Program Vulnerability Repair via Inductive Inference

Yuntong Zhang
yuntong@comp.nus.edu.sg
National University of Singapore, Singapore

Xiang Gao*
xiang_gao@buaa.edu.cn
Beihang University, China

Gregory J. Duck
gregory@comp.nus.edu.sg
National University of Singapore, Singapore

Abhik Roychoudhury
abhik@comp.nus.edu.sg
National University of Singapore, Singapore

ABSTRACT

Program vulnerabilities, even when detected and reported, are not fixed immediately. The time lag between the reporting and fixing of a vulnerability causes open-source software systems to suffer from significant exposure to possible attacks. In this paper, we propose a counter-example guided inductive inference procedure over program states to define likely invariants at possible fix locations. The likely invariants are constructed via mutation over states at the fix location, which turns out to be more effective for inductive property inference, as compared to the usual greybox fuzzing over program inputs. Once such likely invariants, which we call patch invariants, are identified, we can use them to construct patches via simple patch templates. Our work assumes that only one failing input (representing the exploit) is available to start the repair process. Experiments on the VulnLoc data-set of 39 vulnerabilities, which has been curated in previous works on vulnerability repair, show the effectiveness of our repair procedure. As compared to proposed approaches for vulnerability repair such as CPR or SenX which are based on concolic and symbolic execution respectively, we can repair significantly more vulnerabilities. Our results show the potential for program repair via inductive constraint inference, as opposed to generating repair constraints via deductive/symbolic analysis of a *given* test-suite.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; **Software testing and debugging**; • **Security and privacy →** *Software security engineering*.

KEYWORDS

Automated program repair, Snapshot fuzzing, Inductive inference

ACM Reference Format:
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 (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534387

1 INTRODUCTION

In recent years, we have seen the rise of automated program repair (APR) techniques [16] and tools that automatically fix software bugs and vulnerabilities. These techniques fix program bugs by making patched programs satisfy a given correctness criterion. In the most commonly studied problem formulation, the correctness criterion is given as a test-suite. Such APR techniques are called *test-driven automated program repair*. Specifically, when fixing vulnerabilities, given (1) a vulnerable program *Prog*, (2) a set of failing tests T_f that can trigger the vulnerability, and (3) a set of passing tests T_p representing the functionality that should be preserved, test-driven APR fixes *Prog* at a fix location L to pass both T_f and T_p.

A prominent group of APR techniques fix vulnerabilities by (1) inferring a *repair constraint* at the fix location L under which the vulnerability cannot be triggered, and (2) generating a patch to ensure the repair constraint is always satisfied at L [12, 21, 26, 31]. In this work, we examine the possibility of finding probable or likely repair constraints via inductive (as opposed to deductive) inference. These likely repair constraints are called patch invariants.

Formally, suppose S is the program states seen at location L in program executions, S_{benign} represents benign program states of the passing tests, and S_{vul} are the vulnerable states of the failing tests. The inferred patch invariant I holds on observed benign state $s \in S_{benign}$, but does **not** hold on observed vulnerable state $s' \in S_{vul}$. A patch disables vulnerable executions by ensuring I always holds at L. Such patch invariant can be inferred by either static or dynamic program analyses. The static approaches reason about all the feasible program paths soundly by inspecting program code directly. However, doing so usually relies on symbolic program analysis, leading to expensive computations [10, 23, 26]. In contrast, dynamic approaches infer the patch invariant according to a set of program execution traces over a sample of test cases. These approaches limit their attention to the given test cases, and thus can scale to large programs. However, the inferred invariant and the generated patches may work on the given test suite, but cannot be generalized to the other tests. In other words, the inferred patch invariant I only holds on given S_{benign}, but not on other benign states. In program repair literature, this is called the *overfitting* problem.

To alleviate the overfitting problem, one idea is to generate more test cases, so that, we can infer more precise patch invariants and generate higher-quality patches. Grey-box fuzzing, e.g., AFL [32]

[*] corresponding author

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534387

and LibFuzzer [1], is an efficient test generation approach in detecting software bugs/vulnerabilities. These techniques rely on lightweight instrumentation to collect coverage information to guide the test generation. The test generation goal is to maximize code coverage and hence the possibility to detect bugs. Coverage-based greybox fuzzing can be applied to repair vulnerabilities via (1) generating a test suite by fuzzing the program, (2) classifying the test suites into vulnerable and benign inputs depending on whether a test triggers the vulnerability, (3) inferring patch invariant using the augmented vulnerable and benign test suite, and (4) using the invariant to generate a patch. However, we argue this approach is ineffective for the following two reasons. First, to infer a precise patch invariant to discriminate between vulnerable and benign executions, we need the fuzzer to explore the program states at the fix location - generating representative benign and vulnerable states. However, traditional grey-box fuzzing is mainly designed to maximize code coverage, instead of exploring the program states at certain points. Second, to infer a patch invariant at a certain point (the fix location), the fuzzer is required to generate a large number of tests that can reach this location (*reachability problem*). However, solving the reachability problem is considered challenging even for directed grey-box fuzzing tool [5]. According to [5], generating a test to reach a certain point in large programs takes around two hours, which is not efficient enough for our purposes, since we have to generate many such tests.

To address the above challenges, we propose *snapshot fuzzing* to efficiently explore program states with the goal of inferring precise patch invariants. Specifically, instead of mutating the test inputs at the entry point of a program, snapshot fuzzing heuristically mutates the program state (i.e., *snapshot*) at certain program points. We remark that these mutated program states may not be reachable from the beginning of the program, and the patch invariant is inferred using both feasible and infeasible program states. The impact of infeasible states on our patch invariants is not significant, and it is examined in detail in Section 3.2.

The workflow of inferring patch invariant is as follows: with some initial candidate invariant generated from a limited test suite (the given tests plus the tests obtained from traditional fuzzing), snapshot fuzzing attempts to invalidate the current invariant by mutating program states to find counterexamples. Given a candidate invariant, the mutation step invokes an SMT solver to obtain new values for variables that appeared in the invariant. Such mutation finds a counterexample if the program execution result is different from what the candidate invariant suggests - if a program state satisfying the candidate invariant leads to a failure in execution, this state is considered to be a counterexample to the candidate invariant. These new counterexample program states are then used to refine the candidate invariant, which in turn guides the next round of mutation. We realized our idea in a tool called VulnFix for fixing vulnerabilities using Daikon [8] and cvc5 [3] as backend invariant inference engines. Note that, we did not change the inference engine itself, instead, we just focus on producing more valuable tests/states for inferring high-quality invariants. We evaluated VulnFix on a dataset including 39 real-world vulnerabilities. We assume there is only one failing input representing the exploit available to our tool. With Daikon and cvc5 as backend, VulnFix correctly fixes 19 out of 39 vulnerabilities, outperforming state-of-the-art vulnerability repair tools. When comparing with program input fuzzers AFL [32] and ConcFuzz [27], our approach is more efficient in generating counterexamples for refining the inferred invariants.

```
1   sect = find_section_by_type (filedata, SHT_OPTIONS);
2   + if (sect->sh_size < sizeof (* eopt))
3   +     return FALSE;    // developer patch
4   eopt = get_data (NULL, filedata, options_offset, 1,
         sect->sh_size, ("options"));
5   if (eopt) {
6     ...
7     while (offset <= sect->sh_size - sizeof (* eopt)) {
8       Elf_External_Options * eoption;
9       eoption = (Elf_External_Options *) ((char *) eopt
             + offset);
10      option->kind = BYTE_GET (eoption->kind);
11      option->size = BYTE_GET (eoption->size);
12      ...
13      offset += option->size;
14      ++option;
15    }
16  }
```

Figure 1: Simplified code snippet for CVE-2019-9077.

Contributions The contributions of this paper include:

- We propose an approach for fixing vulnerabilities based on counterexample-guided inductive inference. This helps reduce the over-fitting problem in automated program repair, without any significant deductive machinery.
- We implemented our technique in a tool called VulnFix to generate patches in the form of conditions and evaluated it on 39 real-world vulnerabilities. Evaluation results show that our snapshot fuzzing outperformed traditional grey-box fuzzing in generating useful test cases, and VulnFix outperforms state-of-the-art vulnerability repair tools.

2 MOTIVATING EXAMPLE

In this section, we illustrate the workflow of VulnFix for inferring patch invariants to repair a security vulnerability in a real-world application. The vulnerability used in this section is CVE-2019-9077 [1], which is a heap-based buffer overflow vulnerability in the GNU Binutils. Figure 1 shows the code snippet of this bug.

At line 4, the function call get_data allocates a buffer of size sect->sh_size, which is pointed to by eopt. As the two variables used on the right-hand side of the while condition at line 7 are of type unsigned long, if sect->sh_size is less than sizeof(*eopt), the subtraction operation can underflow to a very large number. This causes the while condition to unexpectedly pass, resulting in buffer overflow read at line 11 with the call to BYTE_GET. The developer fixed this bug by adding a check at line 3 to prevent the integer underflow from happening in the while condition, thereby preventing the buffer overflow. In the rest of this section, we describe how VulnFix generates a patch invariant for this example and how it can help fix this bug.

Input-level Fuzzing. Given one exploit input that triggers the bug and the target location $\mathcal{L}_{patch}$ for inferring invariants, input-level fuzzing generates inputs to observe more vulnerable or benign program states at $\mathcal{L}_{patch}$. In this example, we set $\mathcal{L}_{patch}$ as the code at

[1] https://sourceware.org/bugzilla/show_bug.cgi?id=24243

Table 1: Patch invariants and new values generated for Binutils CVE-2019-9077, where `e_shnum` denotes `filedata->file _header.e_shum` in the program.

Phase	#Inv	Examples of New Program States	Benign?	Invariant Example
Exploit-only	-	`{ e_shnum=4, do_segments=1, sect->sh_size=1, symtabno=0, ... }`	vulnerable	-

Input-level Fuzzing	23	`{ e_shnum=4, do_segments=1, sect->sh_size=9, symtabno=0, ... }`
		`{ e_shnum=4, do_segments=1, sect->sh_size=255, symtabno=0, ... }`
		`{ e_shnum=2, do_segments=1, sect->sh_size=1, symtabno=0, ... }`
			benign
			benign
			vulnerable
				`e_shnum < sect->sh_size`

SF Round_1	4	`{ e_shnum=32, do_segments=1, sect->sh_size=32, symtabno=0, ... }`	benign	`do_segments < sect->sh_size`
SF Round_2	3	`{ e_shnum=4, do_segments=-1, sect->sh_size=1, symtabno=0, ... }`	vulnerable	`sect->sh_size-symtabno >= 9`
SF Round_3	1	`{ e_shnum=4, do_segments=1, sect->sh_size=8, symtabno=0, ... }`	benign	`sect->sh_size >= 8`
SF Round_4	1	`{ e_shnum=4, do_segments=1, sect->sh_size=7, symtabno=0, ... }`	vulnerable	`sect->sh_size >= 8`
SF Round_5	1	`{ e_shnum=4, do_segments=1, sect->sh_size=10, symtabno=0, ... }`	benign	`sect->sh_size >= 8`
......				

line 3, which is the same place as the developer patch. Input-level fuzzing generates a few more test inputs that can trigger the same buffer overflow with different program states. For instance, row `Input-level Fuzzing` of Table 1 shows a program state that can trigger this bug with a different value of `e_shnum`. With the augmented program states that demonstrate various scenarios where the bug can be triggered (or not triggered), it is expected that a high-quality invariant can be inferred to classify the vulnerable and benign executions. Based on the observed benign and vulnerable snapshots (given exploit plus the tests generated by AFL), the invariant inference engine infers 23 candidate patch invariants. The last column of Table 1 shows one example invariant. Unfortunately, none of them is correct. Actually, input-level fuzzing only generates limited program states, because they cannot generate enough test inputs that drive program execution to line 3. Even reaching line 3, input-level fuzzing does not generate various program states that are sufficient to infer the correct invariant.

Snapshot fuzzing. To further refine the generated invariant, we use *snapshot fuzzing* to generate counterexamples by directly mutating the program states. A program state, denoted as a *snapshot*, is a mapping from all visible program variables at $\mathcal{L}_{patch}$ to their corresponding values. Compared with input-level fuzzing, the main advantage of snapshot fuzzing is that it could bypass the reachability problem and mutate the program states directly in a controlled way. So that a large number of representative program states could be generated at $\mathcal{L}_{patch}$ efficiently, which can drive the inference engine to infer a high-quality invariant. Specifically, given an existing snapshot, we mutate it with the goal of generating counterexample states that can refine the current patch invariants. For instance, given the current patch invariant (`e_shnum < sect->sh_size`), in the first round (`SF Round_1`), snapshot fuzzing generates a new state `{e_shnum=32, sect->sh_size=32, ...}`. This new state is a counterexample since it violates the above patch invariant but does not trigger the buffer overflow. The refined candidate invariants then guide the next round of snapshot fuzzing. This process continues until a stable solution is reached or time out. In this example, after `SF Round_3`, no candidate invariant is removed (the number of candidate invariants (`#Inv`) is not reduced), and the remaining invariant (`sect->sh_size >= 8`) is no longer changed even with more rounds. Actually, the remaining patch invariant is the correct one which can help to generate the following patch that is semantically equivalent to the developer patch (`sizeof(*eopt)` is a constant which is equal to 8).

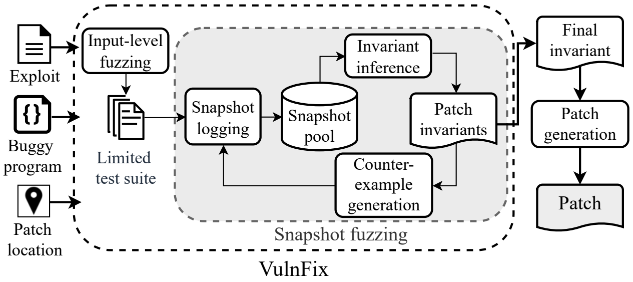

Figure 2: Workflow of VulnFix.

```
+ if (sect->sh_size < 8) return FALSE;
```

Infeasible States. As we mentioned above, snapshot fuzzing could generate infeasible states. In this example, the variable `do_segments` is of type `int`, but the program uses it as an implicit boolean type and only assigns 0 or 1 to it, so all feasible states can only have the value of `do_segments` to be 0 or 1. Since VulnFix does not perform any static analysis on the code, it has no information about this restriction of state feasibility, and it can potentially change the value of `do_segments` to other values, resulting in an infeasible state. Such an infeasible state is shown in `SF Round_2` in Table 1. However, the infeasible states would not affect the correctness of the inferred invariant. Instead, the patch invariants generated based on both feasible and infeasible states are stronger. Meaning that the inferred invariant is not only satisfied when `do_segments` are 0 or 1, but also on other values of `do_segments`.

3 METHODOLOGY

The workflow of VulnFix is shown in Figure 2. VulnFix takes as input a vulnerable program *Prog*, an "exploit" input $\mathcal{I}_{exploit}$ that triggers a known target vulnerability, and a patch location $\mathcal{L}_{patch}$ that indicates where a patch should be applied. We assume that the target vulnerability can be observed via abnormal program termination or crash, including hardware exceptions (`SIGSEGV`, `SIGFPE`,

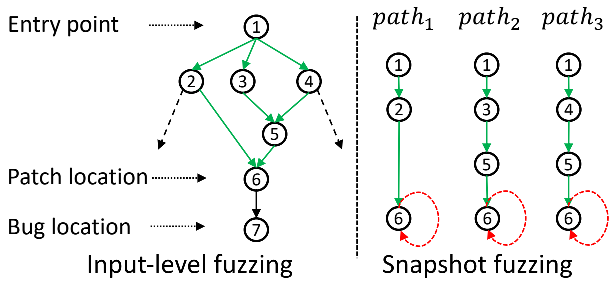

Figure 3: Input-level fuzzing is to explore the paths from the entry point to patch location (the green solid lines), while snapshot fuzzing directly explores the states over the identified paths (the red dot lines).

etc.), assertion failure, or failed sanitizer check (e.g. AddressSanitizer [25]). We also assume a patch location that indicates where the vulnerability should be fixed. In practice, the patch location can be decided using *fix localization* [29] or provided manually.

VulnFix tries to infer a patch invariant at the given patch location $\mathcal{L}_{patch}$ according to the set of observed program states S (donated as snapshots) at $\mathcal{L}_{patch}$. Snapshots S can be partitioned into the set of *benign* program states S_{benign} (that do not trigger the target vulnerability) and the set of *vulnerable* program states S_{vul} (that do trigger the target vulnerability). The output of our workflow is a *patch invariant* that holds for all observed benign program states $s \in S_{benign}$, but does **not** hold for all observed vulnerable program states $s' \in S_{vul}$. The patch invariants capture the underlying conditions that are observed necessary to avoid triggering the vulnerability. Enforcement of the patch invariants can be used to guide program repair. Note that, in this paper, we focus on patch invariant inference, and the way of using patch invariants to generate patches follows existing techniques [10, 12].

VulnFix consists of two main phases *input-level fuzzing* and *snapshot fuzzing*. *Input-level fuzzing* is used to collect an initial set of state observations at the patch location $\mathcal{L}_{patch}$. To avoid overfitting, the second phase *snapshot fuzzing* increases the diversity of states with the aim of generalizing the initial set of patch invariants.

3.1 Input-Level Fuzzing

Our workflow begins with an initial exploit ($\mathcal{I}_{exploit}$) which triggers the vulnerability. In the initial phase, the goal of *input-level fuzzing* is to expand the initial exploit into a *test suite* that exhibits a diversity of both vulnerable and benign program states. The purpose of the initial test-suite is twofold: (1) help to infer an initial set of *patch invariants* based on observed states at the patch location $\mathcal{L}_{patch}$, which acts as a starting point for the alternating loop of invariant refinement and inference, and (2) generate an initial set of *snapshots* that will be mutated in the second phase for invariant refinement.

Specifically, input-level fuzzing plays the role of exploring different paths from the entry point to $\mathcal{L}_{patch}$ as shown in Figure 3 (the green solid lines). As we mentioned in Section 2, snapshot fuzzing directly mutates the program states at $\mathcal{L}_{patch}$, while not changing the execution paths between the entry and $\mathcal{L}_{patch}$. Because of the fact that snapshot fuzzing just mutates a small part of program states (e.g., integer values, boolean values) while keeping most of the states unchanged (e.g., the overall memory layout), it may miss some valid program states. Fortunately, traditional coverage-guided input-level fuzzing can fill this gap by exploring different paths from the entry to $\mathcal{L}_{patch}$. As illustrated in Figure 3, input-level fuzzing explores different paths to the patch location, while snapshot fuzzing further explores the program states along with each path by directly mutating the states.

Initial Test Suite. Our current design builds the first phase on top of standard coverage-based greybox fuzzing tools, namely AFL [32], with a few modifications. Standard fuzzing generates new inputs by mutating the existing inputs, with a higher priority assigned to inputs that increase code coverage. The prioritised inputs are further mutated in the next rounds. This process continues with the goal of increasing code coverage. However, our goal is to find a diversity of inputs that reach the patch location $\mathcal{L}_{patch}$. In addition to code coverage, we thus modify AFL to prioritize inputs that reach $\mathcal{L}_{patch}$. The tests that drive execution to $\mathcal{L}_{patch}$ are saved as the *initial test suite*.

Snapshot Logging. Once the initial test suite $\mathcal{T}$ is generated, the next step is to generate a set of *program states* (a.k.a., *snapshots*) s at the patch location $\mathcal{L}_{patch}$ for each test $t \in \mathcal{T}$. From each snapshot s, we log information useful for invariant inference, including:

(1) a name-value mapping of *live variables* at the patch location ($\mathcal{L}_{patch}$), including global variables, function parameters, and local variables within the current scope;
(2) a name-value mapping of pre-defined *ghost variables* that contain useful values not explicitly represented by the set of live variables; and
(3) a *classification* of whether the snapshot s triggers the vulnerability ($s \in S_{vul}$) or not ($s \in S_{benign}$).

The name-value pairs including basic type variables (e.g., `int`, `bool`, `char`, etc.), pointer variables (e.g., `ptr`), pointer dereference (e.g., `*ptr`), and `struct/class/union` member variables. Since `structs`, `unions`, and pointers can be nested (e.g., `x->y.z`, etc.), the snapshot logger recursively retrieves the nested member variables up to a configurable depth. Pointer values also have a special representation as discussed below.

In addition to the live variables, we also log *implicit* (a.k.a. *ghost*) variables that may contain useful information at the patch location $\mathcal{L}_{patch}$. Such ghost variables may be necessary for inferring a useful invariant that separates the benign and vulnerable cases. For instance, the size of arrays or buffers is usually important when classifying out-of-bound accesses, however, the size of arrays may not be saved in a live variable. Currently, the snapshot logger supports the following ghost variables:

- The size of a global, stack, or heap-allocated buffer. If a buffer is pointed by a visible pointer variable `ptr`, this ghost variable is denoted by `size(ptr)`.
- The base address of the buffer pointed to by a visible pointer variable `ptr`. This ghost variable is denoted by `base(ptr)`. In this case, `ptr` can point to any address within a buffer, and `base(ptr)` is the base address of the buffer.

To obtain the values for `size(ptr)` and `base(ptr)` from the value of a pointer `ptr`, we retrieve the meta-information associated with the corresponding memory defined by sanitizers at runtime. In the current design, we utilise the allocation meta-data from AddressSanitizer [25] to derive the values of the ghost variables.

Our snapshot logger also represents pointer values (e.g., `ptr`) in terms of ghost variables. Specifically, a `ptr` is represented as the *offset* between `ptr` and `base(ptr)` values, which means `ptr` is transformed into `offset(ptr)=ptr−base(ptr)` in the snapshots. For example, if `ptr=base(ptr)+8`, then `ptr` is represented by the offset +8, regardless of the actual value of `ptr` interpreted as an integer. This is because, for most programs, the vulnerability depends on the pointer offset rather than the absolute pointer value at runtime.

The final logged information is the *classification.* For a single input test t, it is possible that multiple snapshots will be recorded, since the execution of t may reach $\mathcal{L}_{patch}$ more than one time (e.g., loops, repeated function calls, etc.). For a given t with snapshots $[s_1, s_2, ...s_k]$, we classify them as follows:

(1) If executing t does not trigger the vulnerability, then all snapshots $s_1, s_2, ..., s_k$ are classified as *benign.*
(2) If executing t does trigger the vulnerability, we classify $s_1, s_2, ..., s_{k-1}$ as *benign*, and s_k is *vulnerable.*

The rationale for (2) is that, if t triggers the vulnerability, s_k is the last state being observed before the program terminates abnormally. The vulnerability could, in principle, be fixed in the last state. Finally, there is a third case where t triggers an *unrelated* vulnerability. Currently, our tool is designed to fix one vulnerability at a time, so this case is discarded.

Patch invariant Inference. Given sets of benign and vulnerable snapshots in the snapshot pool, the next step is to infer a set of invariants. Here, an invariant is a formula over program variables appearing in the snapshots, which evaluates to true for variable values in benign snapshots and false for variable values in vulnerable snapshots. Essentially, this step attempts to infer a formula that separates two sets of concrete values. For this, we use dynamic likely invariant inference [8]. Dynamic invariant inference systems, such as Daikon [8], infer likely program invariants that hold at certain program points for all observed program executions. It works by instantiating invariants according to a list of pre-defined templates. It then uses variable values derived from the observed executions to test the validity of the instantiated invariants. This process eliminates potential invariants that are violated by one or more of the observed executions. The remaining (not eliminated) invariants are deemed "likely invariants" [8], i.e., properties that are invariant over all observed states. With sufficient samples, the remaining invariants can be reasonably accurate, but may "overfit" the invariants to the observations. We slightly generalize the idea of dynamic invariant inference by considering two sets of observations (i.e., S_{benign} and S_{vul}) instead of one. A template is filtered if it either violates a state from S_{benign}, or *passes* a state from S_{vul}. The remaining instantiated templates are deemed *likely patch invariants.*

Besides dynamic invariant inference systems, it is also possible to use program synthesizers such as cvc5 [3] to synthesize a function that classifies the benign and vulnerable values (more details in Section 4).

Algorithm 1: Basic snapshot fuzzing loop

```
  Input: initial snapshot corpus $S$, candidate invariants $\Phi$
  Output: Refined invariants $\Phi$
1 while !Timeout() do
2     $s \longleftarrow$ Select($S$)
3     $s' \longleftarrow$ Mutate($s$)
4     $r \longleftarrow$ Execute($P, s'$)
5     if isCounterExample($r, \Phi$) then
6         $S \longleftarrow S \cup \{s'\}$
7         $\Phi \longleftarrow$ GenerateInv($S$)
8 end
```

3.2 Snapshot Fuzzing

From the given exploit $\mathcal{I}_{exploit}$ and inputs produced by input-level fuzzing, if there are observed benign and vulnerable states at $\mathcal{L}_{patch}$, patch invariant inference produces an initial set of candidate *patch invariants* that distinguishes between the observed benign and vulnerable states. However, since the initial invariants were derived only from observed states, these invariants may be *overfitting* and not generalize to other possible benign/vulnerable states that could arise during the execution of the program. One idea would be to run input-level fuzzing for longer, generating yet more states that can be used to generate more accurate invariants. However, this tends to suffer from the problem of *diminishing returns*, a well-known problem with fuzz testing [4], meaning that ever-larger resources are required for ever-smaller progress. To circumvent this problem, we propose *snapshot fuzzing* to directly mutate states in order to refine the invariants. Unlike input-level fuzzing, snapshot fuzzing can generate large numbers of states quickly, and bypass the *reachability* problem, meaning that it is not necessary to find inputs corresponding to each mutant state.

Basic Algorithm. The basic snapshot fuzzing algorithm is initialized with: (1) an initial set of *patch invariants* Φ, and (2) an initial *snapshot corpus S* consisting of an initial collection of benign/vulnerable states (S_{benign}/S_{vul}). Both (1) and (2) are generated by (and fed from) *input-level fuzzing* in the first phase. The goal of the snapshot fuzzing algorithm is to derive *counterexamples* to the current patch invariants set Φ. Here, a *counterexample* is a state s' satisfying the following property for any $\phi \in \Phi$:

$$\phi(s') \text{ but } s' \text{ is } \textit{vulnerable} \qquad \textbf{OR} \qquad \neg\phi(s') \text{ but } s' \text{ is } \textit{benign}$$

Where $\phi(s')$ means that ϕ is satisfied on state s'. The counterexample s' is a witness demonstrating that the current patch invariants Φ are inaccurate and need refinement. The snapshot fuzzing algorithm is shown in Algorithm 1. The loop (line 1) repeats the following steps: (1) selects an element $s \in S$ from the current snapshot corpus (line 2), (2) mutates s into a new snapshot s' (line 3), and (3) tests the s' against both the current patch invariants Φ and the program P (lines 4,5). Here, `Execute`(P, s') resumes the execution of the program from state s' and observes the result, analogous to running a program from a core dump using a debugger (e.g., gdb). The result $r \in \{\textit{benign}, \textit{vulnerable}\}$ indicates whether the target bug (at location $\mathcal{L}_{vulnerability}$) was observed or not. If a counterexample s' is discovered, then s' is added to the snapshot corpus S, which is

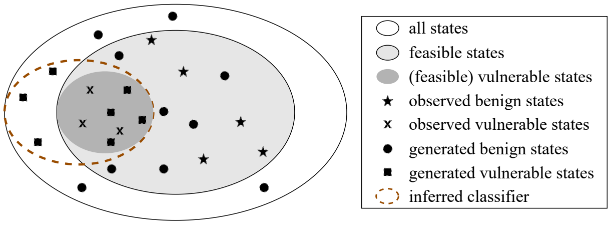

Figure 4: Relationship among different program states.

then used to derive a new (and refined) set of patch invariants. The process continues until a timeout is reached.

Infeasible States. We note that the Mutate operator (line 3) is not guaranteed to preserve *feasibility*, i.e., the mutant state s' need not be reachable from any program input. To clarify the effects of the generated infeasible states on the inferred patch invariant, we show the relationship among various types of states in Figure 4. Although the inferred classifier (i.e. patch invariant) is based on both feasible and infeasible states, it still correctly separates all the observed benign states and observed vulnerable states, which means the patch invariant will still be correct with respect to all the observed states. However, it is indeed possible that the inferred patch invariant is unsatisfied not only by the vulnerable states (both observed and generated), but also by some *unobserved* feasible benign states at the patch location. These unobserved feasible benign states are illustrated in Figure 4 by the two light grey regions inside the inferred classifier oval. Although the inferred classifier can be unsatisfied by some unobserved feasible benign states, we emphasize that this is bound to happen due to inductive inference - where we infer the patch invariant based on the observed states; it is not due to the generation of infeasible states by snapshot fuzzing. Input-level fuzzing techniques, which only generate feasible observed states and generalize the patch invariant from those states, could also result in a patch invariant that is unsatisfied by certain feasible benign states, if they have not been observed.

Mutation Strategy. Algorithm 1 generates new states s' by directly mutating an existing state selected from the current snapshot corpus S. Since the goal is to refine the current patch invariants Φ, we only mutate the visible variables observed by *snapshot logging* from Section 3.1 (e.g., local variables, members, etc.) at location $\mathcal{L}_{patch}$. Other program states (e.g., arbitrary memory addresses, etc.) are not considered by Φ, and will not be mutated.

In principle, the snapshot variables can be mutated arbitrarily within specific constraints (see below). However, we can optimize the mutation strategy with the reference to the current patch invariants Φ which are assumed to be "mostly" correct. Therefore, in addition to random perturbation, we bias mutation towards points that are closer to the *boundary* defined by Φ (the boundary between benign and vulnerable executions). Here, given a patch invariant $\phi \in \Phi$ over variables $\overrightarrow{x} = \langle x_1, .., x_n \rangle$, then $p = \langle v_1, .., v_n \rangle$ is a *boundary point* if there exists another point $p' \neq p$ such that $\phi(p)$, $\neg\phi(p')$ hold (or vice versa) and there exists no intermediate point between p and p' w.r.t. Euclidean distance. The intuition behind this strategy is that any inaccuracies within ϕ are more likely to be exposed by points close to the boundary, as opposed to arbitrary points that comfortably satisfy either ϕ or its negation. We use an SMT solver to generate boundary points in order to guide mutation.

Mutation constraints. Mutations are constrained by *variable types* and other assumptions. For example, a variable c of type `char` can only be mutated to values within the range `CHAR_MIN..CHAR_MAX`. Mutations to ghost variables are similarly constrained to make sure their semantics are preserved. For example, given a pointer `ptr`, the corresponding ghost variable `offset(ptr)` is only mutated to values within the range `0..size(ptr)`, so that `ptr` still points to the original underlying object after mutation. Note that when `offset(ptr)` is mutated, the actual value for `ptr` is also mutated to reflect the changes in `offset(ptr)`.

Object mutation. Since the ghost variable `size(ptr)` represents the size of the underlying object pointed by `ptr`, to preserve its semantics, mutation of `size(ptr)` should be accompanied by the mutation of the actual underlying object `obj` stored in memory. Mutation of the actual object can be achieved by either extending or contracting the length of `obj`. When mutating object length, it is desirable to extend/contract objects *in place* instead of moving them to other locations in memory, so that pointer values can be preserved even at program locations not captured by snapshot logging. In-place contraction can be performed by forbidding access to some bytes at end of the object, while in-place extension requires more design considerations to avoid corrupting other adjacent objects. Our implementation makes use of the *AddressSanitizer* [25] allocator for in-place extension/contraction, which automatically pads all allocations with a configurable *redzone* to detect out-of-bounds errors using memory poisoning. We exploit this technique to implement object extensions, by an in-place reallocation of (some part of) the redzone memory to extend the associated object. During this reallocation, the AddressSanitizer *shadow map* is updated and the extended region of memory is zero-initialized. Similarly, in-place contraction can be implemented by growing the associated redzone, also by updating the shadow map.

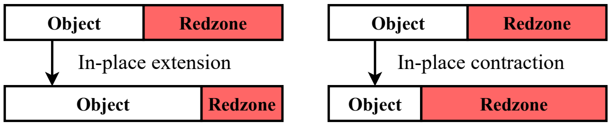

Figure 5: In-place extension and contraction.

4 IMPLEMENTATION

The current implementation of VulnFix consists of three components: (1) an *instrumentation module* for snapshot logging and mutation, (2) a *driver module* for counterexample generation, and (3) a backend for invariant inference. The instrumentation and driver modules form a frontend that generates snapshots for the backend.

Instrumentation. The instrumentation module (written in C) is built on the static binary rewriter e9patch [7]. At the patch location $\mathcal{L}_{patch}$ of the vulnerable program, the instrumentation module inserts a function to record current values of the variables in scope, and optionally mutate some of the variable values based on a given argument. To read and write program variable values at runtime, the instrumented code parses the DWARF debugging information to establish a mapping between variable names and their corresponding runtime locations.

Driver. The driver module (written in Python) invokes various components and communicates data (snapshots, patch invariants) between them. It processes the snapshots produced by the instrumentation code and classifies them based on the program execution status. It also implements the core of snapshot fuzzing for generating counterexample snapshots based on the given patch invariants and test inputs. We use z3 [6] SMT solver for finding boundary values to guide mutation.

Backend. The backend component takes in sets of benign and vulnerable snapshots and performs invariant inference based on them. The current implementation of VulnFix supports two backends: Daikon-based and cvc5-based. For the Daikon backend, we first use Daikon to infer a set of invariants Φ_0 from S_{benign}, and then perform a filtering step that only returns $\phi \in \Phi_0$ if ϕ is violated by all $s \in S_{vul}$. The filtering step is implemented on top of the Daikon `InvariantChecker` utility. Since Daikon initiates invariants based on templates, we add a few extra templates applicable for patching security vulnerabilities:

- $x - y >= a$, where x, y are variables and a is a constant;
- $x < 2^n$, where x is a variable, and 2^n is power-of-two constant representing boundary values for integers.

Cvc5 [3] is a program synthesizer, which takes as input a set of input-output pairs $\{i_1 \mapsto o_1, \ldots, i_n \mapsto o_n\}$ and synthesizes a function f such that $f(i_k) = o_k$ for $k \in \{1, \ldots, n\}$. In our context, we use the cvc5 backend to synthesize a function f, such that $f(s) = \texttt{True}$ for $s \in S_{benign}$ and $f(s') = \texttt{False}$ for $s' \in S_{vul}$. The grammar used for synthesis includes all variables in the snapshot, arithmetic operators ($+, -, \times$), relational operators ($\geqslant, \leqslant, =$), logical operators (`and`, `or`, `not`), and constants (1 to 100, power-of-two values).

Use of sanitizers. Since the snapshots need to be classified into benign and vulnerable based on observing program execution status, we use AddressSanitizer (ASan) [25] and UndefinedBehaviorSanitizer (UBSan) [2] to transform the vulnerabilities into crashes. We also read and write to the ASan redzone metadata for logging and mutating ghost variable values.

5 EVALUATION

In this section, we aim to answer the following research questions:

- RQ1: How effective is VulnFix (with different backends) in synthesizing conditions for fixing real-world CVEs?
- RQ2: What are the strengths and weaknesses of VulnFix compared with other APR tools?
- RQ3: How effective is snapshot fuzzing in refining patch invariants compared to input-level fuzzing?

Benchmark subjects. We evaluate VulnFix on a subset of the VulnLoc [27] benchmark. The VulnLoc benchmark is extended from the ExtractFix [10] and SenX [12] benchmark, and contains 43 real-world CVEs. Out of the 43 vulnerabilities in the VulnLoc benchmark, 4 vulnerabilities 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. The remaining 39 vulnerabilities are used in our evaluation.

Experiment setup. All of our experiments are performed on a 40-core 2.60GHz 64GB RAM Intel Xeon machine, Ubuntu 18.04. We note that the current implementation of VulnFix does not support parallelism and the experiments are performed with sequential algorithms. For most vulnerabilities in the evaluation, we use the following configuration: (1) the developer patch location is used as the target location to infer invariants; (2) the initial input set supplied to VulnFix only includes one exploit input obtained from online bug reports. VulnFix infers a patch invariant classifying the benign and vulnerable execution. We use the patch invariant to disable the vulnerable execution by either (1) integrating the invariant to the original condition if the target location is an `if`, `for`, or `while` statement; or (2) generating an `if`-guard in the form of

`if(!`*constraint*`) exit(ERROR_NUM);`

5.1 RQ1: Efficacy with Different Backends

We evaluated the efficacy of VulnFix with two different backends - Daikon and cvc5. Daikon uses pre-defined templates for instantiating invariant candidates and enumerates the candidates to find the ones that are satisfied on the given traces. While cvc5 synthesizes an expression based on a given grammar via Satisfiability Modulo Theories (SMT) solving. Given that cvc5 is built on top of SMT, it is less scalable than Daikon and takes more time to run especially when the number and size of snapshots grow. In order to obtain meaningful results, for each vulnerability in the benchmark, we set the total timeout to be 30 minutes for Daikon-backend and 3 hours for cvc5-backend. The first 10 minutes are allocated for the input-level fuzzing phase, and the remaining is allocated to snapshot fuzzing and invariant inference.

Since VulnFix infers a patch invariant over existing program variables (as well as ghost variables), which is then used to disable vulnerable executions, VulnFix is not applicable to some vulnerabilities in the benchmark. These vulnerabilities include those that (1) cannot be fixed by modifying or inserting conditions, or (2) require introducing new program variables that are not included in our ghost variable scheme. We identified nine such vulnerabilities according to their developer patches and marked them as "*NA*" (not applicable). These nine vulnerabilities are included in the results for completeness.

For the remaining vulnerabilities, we evaluate the correctness of the generated patches by manually comparing them with developer patches. "Correct (equiv)" means that the result of VulnFix is semantically equivalent to the developer patch; "Correct (not equiv)" means that the produced patch is not semantically equivalent to the developer patch, but still correctly fixes the vulnerability (see examples in the following). "Wrong" means that VulnFix fails to produce a correct patch before timeout. We only regard a result as correct if it is the **only** patch produced by VulnFix and the produced patch correctly fixes the vulnerability.

Results. Table 2 shows the evaluation results, where columns "Daikon backend" and "cvc5 backend" list the result of VulnFix when the corresponding backend is used. Overall, both backends

Table 2: Experimental results of VulnFix (with different backends), CPR, and SenX on the VulnLoc benchmark.

Subject	Bug ID	VulnFix				CPR		SenX
		Daikon backend	Correct?	cvc5 backend	Correct?	Rank	Ratio	Correct?
Binutils	CVE-2017-6965	Correct (not equiv)	✓	Correct (not equiv)	✓	Timeout	Timeout	✗
Binutils	CVE-2017-14745	Wrong	✗	Correct (equiv)	✓	109	0%	✗
Binutils	CVE-2017-15020	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*	✗
Binutils	CVE-2017-15025	Correct (equiv)	✓	Correct (equiv)	✓	36	0%	*NA*
Coreutils	Gnubug-19784	Correct (equiv)	✓	Correct (equiv)	✓	393	0%	✗
Coreutils	Gnubug-25003	Correct (not equiv)	✓	Wrong	✗	29	59%	✗
Coreutils	Gnubug-25023	Correct (not equiv)	✓	Correct (not equiv)	✓	56	0%	✗
Coreutils	Gnubug-26545	Wrong (not spt)	✗	Wrong	✗	168	46%	✗
Jasper	CVE-2016-8691	Correct (equiv)	✓	Correct (equiv)	✓	1	75%	*NA*
Jasper	CVE-2016-9557	Wrong (not spt)	✗	Wrong	✗	Timeout	Timeout	✗
Libarchive	CVE-2016-5844	Correct (not equiv)	✓	Correct (not equiv)	✓	31	54%	✗
Libjpeg	CVE-2012-2806	Correct (equiv)	✓	Correct (equiv)	✓	24	50%	✗
Libjpeg	CVE-2017-15232	Correct (equiv)	✓	Correct (equiv)	✓	36	0%	*NA*
Libjpeg	CVE-2018-14498	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*	✗
Libjpeg	CVE-2018-19664	Wrong	✗	Wrong	✗	1	48%	✗
Libming	CVE-2016-9264	Correct (equiv)	✓	Correct (equiv)	✓	39	57%	✗
Libming	CVE-2018-8806	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*
Libming	CVE-2018-8964	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*
Libtiff	Bugzilla-2611	*NA*	*NA*	*NA*	*NA*	1	61%	*NA*
Libtiff	Bugzilla-2633	Wrong	✗	Wrong	✗	46	48%	✓
Libtiff	CVE-2016-5321	Correct (equiv)	✓	Correct (equiv)	✓	11	47%	✗
Libtiff	CVE-2016-9273	*NA*	*NA*	*NA*	*NA*	57	48%	✗
Libtiff	CVE-2016-9532	Wrong (not spt)	✗	Wrong	✗	Timeout	Timeout	✗
Libtiff	CVE-2016-10092	*NA*	*NA*	*NA*	*NA*	5	0%	✗
Libtiff	CVE-2016-10094	Wrong (not spt)	✗	Wrong (not spt)	✗	27	57%	✓
Libtiff	CVE-2016-10272	*NA*	*NA*	*NA*	*NA*	5	0%	✗
Libtiff	CVE-2017-5225	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*	✓
Libtiff	CVE-2017-7595	Correct (equiv)	✓	Correct (equiv)	✓	1	48%	*NA*
Libtiff	CVE-2017-7599	Wrong (not spt)	✗	Wrong (not spt)	✗	8	0%	✗
Libtiff	CVE-2017-7600	Wrong (not spt)	✗	Wrong (not spt)	✗	45	0%	✗
Libtiff	CVE-2017-7601	Correct (not equiv)	✓	Correct (not equiv)	✓	56	48%	✗
Libxml2	CVE-2012-5134	Correct (equiv)	✓	Correct (equiv)	✓	36	49%	✗
Libxml2	CVE-2016-1838	Wrong (not spt)	✗	Wrong	✗	31	0%	✗
Libxml2	CVE-2016-1839	Correct (equiv)	✓	Correct (equiv)	✓	79	0%	✗
Libxml2	CVE-2017-5969	Correct (equiv)	✓	Correct (equiv)	✓	3	45%	*NA*
Potrace	CVE-2013-7437	Correct (equiv)	✓	Correct (equiv)	✓	Error	Error	✓
Zziplib	CVE-2017-5974	Correct (not equiv)	✓	Correct (not equiv)	✓	130	0%	✗
Zziplib	CVE-2017-5975	Correct (equiv)	✓	Correct (equiv)	✓	36	0%	✗
Zziplib	CVE-2017-5976	Wrong	✗	Wrong	✗	Timeout	Timeout	✗
Total	-	-	19/39	-	19/39	4/39	-	4/39

show similar results in producing correct patches (both produce 19 correct patches). On CVE-2017-14745, Daikon backend fails because it produces two patches in the end, while cvc5 backend produces exactly one correct patch. On Gnubug-25003, cvc5 backend fails while Daikon backend produces the correct patch.

VulnFix produces correct but not equivalent patches on six vulnerabilities. The main reason is that the patch produced by VulnFix is strictly based on whether a vulnerable program behavior is observed, while the developer patch may also take insights from program-specific semantic information. For example, Libtiff consists of an integer overflow vulnerability (CVE-2017-7601), and its relevant code snippet is shown in Figure 6. The bug is triggered when the value of `td->td_bitspersample` is greater than 62, causing the left shift on line 10 to overflow. The developer patch on lines 4-6 adds a check on its value and returns if the value is too big, with the bound 16 chosen based on file format specification. On the other hand, VulnFix produces the patch invariant `td->td_bitspersample <= 62`, where 62 is the maximum value allowed for the left shift on line 10 to not overflow. In this case, VulnFix produces a patch that correctly separates the benign and vulnerable behaviors, while the developer patch additionally considers other program semantics.

As discussed in Section 3.2, it is possible that the patch invariant generated from inductive inference is unsatisfied by certain unobserved feasible benign states. Such patch variants can lead to patches that disable more feasible program behavior than desired, thereby changing the benign functionality of the program. To understand the effect of such patch invariants experimentally, we

```
switch (sp->photometric) {
  case PHOTOMETRIC_YCBCR:
  ...
+ if( td->td_bitspersample > 16 ) {
+   return (0);
+ }
  {
    float *ref;
    if (!TIFFGetField(tif, TIFFTAG, &ref)) {
      top = 1L << td->td_bitspersample;
           // !integer overflow!
    }
  }}
```

Figure 6: Simplified code snippet of CVE-2017-7601.

examined the six correct but not equivalent patches, and found one of them (CVE-2017-6965) restricts more behavior than the developer patch. For this CVE, we applied the VulnFix patch and developer patch to the vulnerable program, and then conducted a 24-hour differential fuzzing campaign to check whether the two patches exhibit different behaviors. After 24 hours of fuzzing, there were no input executions that evaluate the VulnFix patch and developer patch differently, which means no significant restriction of benign functionality was observed from our experimentation.

Besides, there are 11 vulnerabilities marked as "Wrong" or "Wrong (not spt)". "Wrong (not spt)" means that the current VulnFix implementation does not support generating the correct invariant. For example, the correct invariant for CVE-2016-9532 involves inequality with scalar multiplication (e.g., `x * y * z <= constant`), which is not supported by Daikon. Daikon infers invariants based on a set of templates, and invariants that cannot be represented as one of the templates cannot be inferred. As cvc5 synthesizes invariants based on grammar instead of fixed templates, it was expected that cvc5 outperforms Daikon on these vulnerabilities. However, the experimental result shows otherwise: cvc5 backend fails to produce correct patches on the vulnerabilities that Daikon does not support (marked as "Wrong"). This is because these patches usually consist of complex expressions, and cvc5 backend times out before synthesizing such expressions.

> Overall, the Daikon and cvc5 backend each produced 19 correct patches, with a time budget of 30 minutes and 3 hours. From this comparison, Daikon appears to be a more practical backend.

5.2 RQ2: Comparison with Other APR Tools

Comparison with CPR. To understand the strength and weakness of VulnFix in repairing security vulnerabilities, we perform a comparison with CPR [26], a state-of-the-art program repair tool. CPR works by first synthesizing a pool of patch candidates from a given set of patch ingredients, then discarding overfitting patches from the pool by exploring the input space with concolic execution, and finally ranking the remaining patches. This workflow is conceptually similar to counterexample-guided inductive synthesis (CEGIS), i.e., infer initial candidates and then generate new test inputs (counterexample) to rule out incorrect candidates.

In our experiments, we set the timeout for each vulnerability to be 30 minutes for CPR. Since CPR requires patch ingredients to be provided for patch synthesis, we supply five variables at the patch location (including all the variables used in the developer patch) and necessary arithmetic/comparison operators as patch ingredients to the synthesizer used in CPR. Besides, since CPR currently repairs only boolean and integer expressions [26], and does not automatically introduce new program variables, it is not applicable to some vulnerabilities in the benchmark. We identified five such vulnerabilities and marked them as "*NA*" (not applicable).

Results. The evaluation results of CPR are shown in the "CPR" columns in Table 2. Column "Rank" shows the rank of the correct patch in the final patch pool. Column "Ratio" shows the patch pool reduction ratio, which is the percentage of initial patches that are discarded by co-exploration of the patch space and input space. "Timeout" indicates that CPR did not generate patches before the 30-minute timeout, and "Error" indicates that an error occurred during concolic execution and CPR aborted. Overall, with a 30-minute timeout, CPR ranks the correct patch at the top-1 position for four out of 39 vulnerabilities. For 16 vulnerabilities, CPR discards more than 40% of the initial patch candidates by performing concolic execution. However, for 13 other vulnerabilities, CPR has 0% patch space reduction potentially due to the longer paths from loop unrolling [26]. In other words, concolic execution cannot find any test input that can reach patch location or discard plausible patches. Instead of relying on concolic execution, VulnFix performs snapshot mutation to discard overfitting patch invariants, which is shown to be more efficient based on the experimental results.

Comparison with SenX. We also performed a comparison with the security vulnerability repair tool SenX [12], which generates patches based on a pre-specified set of safety properties. The same benchmark consisting of 39 vulnerabilities is used, and the timeout for each vulnerability is set to be 30 minutes. Since SenX currently only supports repairing buffer overflows, bad casts, and integer overflows [12], vulnerabilities that are not of these types are not applicable. There are eight such vulnerabilities in the benchmark, which are marked as "*NA*" (not applicable).

Results. The evaluation results of SenX are shown in the "SenX" column in Table 2. Overall, SenX produces correct patches for four out of 39 vulnerabilities. For the remaining vulnerabilities that are applicable to SenX, either no patch was produced, or the produced patch did not correctly fix the vulnerability.

> VulnFix generated 19 correct patches with 30 minutes for each, while CPR and SenX each produced four correct patches (by just checking the top-ranked patch for CPR).

5.3 RQ3: Comparison with Input-Level Fuzzing

To understand whether snapshot fuzzing can generate states that refine the invariants effectively, we also compare it with traditional input-level fuzzing. Specifically, we replace the snapshot fuzzing step in VulnFix with traditional input-level fuzzing techniques and compare their effectiveness in generating correct patches. For the tests generated by input-level fuzzing, we collect the benign/vulnerable snapshots by considering the non-redundant tests that can reach the fix location.

Table 3: Comparison with input-level fuzzing, where VulnFix^C represents replacing the snapshot fuzzing module with ConcFuzz, while VulnFix^A means that snapshot fuzzing is replaced by AFL.

Bug ID	VulnFix		VulnFix^C		VulnFix^A	
	#Inv	result	#Inv	result	#Inv	result
CVE-2017-6965	1	✓	1	✗	1	✓
CVE-2017-14745	2	✗	0	✗	5	✗
CVE-2017-15025	1	✓	0	✗	5	✗
Gnubug-19784	1	✓	1	✓	1	✓
Gnubug-25003	1	✓	34	✗	23	✗
Gnubug-25023	1	✓	8	✗	7	✗
Gnubug-26545	0	✗	1	✗	0	✗
CVE-2016-8691	1	✓	25	✗	17	✗
CVE-2016-9557	0	✗	0	✗	0	✗
CVE-2016-5844	1	✓	0	✗	60	✗
CVE-2012-2806	1	✓	6	✗	6	✗
CVE-2017-15232	1	✓	0	✗	15	✗
CVE-2018-19664	2	✗	0	✗	18	✗
CVE-2016-9264	1	✓	4	✗	6	✗
Bugzilla-2633	2	✗	0	✗	50	✗
CVE-2016-5321	1	✓	3	✗	5	✗
CVE-2016-9532	36	✗	38	✗	36	✗
CVE-2016-10094	9	✗	24	✗	23	✗
CVE-2017-7595	1	✓	14	✗	3	✗
CVE-2017-7599	0	✗	0	✗	0	✗
CVE-2017-7600	0	✗	0	✗	0	✗
CVE-2017-7601	1	✓	2	✗	1	✗
CVE-2012-5134	1	✓	6	✗	4	✗
CVE-2016-1838	3	✗	0	✗	3	✗
CVE-2016-1839	1	✓	0	✗	1	✓
CVE-2017-5969	1	✓	1	✓	1	✓
CVE-2013-7437	1	✓	0	✗	1	✓
CVE-2017-5974	1	✓	8	✗	5	✗
CVE-2017-5975	1	✓	0	✗	1	✓
CVE-2017-5976	0	✗	0	✗	0	✗
Total	-	19/30	-	2/30	-	6/30

We consider two input-level fuzzing tools: AFL [32] and ConcFuzz [27]. AFL is a widely used grey-box fuzzing tool, which has been proved to be efficient in detecting software vulnerabilities and bugs. For AFL, we re-use the modified version described in Section 3.1 to generate input tests. ConcFuzz "concentrates" on the neighborhood of the given exploit. Specifically, it builds a "concentrated" test suite that drives the program execution to reach each branch location of the given exploit execution trace. Based on the "concentrated" test suite, ConcFuzz can then estimate the probability of each branch being executed by vulnerable inputs and hence determine the fault locations. For the application of invariant inference, exploring the neighborhood of patch location instead of the entire trace is sufficient. Therefore, we implement a modified version of ConcFuzz that only "concentrates" on the patch location. In the experiment, we set a 30-minute timeout for both AFL and ConcFuzz, which is the same as the total time budget for VulnFix. The nine vulnerabilities in the benchmark which are not applicable to VulnFix are excluded from this experiment, as they are also not applicable when snapshot fuzzing is replaced by input-level fuzzing techniques.

Results. The evaluation results of input-level fuzzing are shown in Table 3. Column "VulnFix^C" represents the result when replacing the snapshot fuzzing module in VulnFix with ConcFuzz, while "VulnFix^A" is the result when snapshot fuzzing is replaced by AFL. The column "#Inv" shows the number of invariants produced when the time budget is exhausted and the column "result" indicates whether a **single** correct patch is produced in the end. Overall, VulnFix produces 19 correct patches out of 39 vulnerabilities, while VulnFix^C and VulnFix^A only produce two and six correct patches, respectively. VulnFix^C and VulnFix^A produce very few correct patches because (1) they generate multiple candidate invariants (#Inv is greater than 1), and some of them are incorrect; or (2) although they produce only one candidate invariant on some vulnerabilities, the produced invariant is incorrect and overfits to the generated test inputs. In contrast, directly mutating snapshots enables VulnFix to generate fewer but more precise invariants and hence more correct patches.

> Compared to input-level fuzzing AFL and ConcFuzz, snapshot fuzzing enables VulnFix to generate fewer but more precise invariants and hence more correct patches.

5.4 Threats to Validity

A few threats may affect the validity of our evaluation. The main threat to validity is that the correctness of generated invariants/-patches cannot be guaranteed. Although snapshot fuzzing can explore the program states in a more controlled way, it still cannot ensure that all reachable program states at a fix location are exhaustively explored. Fortunately, the incompleteness does not seem to have a big impact on the effectiveness of VulnFix. The second threat is that we manually inspect whether the generated patches are semantically equivalent to developer patches, which might be error-prone. To mitigate this, two authors of the paper double-checked the generated patches.

Another threat to validity is that our selection of subject programs may not generalize to all programs. To mitigate this threat we used a data-set of subjects developed in a previous work [27]. We evaluated our technique on this existing dataset (filter out some vulnerabilities that cannot be reproduced). In the future, it may be worthwhile to evaluate VulnFix on more CVEs and vulnerabilities.

6 RELATED WORK

The contributions of this paper are related to several areas of research: automated program repair, vulnerability repair and counterexample guided invariant inference. In this section, we present the related work as follows.

Automated program repair. Automated program repair techniques take in a buggy program, and a set of specifications, and aim to generate a patched program satisfying the given specifications [16]. Test-driven automated program repair treats the provided test suite as the specification of intended behavior and generates patches to make the patched program pass all the given tests [15, 18, 19, 21].

Since test cases are incomplete program specifications, the generated patches may overfit the given tests, i.e., the patched program works on the given tests but cannot be generalized to other tests. VulnFix is designed to alleviate the overfitting problem.

Existing work alleviates the overfitting issue by ranking the patches according to their probability of being correct [14, 18], referring to reference implementation [20] or designing customized repair strategies [28]. Typically, those approaches try to generate correct patches by referring to additional program artifacts. Compared with those approaches, VulnFix does not rely on additional inputs (such as reference programs), which gives VulnFix more flexibility. Besides, some approaches alleviate overfitting problem by generating more test cases [9, 30]. Compared to those approaches that generate test inputs, snapshot fuzzing directly mutates program states, which enables VulnFix to bypass the reachability problem in test case generation.

Vulnerability repair. In recent years, we have seen a rising trend of research on automatically fixing vulnerabilities. SenX [12] aims to repair vulnerabilities relying on vulnerability-specific and human-specified safety properties. Some other repair approaches are designed to repair a specific type of vulnerabilities, such as fixing memory errors [17] or concurrency bugs [13]. Compared to SenX and these approaches which are limited to specific classes of bugs, VulnFix does not rely on pre-defined safety properties and is not limited to certain vulnerabilities. ExtractFix [10] fixes vulnerabilities by first inferring crash-free constraints, propagating the constraints to fix location, and synthesizing patches to satisfy the constraints. CPR [26] fixes vulnerabilities by (1) generating a candidate patch space, and (2) detecting and discarding overfitting patches via a systematic co-exploration of the patch space and input space. It leverages concolic execution to systematically traverse the input space (and generate inputs), and uses the produced test inputs to rule out the overfitting patches from the patch space. Compared to ExtractFix and CPR, VulnFix does not rely on heavy symbolic and concolic executions, enabling it to scale to large programs.

Counter-example guided invariant inference. Recent works (e.g., PIE [24], ICE [11], CEGIR [22]) present *CounterExample Guided Invariant geneRation* (CEGIR), i.e., infer a initial set of candidate invariants and then improve them using counterexamples. Specifically, if an initial invariant is invalid for some input, these approaches search for counterexamples which can help to refine the invariant. Such approaches are more efficient than traditional dynamic or static invariant inference, However, they still cannot get rid of the dependence on heavy program analysis. For instance, they still rely on symbolic execution or concolic execution [22, 33] to discover counterexamples. Instead of relying on heavy symbolic analysis, VulnFix investigates using light-weight input/state generation to verify the candidate invariants. Therefore, VulnFix is largely independent of the complexity or size of the programs and thus can scale to large programs.

7 DISCUSSION

In this work, we have presented an approach for automatically repairing program vulnerabilities from a single exploiting test input. Our approach is based on obtaining more states at the fix location via state mutations, and inductively inferring a likely invariant, which is then used to construct patches. Evaluation on a previously proposed data-set of vulnerabilities show higher effectiveness compared to state-of-the-art vulnerability repair engines like SenX and CPR. While our approach is currently focused on fixing vulnerabilities, it shows that inductive inference approaches can be promising for general-purpose program repair. This would contrast with deductive or symbolic approaches for program repair [21] which deduce a repair constraint by symbolically analyzing a *given* test-suite.

Open Source Release
https://github.com/yuntongzhang/vulnfix

ACKNOWLEDGMENT

We thank the anonymous reviewers for insightful suggestions. This research is partially supported by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems).

REFERENCES

[1] 2022. *LibFuzzer.* https://llvm.org/docs/LibFuzzer.html

[2] 2022. *UndefinedBehaviorSanitizer.* https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[3] Clark Barrett, Cesare Tinelli, and et al. 2022. *CVC5.* https://cvc5.github.io

[4] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* New York, NY, USA, 678–689. https://doi.org/10.1145/3368089.3409748

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2329–2344. https://doi.org/10.1145/3133956.3134020

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 151–163. https://doi.org/10.1145/3385412.3385972

[8] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[9] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 8–18. https://doi.org/10.1145/3293882.3330558

[10] 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 Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27. https://doi.org/10.1145/3418461

[11] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512. https://doi.org/10.1145/2914770.2837664

[12] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP).* IEEE, 539–554. https://doi.org/10.1109/SP.2019.00071

[13] Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated concurrency-bug fixing. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12).* 221–236.

[14] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224. https://doi.org/10.1109/SANER.2016.76

[15] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72. https://doi.org/10.1109/TSE.2011.104

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

[17] 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*. 95–106. https://doi.org/10.1145/3236024.3236079

[18] 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*. 298–312. https://doi.org/10.1145/2837614.2837617

[19] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018). https://doi.org/10.1145/3241980

[20] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 129–139. https://doi.org/10.1145/3180155.3180247

[21] Hoang D.T. Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2013.6606623

[22] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 605–615. https://doi.org/10.1145/3106237.3106281

[23] ThanhVu Huy Nguyen. 2014. *Automating program verification and repair using invariant analysis and test input generation*. The University of New Mexico.

[24] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56. https://doi.org/10.1145/2980983.2908099

[25] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[26] 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*. 390–405. https://doi.org/10.1145/3453483.3454051

[27] 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*. 537–549. https://doi.org/10.1145/3433210.3437528

[28] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 727–738. https://doi.org/10.1145/2950290.2950295

[29] 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

[30] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 226–236. https://doi.org/10.1145/3092703.3092718

[31] J Xuan, M Martinez, F Demarco, M Clement, SL Marcote, T Durieux, D Le Berre, and M Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43 (2016). Issue 1. https://doi.org/10.1109/TSE.2016.2560811

[32] Michał Zalewski. 2022. *American fuzzy lop*. https://lcamtuf.coredump.cx/afl/

[33] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. 2014. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 362–372. https://doi.org/10.1145/2610384.2610389