps variable `d` to a symbolic memory m at the point of the program crash. Hence the set of symbolic variables $\mathcal{S}$ is $\{m\}$.

3.2 Fix Localization

The next step is to identify all program locations in the execution trace $\mathcal{T}$ that have a data dependency to the set of symbolic variables $\mathcal{S}$. CrashRepair analyzes all instructions in trace $\mathcal{T}$ to identify potential fix locations. Procedure GetLine (line 6) translates each instruction i to a source location l. Procedure GetFunction (line 6) queries the AST of the program to identify the corresponding function f. For each source location l in the execution trace $\mathcal{T}$, KLEE records the observed symbolic values, which are extracted by the procedure SymbolicVar (line 6). The symbolic source can be either user inputs to the program, memory locations manipulated by the program, or both.

A program function f is deemed as a potential fix function (lines 6-8) if any of the instructions in the function f uses a symbolic variable in $\mathcal{S}$. This is determined by analyzing each instruction in the execution trace $\mathcal{T}$. Each instruction i is mapped to a source location l, which can be mapped to a function f. A program location l is determined as a potential fix location if the observed symbolic sources $\mathcal{S}'$ at the location l have an intersection with the symbolic sources $\mathcal{S}$ influencing the program crash. If l is a potential location, CrashRepair records the function f in which l belongs, thereby identifying all potential functions where a fix location can be determined. For each identified function f, CrashRepair then finds the earliest location in f, which is also in the execution trace $\mathcal{T}$ (lines 9-14) where all symbolic variables $\mathcal{S}$ are observed. A location l in function f executed in trace $\mathcal{T}$ where all possible symbolic variables are observed is a potential location to enforce the crash-free constraint `CFC`.

In our motivational example, CrashRepair identifies four functions across four different source files as candidate fix functions. This includes the crashing function `_TIFFmemcpy` and the function `readContigStripsIntoBuffer`, where the developer has applied the patch. Furthermore, CrashRepair identifies 35 potential fix locations across these four functions that can be used to generate a repair candidate.

Algorithm 1: Fix Localization

```
Input: program $\mathcal{P}$, failing test case $t_F$
Output: set of potential fix locations $\mathcal{L}$
$\mathcal{L} \leftarrow \emptyset, \mathcal{F} \leftarrow \emptyset$
$\mathcal{T} \leftarrow$ ConcolicExec($\mathcal{P}$, $t_F$) // collect execution trace
CFC, $\mathcal{S} \leftarrow$ CrashAnalysis($\mathcal{T}, \mathcal{P}$) // generate a constraint for the violated security property
for instruction $i \in reverse(\mathcal{T})$ do
    // iterate in reverse order starting from crashing instruction
    $l \leftarrow$ GetLine($i$), $f \leftarrow$ GetFunction($l$) , $\mathcal{S}' \leftarrow$ SymbolicVar($l$)
    if $\mathcal{S} \cap \mathcal{S}' \neq \emptyset$ then
        $\mathcal{F} \leftarrow \mathcal{F} \cup \{f\}$
for function $f \in \mathcal{F}$ do
    $\mathcal{S}' \leftarrow \emptyset$
    for line $l \in$ Sorted($f \cap \mathcal{T}$) do
        $\mathcal{S}' \leftarrow \mathcal{S}' \cup$ SymbolicVar($l$)
        if $\mathcal{S} \subseteq \mathcal{S}'$ then
            $\mathcal{L} \leftarrow \mathcal{L} \cup \{l\}$
return $\mathcal{L}$, CFC
```

3.3 Constraint Translation

The next step is to translate the crash-free constraint CFC from the crash location to the identified fix locations by adapting the constraint to the local context. Using concolic execution to generate the trace $\mathcal{T}$, CrashRepair is able to collect symbolic expressions for each executed instruction. A symbolic expression v represents the relation between the computed value for each instruction and the symbolic variables. Leveraging the expressiveness of the symbolic expressions, CrashRepair can translate the CFC at any location l in the trace $\mathcal{T}$, provided all expressions can be adapted to the local context. Algorithm 2 summarizes the computation of repair constraints where a constraint CFC is translated to the scope of a fix location l in trace $\mathcal{T}$.

For each potential fix location l_{fix}, CrashRepair attempts to translate the crash-free constraint CFC to the local context at l. The procedure ListExpressions (line 3) generates the list of expressions listed in program location l_{fix} by parsing the AST of the program. In order to translate the CFC into the local context, each expression appearing in the CFC needs to be mapped with a semantically equivalent expression at program location l_{fix}.

Using symbolic expressions captured at each location, CrashRepair performs an equivalence check for each expression appearing in the constraint C with program expressions available at location l_{fix}. The procedure IsEquivalent (line 6) identifies equivalent expressions using an SMT solver, e.g., as in our case Z3 [11]. Once all such expressions in CFC can be mapped to a local variable at location l_{fix}, a translated CFC denoted as C can be obtained. The procedure Translate (line 10) takes a set of expressions mapped with the expressions appearing in CFC and replaces the mappings to obtain a new constraint C.

If it fails to translate the CFC, CrashRepair attempts to synthesize an expression in the search space of available expressions at location l_{fix}. For example, program variables a, b at location l_{fix} may not map directly to an expression in CFC. However, the expression a+b, which does not appear at the program location l_{fix}, can be mapped to an expression in CFC. For this purpose, we enumerate possible expressions using a lightweight grammar in the search space of available

Algorithm 2: Constraint Translation

```
   Input: constraint CFC, fix locations $\mathcal{L}$
   Output: set of fix locations with repair constraints and state values $\mathcal{L}_{fix} = \{(l_{fix}, c_{fix})\}$
1  $\mathcal{L}_{fix} \leftarrow \emptyset$
2  for location $l_{fix} \in \mathcal{L}$ do
3      $\mathcal{M} \leftarrow \emptyset, \mathcal{S} \leftarrow \emptyset, \mathcal{E} \leftarrow$ ListExpressions($l$)
4      for expression $e \in$ CFC do
5          for expression $e_0 \in \mathcal{E}$ do
6              if IsEquivalent($e, e_0$) then
7                  $\mathcal{M} \leftarrow \mathcal{M} \cup \{e, e_0\}$
8              if IsTainted($e, e_0$) then
9                  $\mathcal{S} \leftarrow \mathcal{S} \cup \{e, e_0\}$
10     $C \leftarrow$ Translate(CFC, $\mathcal{M}$)
11     if $C = \emptyset$ then
12         $C \leftarrow$ Synthesize(CFC, $\mathcal{S}$)
13     if $C$ then
14         $\mathcal{L}_{fix} \leftarrow \mathcal{L}_{fix} \cup \{l_{fix}, C\}$
15 return $\mathcal{L}_{fix}$
```

expressions at location l_{fix}. For expressions observed at location l_{fix} we enumerate the possibilities of $[e + C, e \times C, e/C, e_1 + e_2, e_1 * e_2]$ where C is a constant and e, e_1, e_2 are expressions. We restrict the search space to expressions that are tainted by the same sources, which also taints CFC. The IsTainted procedure (line 8) determines if two expressions are tainted with the same symbolic source, i.e., if the symbolic expression for both contain the same symbolic sources. The procedure Synthesize (line 12) uses a list of expressions tainted with the same source to enumerate and find a combination of expressions that satisfies the equivalence with an expression in CFC.

If a translated constraint can be obtained at l_{fix}, it will be recorded as a fix location together with the translated constraint C. Note that the fix locations reported by CrashRepair are among the locations executed by the exploit input t_F.

3.4 Patch Generation

Our patch generation algorithm performs AST-level transformations to produce patch candidates at the source level. Algorithm 3 shows the overview of our transformation strategy. For each identified fix location l_{fix}, we explore a set of different program transformations, which are guided and validated by the obtained repair constraint c_{fix} and the collected concrete state v_{fix}. v_{fix} is obtained via the procedure StateValues (line 3), which queries the symbolic state map $\mathcal{V}$ to collect all concrete values observed at a specified program location.

Firstly, we apply transformations to mutate expressions (line 4). We mutate an expression e by replacing variable references and swapping binary operators (e.g., <, >, =, ...). We prune the space of possible expression replacements by checking whether they satisfy the repair constraint c_{fix} using the provided state v_{fix}. In addition, we apply transformations to strengthen existing conditions or to add extra conditions to the program. If the fix location is a *conditional* statement, we propose to enforce the repair constraint at this point (line 9). If the fix location is a non-conditional statement, we propose several options for altering the control flow depending on c_{fix} (line 11). In particular, if l_{fix} is in a function, we can add a conditional *return* before executing the current statement. If l_{fix}

Algorithm 3: Patch Generation

```
Input: program $\mathcal{P}$, symbolic state map $\mathcal{V}$, $\mathcal{L}_{fix} = \{(l_{fix}, c_{fix})\}$
Output: a set of patch candidates $\mathcal{R}$
$\mathcal{R} \leftarrow \emptyset$
for $(l_{fix}, c_{fix}) \in \mathcal{L}_{fix}$ do
    $v_{fix} \leftarrow$ StateValues($l_{fix}$, $\mathcal{V}$)
    if $l_{fix}$ has an expression $e$ then
        $E_1 \leftarrow$ mutate $e$ by replacing variables
        $E_2 \leftarrow$ mutate $e$ by swapping binary operators
        $E \leftarrow$ validate $E_1, E_2$ by checking $c_{fix}$ using $v_{fix}$
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{$replace $e$ in $\mathcal{P}[l_{fix}]$ with $e' \in E\}$
    if $l_{fix}$ is a statement with conditional $c$ then
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{$replace $c$ in $\mathcal{P}[l_{fix}]$ with $c \wedge c_{fix}\}$
    else
        if $l_{fix}$ is in a function then
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{$add conditional return using $c_{fix}$ to $\mathcal{P}$ right before $l_{fix}\}$
        if $l_{fix}$ is in a loop then
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{$add conditional break using $c_{fix}$ to $\mathcal{P}$ right before $l_{fix}\}$
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{$add conditional continue using $c_{fix}$ to $\mathcal{P}$ right before $l_{fix}\}$
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{$add conditional around $\mathcal{P}[l_{fix}]$ using $c_{fix}\}$
    return $\mathcal{R}$
```

Table 3. Repair operators implemented in CrashRepair

Operator	Description	Example
insert-conditional-control-flow	inserts a conditional control-flow statement	`S → S; if(A){exit/break/return}`
strengthen-branch-condition	add new condition to existing condition	`if(A) → if (A && B)`
weaken-branch-condition	remove existing predicate or append new predicate as a disjunction	`if(A) → if(A || B)`, `if(A && B) → if(A)`
guard-statement	adds a guard condition to existing statement	`S → if(C) S`
expression-mutation	replace existing expressions by mutating operators and variables	`a + b → a - b, a → a * b`

is in a loop, we can add a conditional *break* or *continue*. Instead of stopping the overall execution, we can also perform a conditional skip of the current statement by wrapping it within a conditional using c_{fix}. The resulting program transformations are inspired by search-based program repair approaches but enriched with the knowledge from the semantic analysis. Table 6 summarizes the mutations we have implemented in CrashRepair. After generating the patch candidate set, we prioritize patches based on dependency "distance" (how many hops of dependency edges) from the crash, where the strongest preference is given to patches closer to the crash location. Generally, these transformations can be extended/customized for a specific purpose; however, we opted to build CrashRepair on top of mutations that are common in the APR domain [30, 31].

3.5 Patch Validation

Algorithm 4 shows our patch validation strategy, which is based on fuzzing in the neighborhood of the exploit to generate a concentrated test suite [41]. We iterate over the generated inputs until a given timeout is reached. We split the generated inputs into three sets based on how their behavior differs from that of the provided exploit on the original program:

$I_{\equiv}$ inputs that, when executed, produce the same violation as the provided exploit input.
$I_{\times}$ inputs that, when executed, result in a different violation to the provided exploit.
$I_{\checkmark}$ inputs that, when executed, do not result in a violation (i.e., passing inputs).

We report $I_{\times}$ to the developer as evidence of additional vulnerabilities in the program before discarding them for the purpose of patch validation since we have no oracle for their expected behavior. Finally, we discard any inputs in $I_{\checkmark}$ that exhibit non-deterministic behavior w.r.t. exit code, stdout, and stderr. For example, in the case of gnubug-25023 in Coreutils the output includes the timestamp of the run that leads to non-deterministic results for stdout comparisons. By excluding non-deterministic cases, we can use exact matching of the original program's exit code, stdout, and stderr as an oracle. We then use $I_{\equiv}$ and $I_{\checkmark}$ along with the original exploit, i_F, to validate each candidate patch using Execute procedure, which takes as input a program and an input. Intuitively, $I_{\equiv}$ helps to ensure that the patch is general and does not overfit to the specifics of a single failing execution, Moreover, $I_{\checkmark}$ helps to prevent the patch from compromising existing functionality or introducing additional vulnerabilities.

Algorithm 4: PatchValidation

```
Input: program $\mathcal{P}$, set of patch candidates $\mathcal{R}$, the identified property violations $v$, exploit input $i_F$
Output: acceptable repair set $\mathcal{R}'$
$\mathcal{I}_{\equiv} \leftarrow \emptyset, \mathcal{I}_{\times} \leftarrow \emptyset, \mathcal{I}_{\checkmark} \leftarrow \emptyset$
while $i \leftarrow \textit{ConcFuzz}(\mathcal{P}, i_F) \land \neg \textit{timeout}$ do
    $o_{\mathcal{P}} \leftarrow \textsc{Execute}(\mathcal{P}, i)$ // observation original execution
    if no violation in $o_{\mathcal{P}}$ then
        $o_{\mathcal{P}_2} \leftarrow \textsc{Execute}(\mathcal{P}, i)$ // repeat execution
        if $o_{\mathcal{P}} \neq o_{\mathcal{P}_2}$ then
            skip // non-deterministic passing input
        else
            $\mathcal{I}_{\checkmark} \leftarrow \mathcal{I}_{\checkmark} \cup \{i\}$
    else if $o_{\mathcal{P}} = v$ then
        $\mathcal{I}_{\equiv} \leftarrow \mathcal{I}_{\equiv} \cup \{i\}$ // additional proof of same vulnerability
    else
        $\mathcal{I}_{\times} \leftarrow \mathcal{I}_{\times} \cup \{i\}$ // different vulnerability discovered
$\mathcal{R}' \leftarrow \emptyset$
foreach $r \in \mathcal{R}$ do
    foreach $i \in (\mathcal{I}_{\equiv} \cup \{i_F\})$ do
        if $\textit{Execute}(r, i) = v$ then
            skip // exploit not fixed
    foreach $i \in \mathcal{I}_{\checkmark}$ do
        if $\textit{Execute}(r, i) \neq \textit{Execute}(\mathcal{P}, i)$ then
            skip // bug introduced by patch
    $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{r\}$
```

3.6 Ranking

First of all, our ranking prefers patches that have been successfully tested by many inputs during patch validation. Therefore, we increase the score of patches that show no violation, and those changes are exercised by the test execution. Secondly, we detect inputs that show a different

Table 4. Experiment results of CrashRepair on VulnLoc [41] benchmark (BO - Buffer Overflow, DZ - Divide-by-Zero, NPD - Null Pointer Dereference, IO - Integer Overflow, DTO - Data-type Overflow, UAF - Use-after-Free). Patch Distance is the computed average across all 30 trials. Patch Rank shows the best-recorded rank among 30 trials. The Patched? column indicates if a plausible repair was generated in any of the trials.

Subject	**Bug Type**	**Bug ID**	**Fix Localization**		**Constraint Generation**		**Patch Generation**		**Patch Distance**	
			Function	Line	Correct?	Equivalent?	Patched?	Rank	Top-5	Top-10
BinUtils	IO	CVE-2017-14745	N/A	N/A	✗	✗	✗	N/A	N/A	N/A
	BO	CVE-2017-15020	1	1	✗	✗	✗	N/A	N/A	N/A
	DZ	CVE-2017-15025	1	3	✓	✓	✓	1	7.8	12.0
	BO	CVE-2017-6965	N/A	N/A	✓	✓	✓	N/A	19.8	25.0
CoreUtils	BO	gnubug-19784	1	1	✓	✓	✓	1	1.0	1.0
	IO	gnubug-25003	1	2	✓	✗	✓	N/A	3.0	3.0
	BO	gnubug-25023	N/A	N/A	✓	✗	✓	N/A	1.0	8.0
	IO	gnubug-26545	N/A	N/A	✗	✗	✗	N/A	N/A	N/A
Jasper	DZ	CVE-2016-8691	N/A	N/A	✓	✓	✓	1	3.0	3.0
	IO	CVE-2016-9557	1	N/A	✓	✓	✓	1	1.0	5.50
LibArchive	IO	CVE-2016-5844	1	1	✓	✓	✓	1	1.0	1.0
LibJPEG	BO	CVE-2012-2806	1	1	✓	✓	✓	2	1.0	1.0
	NPD	CVE-2017-15232	1	1	✓	✓	✓	1	1.0	1.0
	BO	CVE-2018-14498	1	1	✓	✗	✗	N/A	N/A	N/A
	BO	CVE-2018-19664	N/A	N/A	✓	✗	✗	N/A	N/A	N/A
LibMING	BO	CVE-2016-9264	1	24	✓	✓	✓	1	1.0	1.0
	UAF	CVE-2018-8806	1	1	✓	✓	✓	2	1.0	1.0
	UAF	CVE-2018-8964	1	1	✓	✓	✓	1	1.0	1.0
LibTIFF	DZ	bugzilla-2611	N/A	N/A	✓	✗	✓	N/A	3.0	3.0
	BO	bugzilla-2633	1	N/A	✗	✗	✓	N/A	1.0	1.0
	BO	CVE-2016-10092	3	4	✓	✗	✓	27	175.0	175.0
	BO	CVE-2016-10094	2	N/A	✓	✗	✓	N/A	N/A	N/A
	BO	CVE-2016-10272	2	2	✓	✗	✓	27	175.0	175.0
	BO	CVE-2016-3186	1	3	✓	✓	✓	1	2.0	2.0
	BO	CVE-2016-5314	2	6	✓	✓	✓	N/A	314.40	336.33
	IO	CVE-2016-5321	1	2	✓	✓	✓	2	1.0	1.0
	BO	CVE-2016-9273	N/A	N/A	✗	✗	✗	N/A	N/A	N/A
	BO	CVE-2016-9532	1	1	✓	✓	✓	1	1.0	1.0
	BO	CVE-2017-5225	1	N/A	✓	✗	✓	N/A	1.0	1.2
	DZ	CVE-2017-7595	1	15	✓	✓	✓	8	3.0	3.8
	DTO	CVE-2017-7599	1	1	✓	✓	✓	N/A	1.0	1.40
	DTO	CVE-2017-7600	1	N/A	✓	✓	✓	N/A	1.0	1.0
	IO	CVE-2017-7601	1	N/A	✓	✗	✓	N/A	3.0	3.0
LibXML2	BO	CVE-2012-5134	1	1	✓	✓	✓	1	1.0	41.80
	BO	CVE-2016-1838	1	1	✓	✓	✓	1	5.80	9.90
	BO	CVE-2016-1839	3	13	✓	✓	✓	N/A	4.0	6.30
	NPD	CVE-2017-5969	1	1	✓	✓	✓	1	1.0	57.19
Potrace	BO	CVE-2013-7437	3	6	✓	✓	✓	3	25.0	25.0
ZzipLib	BO	CVE-2017-5974	N/A	N/A	✗	✗	✗	N/A	N/A	N/A
	BO	CVE-2017-5975	1	1	✓	✓	✓	2	1.0	5.64
	BO	CVE-2017-5976	1	1	✗	✗	✓	N/A	1.0	1.10
Total/Average		41	32	26	34	24	33	21	23.12	27.76

property violation in the patched program than in the original program. Such behavior does not necessarily mean a malicious side-effect by the repair but could also reveal a previously masked vulnerability. Nevertheless, we prefer patches that do not show such violations anymore, and hence, we decrease the score in such cases. Finally, we sort patches with the same final score by their dependency "distance" (how many hops of dependency edges) from the crash. We de-prioritize patches very close to the crash (such as patches at the crash location simply disabling the crash).

4 EVALUATION

In our evaluation, we investigate the effectiveness of our approach in fix localization, constraint translation, and patch generation. In particular, we explore the accuracy of the generated fix locations and the contributions of the various mutation strategies of CrashRepair. We also evaluate the patch quality of CrashRepair compared to the state of the art. Therefore, we ask the following research questions:

RQ1 How accurate are the generated fix locations and the corresponding constraints?
RQ2 Can CrashRepair generate correct fixes for security vulnerabilities?
RQ3 How successful are the different repair operators of CrashRepair?
RQ4 How does CrashRepair compare to the state of the art in security vulnerability repair?

With RQ1, we investigate whether our fix locations and constraints generally help to fix the given vulnerability. RQ2 investigates if the generated constraint can be used to successfully obtain a correct patch. In RQ3, we explore the contributions of our repair operators to the generation of correct patches. Finally, in RQ4 we compare our performance against state-of-the-art vulnerability repair techniques.

4.1 Implementation Details

The majority of our CrashRepair is implemented in C++ and Python. The localization and constraint computation is implemented by extending KLEE [6] for concolic execution. The patch generation uses source-level code mutations, and our fuzzer is a modified version of ConcFuzz by VulnLoc [41].

4.2 Experimental Setup

Tools. Multiple APR approaches [17, 21, 23, 25, 27, 31, 34] have been proposed to localize and repair various defect classes. Senx [21], ExtractFix [17], CPR [39], and VulnFix [49] are the most recent works on tackling automated repair of security vulnerabilities in C projects. These approaches have been shown to outperform previous techniques [31, 34] and, therefore, we compare CrashRepair against them. Among the baseline tools only VulnFix has non-deterministic behavior due to the use of fuzzing. Due to the non-deterministic components in both CrashRepair and VulnFix, results for these two tools are reported using 30 repetitions for each tool. All our experiments are executed using the program repair framework Cerberus [38].

Dataset. Our evaluation dataset consists of bugs from the VulnLoc [41] benchmark, which provides a diverse set of 43 vulnerabilities related to buffer overflows, divide-by-zero, integer overflows, null pointer dereferences, heap use-after-free, and data-type overflows. Out of the 43 vulnerabilities in our dataset, two vulnerabilities (i.e., ffmpeg subject) cannot be reproduced in our environment (ubuntu-18.04 and gcc-7.5/clang-10) because they are incompatible with the experimental system or libraries. Therefore, we use the remaining 41 vulnerabilities in our evaluation.

We conducted all our experiments with a timeout of 1 hour, which was reported as a realistic and tolerable timeout for developers [35]. All of our experiments are performed on a 40-core 2.60GHz 64GB RAM Intel Xeon machine, Ubuntu 18.04.

4.3 Fix Localization and Repair Constraints

To answer RQ1, we evaluate the effectiveness of CrashRepair in the following two aspects: 1) finding fix locations and 2) translating CFC to the fix locations. Note that security vulnerabilities are reported at most with a single failing test case, which does not include a developer-provided test suite. Hence, using existing fault localization techniques would be unable to identify the correct fix location due to the unavailability of passing test cases. Our proposed fix localization (ref Algorithm 1) based on concolic execution uses data dependency to identify potential fix locations.

Table 4 shows the overall results of CrashRepair. Column “Fix Localization” depicts the average rank of the developer location identified by CrashRepair in terms of the fixed function and the fixed source line. Sub-column “Function” indicates the average rank of the developer fixed function among the functions identified as potential fix locations by CrashRepair. Similarly, the sub-column "Line" indicates the average rank of the developer fixed line among the lines identified as potential fix locations by CrashRepair. Instances for which CrashRepair cannot identify the developer fixed function or source line is marked as 'N/A'.

CrashRepair can correctly place the developer fix location with respect to the exact line number in the top-1 ranking for 15 instances and in the top-5 ranking for 21 instances. We note that a precise comparison based on line number is not an accurate measure since a patch can be inserted in the near neighborhood; hence, we also report the fix function. CrashRepair can correctly place the developer fixed function in the top-1 ranking for 26 instances and in the top-3 ranking for 32 instances. CrashRepair was unable to determine a fix location for 8 of the bugs due to being unable to determine a location in the trace of the failing test case where all symbolic variables appearing in the constraint `CFC` are observed in a single function.

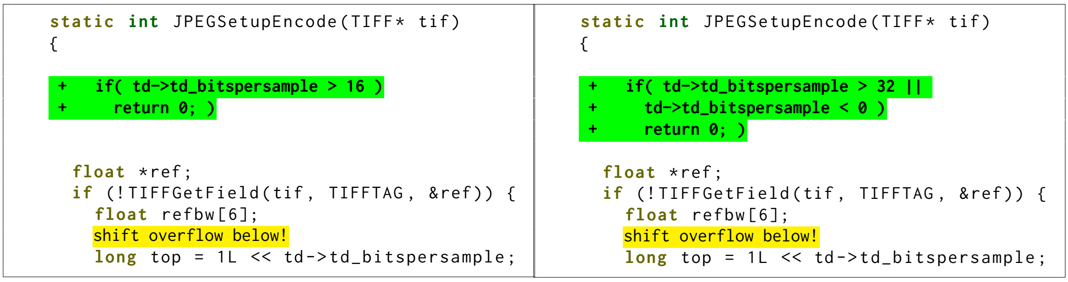
```
static int jpc_siz_getparms(jpc_ms_t *ms, pc_cstate_t *cstate, jas_stream_t *in)
{

+   if (siz->comps[i].hsamp == 0 || siz->comps[i].hsamp > 255) {
+     jas_eprintf("invalid XRsiz value %d \n", siz->comps[i].hsamp);
+     jas_free(siz->comps);
+     return -1;
+   }

}
```

Listing 3. Code Snippet of the developer patch for the vulnerability CVE-2016-8691 in Jasper project.

We manually investigated the bugs for which CrashRepair did not determine the correct fix function in the top-3 ranking. Listing 3 depicts one such example where the developer provided a patch in jpc_siz_getparms function. CVE-2016-8691 is a program crash caused by a division by zero error in Jasper program. The developer patch utilizes the variable siz->comps[i].hsamp, which was not observed at the function jpc_siz_getparms, during the execution of the buggy version of the program. The localization in CrashRepair is restricted to the program variables observed in the execution trace. This prevents our localization algorithm from correctly identifying the developer's fix location. Extending the analysis to all reachable live variables can address this limitation. However, it would decrease the performance of the analysis due to the explosion of all possible variables at each location.

Out of 41 instances, CrashRepair can generate a correct constraint for 34 instances, where 24 of them are semantically equivalent to the developer fix. The results show that our constraint translation can effectively compute a crash-free constraint, especially for integer overflow, divide-by-zero, and developer assertions. Localizing crash-free constraints generated for buffer overflow vulnerabilities remains challenging due to missing fix ingredients at the fix location, such as `buffer_size` and `buffer_base`.

Figure 4 illustrates a scenario where the constraint generated by CrashRepair is different from the developer written patch. CVE-2017-7601 is a shift overflow error observed in the LibTIFF library. Figure 4b shows a simplified version of the patch generated by CrashRepair for the error. The CFC generated for this vulnerability constraints the second operand (i.e., td->td_bitspersample of the shift operator) to be within the minimum and maximum allowed for a valid shift operation. However, the developer patch as shown in Figure 4a has a stronger constraint restricting the variable td->td_bitspersample to be less than 16, which is derived from the file format specification of the input. Although the constraint generated by CrashRepair correctly remediates the vulnerability, it is not equivalent to the developer patch constraint td->td_bitspersample, which is based on additional program semantics.



```
static int JPEGSetupEncode(TIFF* tif)
{

+   if( td->td_bitspersample > 16 )
+     return 0; )


  float *ref;
  if (!TIFFGetField(tif, TIFFTAG, &ref)) {
    float refbw[6];
    shift overflow below!
    long top = 1L << td->td_bitspersample;
```

(a) Simplified patch written by developer

```
static int JPEGSetupEncode(TIFF* tif)
{

+   if( td->td_bitspersample > 32 ||
+     td->td_bitspersample < 0 )
+     return 0; )

  float *ref;
  if (!TIFFGetField(tif, TIFFTAG, &ref)) {
    float refbw[6];
    shift overflow below!
    long top = 1L << td->td_bitspersample;
```

(b) Simplified patch generated by CrashRepair

Fig. 4. Comparison of the repair constraints for the vulnerability CVE-2017-7601 in LibTIFF

RQ1 – Fix Localization and Constraint Generation: CrashRepair can correctly place the developer fixed line in the top-5 ranking for 21 instances and the developer fixed function in the top-3 ranking for 32 instances. For each localized function, CrashRepair is able to generate a crash-free constraint for 34 instances, out of which 24 are semantically equivalent to the constraint employed in the developer fix.

4.4 Fixing Security Vulnerabilities

Using semantic analysis CrashRepair identifies fix locations and generates a repair constraint at each fix location. Once a repair constraint is generated, the program is modified to satisfy the constraint. Guided by the semantic analysis, CrashRepair employs a search-based approach to find a patch that satisfies the repair constraint (ref Algorithm 3). Table 4 shows the overall results of CrashRepair. Column "Patch Generation" depicts the quantitative and qualitative analysis of the patches generated by CrashRepair. The column "Patched?" reports whether a plausible patch that passes the failing test case without additional side effects was generated in any single trial. Sub column "Rank" captures the highest ranking of the developer patch among the list of plausible patches generated by CrashRepair. The column is marked as 'N/A' if the developer patch is not found. Column "Patch Distance" depicts the average patch distance for `Top-5` and `Top-10` patches generated by CrashRepair. The patch distance is computed as the number of unique lines in the trace of the exploit between the crash location and the fix location.

A patch is determined plausible if it a) successfully mitigates the identified security vulnerability, b) matches the expected return code of the program from the developer patch, and c) does not introduce new vulnerabilities with respect to the sanitizer used. We note that security vulnerabilities can be fixed by simply modifying an existing statement. Since the oracle is a single failing test case, it is difficult to identify over-fitting patches. Hence, as a preliminary step, we only generate patches that strictly meet the above criteria.

Figure 5 depicts the distribution of the best rankings for the correct patch, the correct fix location, and the correct fix line. The violin plot in Figure 5 captures the distribution and the frequency for each ranking. The majority of the ranking can be seen within the top-5 for each category. Except for the Correct Fix Function, both the correct patch and correct fix line reach beyond the top-5. More specifically in most instances where a correct patch is generated, it is placed at the top-3. This reflects the ability of CrashRepair to generate the correct patch and to rank it in the highest order. Using the generated repair constraint CrashRepair can generate a plausible patch for 33 subjects, out of which 21 of them are equivalent to the developer fix. In addition, the equivalent

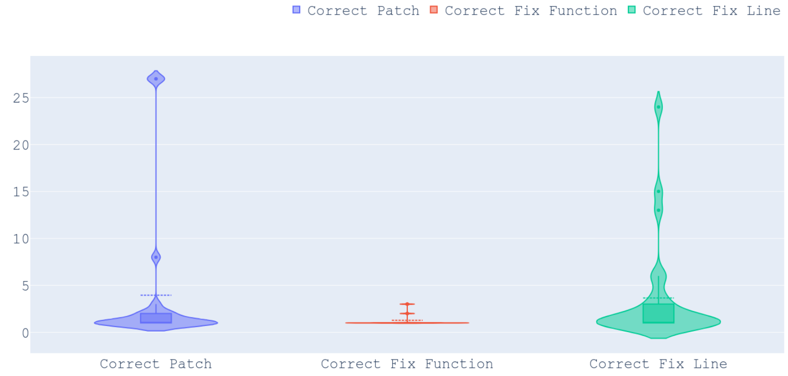


Fig. 5. Distribution of ranking for the correct patch, the correct fix location, and the correct fix line.

developer patch is placed in the top-1 ranking and top-3 ranking for 13 subjects and 18 subjects respectively (see column "Rank") in Table 4.

CrashRepair can generate correct patches earlier in the execution trace with patches having a patch distance greater than 10. The average patch distance for 33 instances where CrashRepair generated a plausible patch is 23.12 and 27.76 for top-5 and top-10 ranked patches respectively. The largest patch distance is observed CVE-2016-5314 with 314.40 lines earlier in the execution trace. The distance is computed in terms of unique source lines in the execution trace. This indicates that CrashRepair can generate patches addressing the root cause of the error, rather than avoiding the crash at the crash location.

Although CrashRepair generates a plausible patch for 33 subjects, for 12 subjects it does not generate a semantically equivalent patch to the developer fix. Manually investigating, we found that the developers used additional domain-specific knowledge to fix the vulnerability. Figure 6 depicts the comparison between a correct patch generated by CrashRepair and the patch written by the developer. Bugzilla-2611 is a divide-by-zero error observed in the LibTIFF library. Figure 6b shows the patch generated by CrashRepair for the error, which checks if the divisor of the modulo division operation is non-zero. However, the developer patch for the division by zero error is to check if a specific field has been set (i.e., sp->decoder_ok). Figure 6a shows a simplified version of the developer patch, which adds an extra field to disable the computation that would trigger the division by zero error. Although the two patches are not equivalent, the patch shown in Figure 6b prevents the division by zero error and successfully fixes the vulnerability.

In addition, we investigate the subjects for which CrashRepair was not able to generate a plausible patch. Due to limitations in the memory violation detection in KLEE, CrashRepair is not

```
static int OJPEGDecode (TIFF* tif, uint8*
    buf, tmsize_t cc, uint16 s)
{

+   if( !sp->decoder_ok )
+     return 0; )
```

(a) Simplified patch written by developer

```
static int OJPEGDecodeRaw(TIFF* tif, uint8
    * buf, tmsize_t cc)
{
+   if (cc%sp->bytes_per_line!=0)
+   if (sp->bytes_per_line!=0
+     cc%sp->bytes_per_line!=0 ) )
```

(b) Simplified patch generated by CrashRepair

Fig. 6. Comparison of the repair constraints for the vulnerability bugzilla-2611 in LibTIFF

Table 5. Statistical tests for CrashRepair over 30 repetitions.

Subject	Bug Type	Bug ID	Input Fuzzing		Patch Generation		
			Failing	Passing	Patched?	Plausible	Rank
BinUtils	IO	CVE-2017-14745	5.47 ± 1.98	0.5 ± 1.53	0	0 ± 0	NA
	BO	CVE-2017-15020	6.73 ± 0.58	0 ± 0	0	0 ± 0	NA
	DZ	CVE-2017-15025	15.43 ± 2.62	0 ± 0	30	15.4 ± 0.67	1 ± 0
	BO	CVE-2017-6965	5.97 ± 1.07	0 ± 0	29	7.73 ± 1.46	NA
CoreUtils	BO	gnubug-19784	197.47 ± 8.78	0 ± 0	30	3 ± 0	1 ± 0
	IO	gnubug-25003	14 ± 0	0 ± 0	30	3 ± 0	NA
	BO	gnubug-25023	23.03 ± 42.47	0 ± 0	30	18 ± 0	NA
	IO	gnubug-26545	0 ± 0	0 ± 0	0	NA	NA
Jasper	DZ	CVE-2016-8691	19.8 ± 12.76	0 ± 0	30	40 ± 0	1 ± 0
	IO	CVE-2016-9557	143.87 ± 34.43	0 ± 0	30	40 ± 0	1 ± 0
LibArchive	IO	CVE-2016-5844	1 ± 0	0 ± 0	30	6 ± 0	1 ± 0
LibJPEG	BO	CVE-2012-2806	5.93 ± 2.24	0 ± 0	30	40 ± 0	2 ± 0
	NPD	CVE-2017-15232	127.57 ± 72.77	0 ± 0	28	5.6 ± 1.52	1 ± 0
	BO	CVE-2018-14498	0.73 ± 1.55	0 ± 0	0	0 ± 0	NA
	BO	CVE-2018-19664	106.03 ± 74.53	0 ± 0	0	0 ± 0	NA
LibMING	BO	CVE-2016-9264	3.2 ± 3.09	2.7 ± 3.1	23	1.53 ± 0.86	1 ± 0
	UAF	CVE-2018-8806	5.13 ± 2.49	0.8 ± 1.67	30	5.47 ± 1.38	2 ± 0
	UAF	CVE-2018-8964	4.3 ± 2.58	1.67 ± 2.51	29	1.93 ± 0.37	1 ± 0
LibTIFF	DZ	bugzilla-2611	19.63 ± 10.9	0 ± 0	30	3 ± 0	NA
	BO	bugzilla-2633	102.23 ± 84.56	0.23 ± 0.77	26	34.67 ± 13.83	NA
	BO	CVE-2016-10092	86.87 ± 91.52	0 ± 0	30	40 ± 0	27 ± 0
	BO	CVE-2016-10094	137.2 ± 69.14	0 ± 0	0	0 ± 0	NA
	BO	CVE-2016-10272	98 ± 85.57	0 ± 0	30	40 ± 0	27 ± 0
	BO	CVE-2016-3186	180.43 ± 27.07	0 ± 0	1	0.2 ± 1.1	1 ± 0
	BO	CVE-2016-5314	9.9 ± 18.43	0 ± 0	30	9 ± 0	NA
	IO	CVE-2016-5321	88.53 ± 95.45	0 ± 0	30	40 ± 0	2 ± 0
	BO	CVE-2016-9273	39.5 ± 59.55	0 ± 0	0	0 ± 0	NA
	BO	CVE-2016-9532	170.93 ± 22.08	0 ± 0	30	3 ± 0	1 ± 0
	BO	CVE-2017-5225	96.93 ± 81.22	0 ± 0	30	18 ± 0	NA
	DZ	CVE-2017-7595	28.97 ± 15.65	0 ± 0	30	40 ± 0	8 ± 0
	DTO	CVE-2017-7599	76.66 ± 49.16	0 ± 0	29	40 ± 0	NA
	DTO	CVE-2017-7600	62.37 ± 41.11	0 ± 0	30	40 ± 0	NA
	IO	CVE-2017-7601	37.33 ± 62.58	0 ± 0	30	12 ± 0	NA
LibXML2	BO	CVE-2012-5134	169.96 ± 36.26	5.57 ± 13.17	28	40 ± 0	1 ± 0
	BO	CVE-2016-1838	81.1 ± 30.57	0 ± 0	30	15 ± 0	1 ± 0
	BO	CVE-2016-1839	0 ± 0	29.43 ± 6.81	30	15 ± 0	NA
	NPD	CVE-2017-5969	105.07 ± 79.94	56.13 ± 72.55	30	22.3 ± 6.9	1 ± 0
Potrace	BO	CVE-2013-7437	4.03 ± 2.33	0 ± 0	30	40 ± 0	3 ± 0
ZzipLib	BO	CVE-2017-5974	55.37 ± 86.37	1.97 ± 3.6	0	0 ± 0	NA
	BO	CVE-2017-5975	36.3 ± 75.33	0.03 ± 0.18	30	38 ± 5.48	2 ± 0
	BO	CVE-2017-5976	13.1 ± 49.86	13.33 ± 36.37	30	40 ± 0	NA

able to generate a correct repair constraint for two subjects: CVE-2017-14745 and gnubug-26545. Improving the capabilities of KLEE could lead to better performance of CrashRepair. In addition, we observe that for some instances CrashRepair removes the correct developer patch in the validation step. Although CrashRepair can correctly generate the developer fix as a candidate patch, the differential testing removes the correct patch due to behavioral changes compared to the original program. Improving the differential test oracle could mitigate such instances, which we leave as future work to investigate.

Accounting for the non-deterministic behavior of CrashRepair due to the differential fuzzing, we repeat our evaluation of CrashRepair for 30 independent trials following the guidelines from

Table 6. Relative effectiveness of CrashRepair's operators.

Operator	Candidates		Subjects
	Total	Plausible	Plausible
insert-conditional-control-flow	83880	15120 (18.03%)	25
strengthen-branch-condition	17352	3086 (14.57%)	12
guard-statement	9588	1290 (13.45%)	10
weaken-branch-condition	8676	959 (11.05%)	9
expression-mutation	8910	960 (10.77%)	2

Arcuri et al. [4]. For each bug in our dataset, we run our tool CrashRepair for 30 trials where each run is provided with a unique random seed. Table 5 summarizes the results of the multiple trials. Column "Input Fuzzing" depicts the distribution of the failing and passing test cases generated for each bug in the format of $x \pm y$, where x is the mean of the distribution and y is the standard deviation. Similarly, sub-columns "Plausible" and "Rank" represent the distribution for the number of plausible patches and the rank of the developer patch for each bug. Sub-column "Patched?" indicates the number of trials that produce at least one plausible patch.

According to the results in Table 5 the output of the concentrated fuzzer varies significantly, as observed with larger standard deviations for both passing and failing test input generation. For 25 vulnerabilities CrashRepair consistently generated a plausible patch, out of which for 16 instances the correct patch was consistently generated and ranked in the same order. In 4 subjects (CVE-2017-15232, CVE-2016-9264, CVE-2018-8806, and CVE-2018-89645), a plausible patch was not generated in some trials while in CVE-2016-3186 only a single trial generated a plausible patch. Investigating further, we identified that differential testing removes the correct patch due to imprecise test oracles and possible flaky behavior. Improving the test oracle and incorporating flakiness suppression mechanisms could yield better results; we leave such improvements as future work.

RQ2 – Patch Generation: CrashRepair is able to generate a patch equivalent to the developer fix for 21 instances. In addition, the developer fix is placed in the top-1 ranking and top-10 ranking for 13 and 19 instances, respectively.

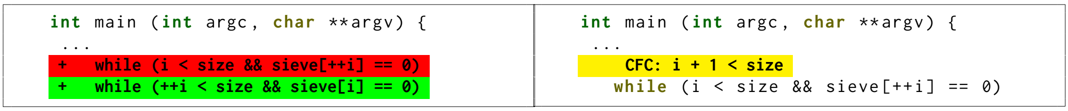
4.5 Repair Operators

Table 6 presents an overview of the relative effectiveness of each of CrashRepair's repair operators. In terms of the number of scenarios that are plausibly fixed by an operator, we see that *insert conditional-control-flow* is the best-performing operator (25 of 41 scenarios). However, no single operator can repair all scenarios.

Looking at the total number of candidate patches produced by each operator, we see that *insert conditional control flow* produces an order of magnitude more candidates than any other operator. The primary reason for the large number of patches lies in the implementation of the *insert conditional return* sub operator: The repair module will generate a candidate patch for each type-compatible variable that is in scope at a given fix location.

Figure 7 depicts an example for which CrashRepair was able to generate the CFC correctly but failed to generate the developer patch. The vulnerability gnubug-19784 in CoreUtils is an out-of-bound memory access caused by an incremented index. Analysis of the CrashRepair identifies the out-of-bound access and generates the constraint (see Figure 7b) for bounds check of the index.

The developer patch is shown in Figure 7a, where the bounds check and the index is swapped. In our current implementation of CrashRepair we do not include a repair operator for swapping expressions. However, the transformations for the patch generation can be extended to support additional repair operators.



```
int main (int argc, char **argv) {
 ...
+   while (i < size && sieve[++i] == 0)
+   while (++i < size && sieve[i] == 0)
```

(a) Simplified patch written by developer

```
int main (int argc, char **argv) {
 ...
    CFC: i + 1 < size
   while (i < size && sieve[++i] == 0)
```

(b) CFC generated by CrashRepair

Fig. 7. GNUBUG-19784 in CoreUtils, for which CrashRepair generated the correct constraint but failed to generate the correct patch.

In terms of the yield of each operator (i.e., the percentage of its patches that are plausible), *insert conditional control flow* is the best performing operator (18.03%) whereas *expression mutation* is the worst performing operator (10.77%).

RQ3 – Repair Operators: Each of CrashRepair's repair operators contributes to its overall performance.

4.6 Comparison with the State of the Art

We compare our results with existing state-of-the-art techniques for vulnerability repair. For each repair approach, it shows the number of plausible patches generated for each of the 41 subjects in the VulnLoc benchmark. Table 7 shows the qualitative and quantitative comparisons with each approach. Column "Plausible Patch" indicates the number of bugs a tool was able to generate a plausible patch. This comparison provides a quantitative measure of the ability of each technique to find a working patch. Column "Correct Patch" indicates the number of bugs the tool finds a developer-equivalent patch for. This comparison provides a qualitative measure of the tools' capabilities to generate a correct patch. For VulnFix and CrashRepair we report the best result out of 30 trials for each bug. The presented results show CrashRepair can produce plausible patches for *33* subjects, while Senx only for *12*, and ExtractFix only for *12*. VulnFix and CPR are able to generate a plausible patch for 17 and 35 instances, respectively. In terms of plausible patches, CPR has the highest count while CrashRepair has the second highest with a significant margin (i.e., 16 additional) over the rest.

Figure 8 shows the distribution of unique bugs each repair tool was able to generate a plausible and correct patch. Figure 8a depicts the breakdown of bugs for which each repair tool was able to generate a plausible patch. Only 1 bug was fixed by all repair tools while CPR and CrashRepair have 5 and 4 uniquely fixed bugs respectively. Similarly, Figure 8b captures the unique bugs each tool was able to correctly fix, correctness is measured in terms of semantic equivalence to the developer patch. CPR and CrashRepair has the most number of correctly fixed unique bugs with 10 and 5 respectively. We note that both CPR and VulnFix assume perfect fix localization, which requires additional input, such as the developer fix location. Such additional information helps to restrict the search space for both finding a fix location and generating a correct patch. In contrast, CrashRepair automatically determines the correct fix location and generates a correct patch using only the single failing test case.

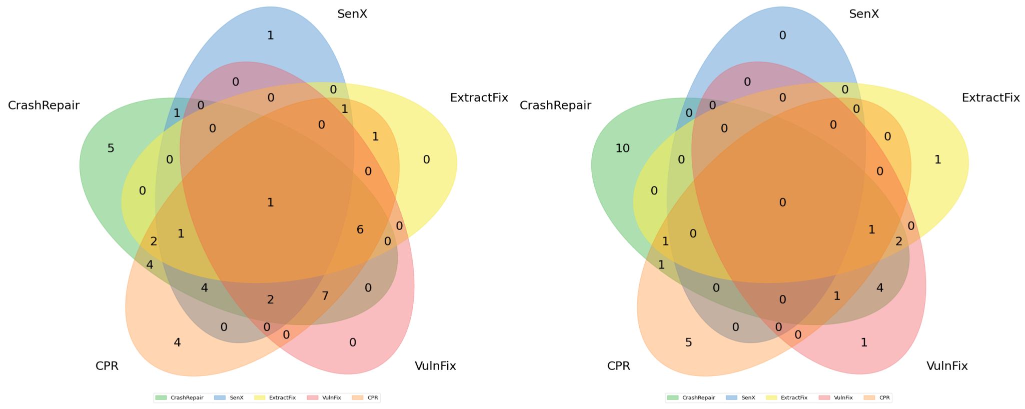
For patch correctness evaluation, we only considered the top-10 ranked patches of each tool. Although having the correct patch in the search space is important, placing the patch in the

Table 7. Comparison with state-of-the-art tools on VulnLoc benchmark.

Subject	**#Vulns**	**Plausible Patch**					**Correct Patch**				
		CrashRepair	Senx	ExtractFix	VulnFix	CPR	CrashRepair	Senx	ExtractFix	VulnFix	CPR
BinUtils	4	2	0	1	2	3	1	0	1	1	0
CoreUtils	4	3	0	2	3	4	1	0	1	0	0
Jasper	2	2	0	0	1	2	2	0	0	0	1
LibArchive	1	1	1	0	0	1	1	0	0	0	0
LibJPEG	4	2	1	1	2	3	2	0	1	2	0
LibMING	3	3	1	0	1	1	3	0	0	1	0
LibTIFF	15	13	8	7	4	14	4	3	2	1	7
LibXML2	4	4	1	1	3	4	3	0	0	3	1
Potrace	1	1	0	0	0	1	1	0	0	0	0
ZzipLib	3	2	0	0	1	2	1	0	0	1	0
Overall	41	33	12	12	17	35	19	3	5	9	9

top rank is similarly important as developers would only examine a few patches [35]. Using the top-10 plausible patches, we determined for how many bugs the tool can generate a developer equivalent patch. In terms of correct patches, CrashRepair has the highest count with 19 subjects generating a correct patch in the top-10 ranking. Senx and ExtractFix generate 3 and 5 correct patches, respectively, while VulnFix and CPR only generate a correct patch for 9 subjects. In terms of qualitative measures, CrashRepair outperforms existing techniques by generating a developer-equivalent patch for 19 instances.

RQ4 – Comparative Performance: CrashRepair outperforms existing state-of-the-art techniques in vulnerability repair by generating high-quality patches in the top-10 ranking for 19 subjects in VulnLoc benchmark.



(a) Plausible Patches

(b) Correct Patches

Fig. 8. (a) Venn diagram of bugs for which repair tools found a plausible patch. (b) Venn diagram of bugs for which repair tools found a correct patch in the top 10.

4.7 Threats to Validity

External Validity. To reduce the risk of an unrepresentative evaluation, we evaluate CrashRepair on the established VulnLoc dataset [41], which holds 43 CVEs from 11 real-world applications. It includes a diverse set of vulnerabilities, including buffer overflows, divide-by-zero, integer overflows, null pointer dereferences, use-after-free, and data-type overflows.

Internal Validity. The main threat to internal validity is the correctness of our implementation because our constraint generation is based on a single trace execution, which holds true only for a single path in the program. Although we use a single failing test case by employing concolic execution, we are able to generalize the constraint to the extent of all possible values along the trace of the failing test case. Additionally, face validity showed that the results are consistent with the expected outcome. Additionally, we have chosen one hour as repair timeout based on recent studies [35], which was reported as a realistic and tolerable timeout. It is possible that other timeouts can lead to other observations. Moreover, all tools have been executed with their default run configurations; e.g., fine-tuning parameters can lead to other results. The concentrated fuzzer [41] used in CrashRepair can potentially generate tests that introduce flaky behavior. The experiment subjects in our evaluation are file processing software, which does not depend on previous test executions. Hence, the impact of flaky tests is minimal. Incorporating flakiness suppression mechanisms ensures that generated tests are non-flaky. However, this is outside of the scope of this work, as the fuzzing integration is only a post-processing step and not the main contribution of this work.

Construct Validity. To determine the correctness of the generated repairs, we manually compared them with the developer fixes. To alleviate this threat, three of the co-authors independently reviewed the patches manually to verify the correctness of the ratings.

5 DISCUSSION

5.1 Vulnerability Detection in KLEE

KLEE [6] symbolic execution engine is capable of executing each instruction in a binary program and symbolically analyze the result of each instruction. KLEE provides in-built support to detect vulnerabilities in the class of memory errors. CrashRepair uses an extended version of KLEE that also detects undefined behavior errors such as Integer Overflow, Shift Overflow etc. Although KLEE can be easily extended to detect additional types of vulnerabilities, it does not support executing binary instrumented with an AddressSanitizer (ASAN)[1] and has limited support for floating point instructions. Hence, some of the errors detected by ASAN will not be detected by KLEE. In our experiments, we could not detect the vulnerability reported in CoreUtils gnubug-26545 using KLEE. The error in gnubug-26545 is due to an overlapping memory region using *memcpy* LibC function. ASAN checks the parameters for the memcpy function, however KLEE does not implement this check. KLEE can be extended to improve its detection capabilities by improving the in-built security properties and supporting out-of-the-box sanitizers. Such extensions are beyond the scope of this work, and will be explored in future work.

5.2 Repair Operators in CrashRepair

The patch generation of CrashRepair constructs a search space of candidate patches using observed program variables, expressions, and C operators at the localized program location. Our implementation closely focuses on prior work to identify suitable repair operators, specifically Angelix [31] and F1X [30]. The complete set of operators is listed in Table 6, which does not cover

[1] https://github.com/klee/klee/issues/1254

all possible transformations that can be applied. For our implementation, we restricted the repair operators that are commonly used and observed in prior work. However, our implementation can be extended to support additional repair operators to generate more specialized program transformations. The vulnerability gnubug-19784 is an example where CrashRepair fails to generate the developer patch despite generating the correct constraint as discussed in Section 4.5. Extending our transformations to include additional repair operators such as *Swapping* can produce the developer patch for gnubug-19784.

6 RELATED WORK

The three stages of handling security vulnerabilities, i.e., *detection* [32, 48], *fix localization* [22, 41, 50], and *repair generation* [17, 21], have been mostly handled separately in the past. CrashRepair combines all of them in one workflow.

6.1 Security Vulnerability Detection

Fuzzing [32] has generated enormous interest in recent decades for the detection of security vulnerabilities. Especially greybox fuzzing is largely applied in industry [5, 13], e.g., to detect program crashes, assertion failures, and memory errors, which all might be exploited by a potential attacker. Static analysis is another popular technique to detect security vulnerabilities in practice [18, 36]. For instance, Facebook's Infer [7] is a static analyzer to detect memory safety issues based on separation logic. CRed [48] performs a pointer-analysis-based static analysis to detected use-after-free (UAF) errors in large programs. Leopard [14] searches for vulnerable functions by ranking them with so-called vulnerability metrics, which are provided as input to their technique.

6.2 Fault Localization for Vulnerabilities

Wong et al. [45] survey the existing fault localization techniques and conclude that Spectrum-based Fault Localization (SBFL) is the most investigated fault localization technique. SBFL techniques rely on the existence of many passing and failing tests, while for security vulnerabilities, we may have only one exploit. Therefore, Shen et al. [41] propose VulnLoc, which essentially uses fuzzing to generate a condensed test suite for the neighborhood of the exploit. Küçük et al. [22] focus on the confounding bias of SBFL, where the correlation may be mistaken as causation and produce fault localization based on statistical causal inference. Following a different direction, Gao et al. [17] propose in their work on ExtractFix the usage of a control/data dependency analysis to identify potential fix locations. With regard to fault/fix localization, CrashRepair is most related to ExtractFix, with which we performed an experimental comparison in Section 4. Overall, CrashRepair does not rely on the generation of test inputs but needs a dependency analysis at the level of LLVM IR.

6.3 Program Repair related to Vulnerabilities

In the context of security vulnerability repair, one can distinguish techniques that are *generally* applicable and techniques that are tailored for specific bug types like *memory errors* or *integer* and *buffer overflows*. The most related works are ExtractFix [17] and Senx [21]. ExtractFix uses sanitizers to detect violations followed by a weakest precondition computation to propagate the repair constraint from the sanitizer to potential fix locations. Repairs are synthesized with the goal of satisfying the repair constraint. Senx requires a human-provided property to identify violations during symbolic execution. It then extracts a predicate based on the available variables in scope to enforce the safety property at a suitable fix location. In another recent work [39], the authors propose concolic program repair (CPR) for the efficient co-exploration of input and patch space to achieve the repair of security vulnerabilities. CPR assumes the fix location as input and requires a user-provided specification to reason about additionally generated inputs. VulnFix [49] is a fuzzing-based technique to infer likely invariants that can act as repair constraints. Provided the fix location, VulnFix performs intensive mutations of the program state at this location. We compared CrashRepair to all these four techniques in our evaluation (see Section 4).

More recently, learning based techniques have been proposed to fix security vulnerabilities [8, 15, 46]. VRepair [8], is an automated vulnerability repair approach that uses Transformer-based Neural Machine Translation (NMT) to fix security vulnerabilities. VulRepair [15] propose an transformer based encoder-decoder approach by fine-tuning a CodeT5 model to repair C vulnerabilities. These techniques also assumes perfect fault localization and the evaluation is based on sequence accuracy as compared to more practical APR setting of verifying the vulnerability remediation by executing the failing test-case. More recent study [46] using learning based repair tools and large language model based tools on Java vulnerabilities shows that, this line of work can only fix very few vulnerabilities. Additionally, these learning based techniques require larger vulnerability repair training datasets, which is more difficult to obtain as vulnerability fixes are scarce.

6.4 Repair of Memory Errors

Multiple approaches have been proposed to repair vulnerabilities explicitly related to memory errors [16, 19, 25, 44, 47]. For example, one can leverage static analyzers like *Infer* to identify memory-related vulnerabilities [19, 44]. Footpatch [44] uses separation logic-based reasoning to generate patches guaranteed to satisfy specific heap properties. However, their work is prone to introduce new errors, such as double-free as a side-effect [19]. SAVER [19] uses a program verification technique and produces patches that include conditional deallocation and relocation of pointer dereferences. Other relevant tools either face scalability problems [25] or have a low repair rate [16]. Xu et al. proposed VFix [47], a value-flow-guided APR approach to repair null pointer dereferences using data and control dependency analysis. Finally, these approaches are fundamentally different from our approach because these approaches lack proactive vulnerability detection and repair. For example, CrashRepair can be combined with a fuzzer to detect and repair vulnerabilities. Besides, most of the generated repairs are crash-avoiding repairs; such repairs make the code hard to maintain later.

6.5 Repair of Buffer and Integer Overflows

Various approaches [9, 10, 26, 28, 29, 33, 40] have been proposed to combat overflow-related vulnerabilities. For example, IntRepair [33] targets integer overflows. However, it uses symbolic execution and SMT solver to reason about the repair and thus suffers the path explosion issues. Cheng et al. [9] proposed IntPTI, an APR approach to support developers in improving code quality against integer errors. They use a static value analysis to achieve proper-type inference for expressions and variables. These inferred types are utilized to generate template-based repairs deduced from common fix patterns. Long et al. [28, 29] use static analysis to generate sound input filters that avoid subsequent integer overflows [10, 40] describe transformation templates that can be applied for fixing buffer and integer overflows in C programs. Such templates can be applied as refactoring rules. Logozzo et al. [26] synthesize non-overflowing integer arithmetic expressions leveraging numerical properties that had been inferred with abstract interpretation. Unlike these approaches, CrashRepair does not rely on static analysis, templates, or abstract interpretation. Instead, we leverage efficient concolic execution guided by a concrete input to identify fix locations for which we extract repair constraints that guide source-code level mutations.

7 CONCLUSION

In this work, we propose CrashRepair, the combination of semantic analysis and search-based patch generation for the repair of security vulnerabilities. The semantic analysis produces a set of fix locations with corresponding repair constraints. The search-based repair operators are steered by the repair constraints and mutate statements at the identified fix locations. Experimental results show the successful combination of semantic analysis with search-based repair in our CrashRepair engine for fixing security vulnerabilities.

Our tool and experiment setup is available via: https://github.com/nus-apr/CrashRepair and an archived replication package is hosted in Zenodo at https://doi.org/10.5281/zenodo.13751398

ACKNOWLEDGMENTS

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant ”Automated Program Repair”, MOE-MOET32021-0001.

This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-19-1-0501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

REFERENCES

[1] 2017. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2023-10-17.

[2] 2023. Blog. https://blogs.gentoo.org/ago/2017/01/01/libtiff-multiple-heap-based-buffer-overflow/.

[3] 2023. LibTIFF Project. http://www.libtiff.org/.

[4] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1985793.1985795

[5] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and Reflections. *IEEE Software* (2020), 0–0. https://doi.org/10.1109/ms.2020.3016773

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[7] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11.

[8] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2023. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering* 49, 1 (2023), 147–165. https://doi.org/10.1109/TSE.2022.3147265

[9] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: Automatic integer error repair with proper-type inference. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 996–1001. https://doi.org/10.1109/ASE.2017.8115718

[10] Zack Coker and Munawar Hafiz. 2013. Program transformations to fix C integers. In *2013 35th International Conference on Software Engineering (ICSE)*. 792–801. https://doi.org/10.1109/ICSE.2013.6606625

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[12] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. *CoRR* abs/2103.11518 (2021). arXiv:2103.11518 https://arxiv.org/abs/2103.11518

[13] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. *arXiv preprint arXiv:2103.11518* (2021).

[14] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 60–71. https://doi.org/10.1109/ICSE.2019.00024

[15] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3540250.3549098

[16] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 459–470. https://doi.org/10.1109/ICSE.2015.64

[17] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 14 (Feb. 2021), 27 pages. https://doi.org/10.1145/3418461

[18] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–23. https://doi.org/10.1109/SCAM.2018.00009

[19] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 271–283. https://doi.org/10.1145/3377811.3380323

[20] Pieter Hooimeijer and Westley Weimer. 2007. Modeling Bug Report Quality. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/1321631.1321639

[21] Z. Huang, D. Lie, G. Tan, and T. Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. 539–554. https://doi.org/10.1109/SP.2019.00071

[22] Yiğit Küçük, Tim A. D. Henderson, and Andy Podgurski. 2021. Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 649–660. https://doi.org/10.1109/ICSE43902.2021.00066

[23] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. https://doi.org/10.1145/3318162

[25] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/3236024.3236079

[26] Francesco Logozzo and Matthieu Martel. 2013. Automatic Repair of Overflowing Expressions with Abstract Interpretation. *Electronic Proceedings in Theoretical Computer Science* 129 (Sep 2013), 341–357. https://doi.org/10.4204/eptcs.129.21

[27] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[28] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound Input Filter Generation for Integer Overflow Errors. *SIGPLAN Not.* 49, 1 (Jan. 2014), 439–452. https://doi.org/10.1145/2578855.2535888

[29] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound Input Filter Generation for Integer Overflow Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 439–452. https://doi.org/10.1145/2535838.2535888

[30] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence Analysis for Automatic Patch Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27 (2018). Issue 4.

[31] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. https://doi.org/10.1145/2884781.2884807

[32] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[33] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. 2019. IntRepair: Informed Repairing of Integer Overflows. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2946148

[34] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[35] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2228–2240. https://doi.org/10.1145/3510003.3510040

[36] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[37] Koushik Sen. 2007. Concolic Testing. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 571–572. https://doi.org/10.1145/1321631.1321746

[38] Ridwan Shariffdeen, Martin Mirchev, Yannic Noller, and Abhik Roychoudhury. 2023. Cerberus: a Program Repair Framework. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 73–77. https://doi.org/10.1109/ICSE-Companion58688.2023.00028

[39] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 390–405. https://doi.org/10.1145/3453483.3454051

[40] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 124–135. https://doi.org/10.1109/DSN.2014.25

[41] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 537–549. https://doi.org/10.1145/3433210.3437528

[42] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[43] Christopher Timperley et al. [n. d.]. Darjeeling: language agnostic search-based repair tool. https://github.com/squaresLab/Darjeeling.

[44] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[45] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[46] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1282–1294. https://doi.org/10.1145/3597926.3598135

[47] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 512–523. https://doi.org/10.1109/ICSE.2019.00063

[48] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-After-Free Vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 327–337. https://doi.org/10.1145/3180155.3180178

[49] Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2022. Program Vulnerability Repair via Inductive Inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 691–702. https://doi.org/10.1145/3533767.3534387

[50] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2021), 332–347. https://doi.org/10.1109/TSE.2019.2892102