

SYMRADAR: PoC-Centered Bounded Verification for Vulnerability Repair

Seunghoon Han

shhan@unist.ac.kr

UNIST

South Korea

Yeseung Lee

yeseung.lee@unist.ac.kr

UNIST

South Korea

YoungJae Kim

kyj1411@unist.ac.kr

UNIST

South Korea

Jooyong Yi

jooyong@unist.ac.kr

UNIST

South Korea

Abstract

In this paper, we tackle the problem of patch verification. While automated vulnerability repair (AVR) techniques are gaining traction, it is not sufficient to merely generate patches; providing evidence of their correctness is also essential. However, the current state-of-the-art patch verification methods are not sufficiently effective. To address this issue, we present SYMRADAR, a patch verification tool based on under-constrained symbolic execution (UC-SE). What distinguishes SYMRADAR from existing patch verification techniques is its use of function-level symbolic execution with inputs centered around the provided proof-of-concept (PoC) input. As demonstrated in our evaluation, this PoC-centered symbolic execution is effective, achieving the highest recall (100%) and specificity (78%) among all compared techniques.

CCS Concepts

• Software and its engineering → Formal software verification.

Keywords

Security vulnerabilities, Automated Vulnerability Repair, Patch Verification, Symbolic Execution

ACM Reference Format:

Seunghoon Han, YoungJae Kim, Yeseung Lee, and Jooyong Yi. 2025. SYMRADAR: PoC-Centered Bounded Verification for Vulnerability Repair. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*, April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Timely patching of security vulnerabilities is crucial for maintaining the security of software systems. Failing to do so can lead to

severe consequences, such as financial losses [6], reputation damage [12], and even human death [56] and cyber warfare [19]. With the advancement of automated vulnerability detection techniques such as fuzzing, finding security vulnerabilities is becoming easier. However, the process of patching these vulnerabilities is still largely manual. This technological gap between automated vulnerability detection and manual patching can paradoxically leave the software ecosystem more susceptible to attacks. To fill this gap, researchers have proposed *automated vulnerability repair (AVR)* techniques [10, 16, 22, 28, 48, 64, 65], which automatically generate patches for security vulnerabilities.

AVR is a specialized form of APR (Automated Program Repair) [21, 39] that specifically targets security vulnerabilities. Just as test-driven APR tools generate patches that pass a provided test suite, many AVR tools [18, 28, 48, 64] generate patches that pass a given vulnerability-exposing input, such as a proof-of-concept (PoC) input. Consequently, these AVR tools are subject to the well-known “overfitting” problem [50] common in APR. That is, an AVR tool may generate an incorrect patch that merely prevents the vulnerability from manifesting for the provided input, leaving the vulnerability exploitable for other inputs. *Verifying the correctness of automatically generated vulnerability repairs is necessary, which is the subject of this paper.*

Existing Relevant Techniques. Many existing AVR tools perform patch verification beyond the provided PoC input. CPR [48] introduced a promising patch verification approach to using a PoC input: it explores the input space near the provided PoC to verify the generated patches. This PoC-centered verification can improve both the effectiveness of patch verification and confidence in the verification results. To enable this approach, CPR performs concolic execution at the system level. However, the system-level concolic execution often fails to reach the patch location. The same research group who proposed CPR mitigates this issue in another AVR tool, VULNFix [64], by performing function-level fuzzing. It executes the patched program with the PoC and mutates the program state at the entry point of the patched function. However, the level of confidence in the patch’s correctness is reduced by replacing concolic execution with fuzzing.

Meanwhile, there are also AVR approaches based on static analysis [20, 55, 63]. These tools determine the correctness of a patch based on whether vulnerability warnings produced by static analyzers disappear after applying the patch. However, industry-grade

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '26, Rio de Janeiro, Brazil

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

static analyzers, such as INFER [4], are neither sound nor complete [1]. As a result, AVR tools built on top of such static analyzers may miss vulnerabilities that can be manifested by a PoC input or produce incorrect patches [20].

Deep-learning techniques can also be used to verify patches [9, 33, 52]. These techniques classify a given function as vulnerable or safe by learning patterns of vulnerable and safe functions from large code corpora. However, current approaches classify functions as vulnerable or safe in isolation without considering their calling contexts. As a recent empirical study [44] has pointed out, this is problematic since the same function can be vulnerable in one calling context but safe in another.

Our Approach. In this work, we propose a novel patch verification technique named SYMRADAR designed to achieve both high efficacy and confidence in patch verification. To achieve our goals, SYMRADAR performs PoC-centered verification at the function-level using well-known under-constrained symbolic execution (UC-SE) [43], instead of fuzzing. However, since existing UC-SE does not support PoC-centered verification, SYMRADAR extends UC-SE to enable this.

Figure 1 provides a high-level comparison between SYMRADAR and the existing UC-SE. Suppose a crash-inducing PoC input i_c is given to an AVR tool, which then generates a patch for a vulnerable function f . Most AVR tools [18, 28, 48, 64] fix a single function, and our work also focuses on verifying a patched function. As shown in the right part of the figure, UC-SE performs bounded verification around the “uninitialized” input state denoted by i_\perp , which is disconnected from i_c . In contrast, SYMRADAR performs bounded verification around $i_c \llbracket f \rrbracket$, the snapshot (i.e., program state) observed at the entry point of f before the crash occurs. To achieve this, SYMRADAR converts the obtained concrete snapshot $i_c \llbracket f \rrbracket$ into the corresponding abstract snapshot $i_c \llbracket f \rrbracket$ and performs symbolic execution with $i_c \llbracket f \rrbracket$; see § 3 for details.

We evaluated SYMRADAR on 3,037 patches from 28 distinct vulnerable programs. While all these patches prevent the crash when tested with the given PoC inputs, only some of them are correct, motivating the need for patch verification. We performed patch verification on these patches using SYMRADAR and four other verification techniques (CPR [48], VULNFix [64], SPIDER [35], and UC-KLEE [43]). The results show that SYMRADAR detects all correct patches (i.e., 100% recall), showing the highest recall among the compared techniques. SYMRADAR also identifies 78% of incorrect patches (i.e., 78% specificity), which is also the highest among the compared techniques. Overall, SYMRADAR balances recall and specificity, achieving the highest balanced accuracy (89%) among the compared techniques.

We find that the key to SYMRADAR’s high performance is its PoC-centered verification. SYMRADAR generates inputs that are highly relevant to the PoC, thus effectively detecting incorrect patches that are overfitted to the PoC. *Most of all, the fact that SYMRADAR performs effective bounded verification is SYMRADAR’s unique strength, as it guarantees the correctness of a patch within a clear bound.* In summary, we make the following contributions in this paper:

- (1) **Novel Patch Verification Technique.** We propose SYMRADAR, the first bounded patch verification technique that performs

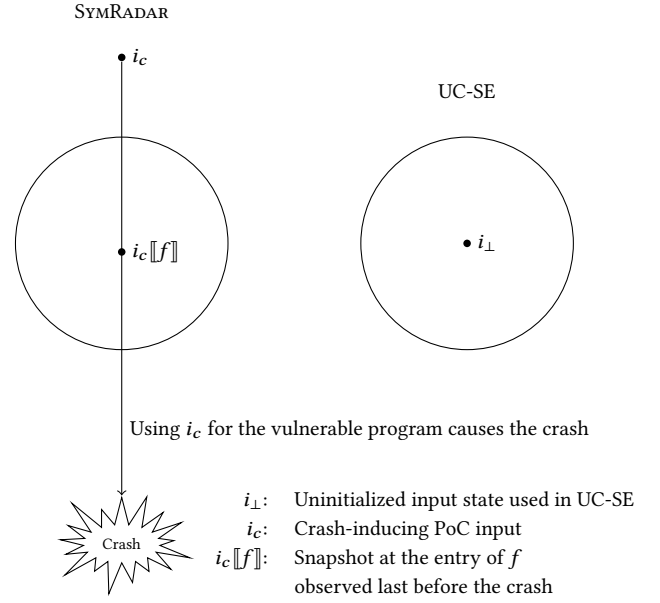


Figure 1: Comparison between SYMRADAR and UC-SE

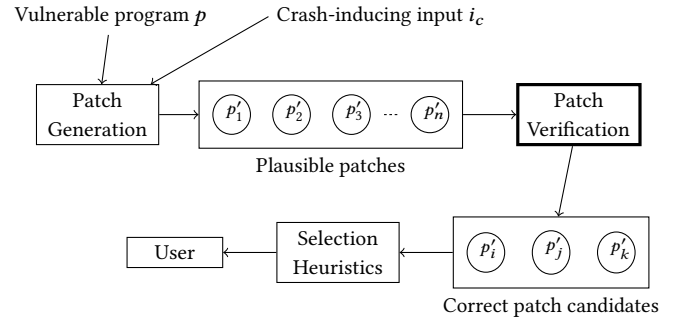


Figure 2: A common workflow of AVR. Patch verification is the subject of this paper.

PoC-centered verification on the patched function. By exhaustively exploring the input space around the PoC input, SYMRADAR provides strong assurance of patch correctness.

- (2) **Comparison with Existing Techniques.** We provide empirical evidence of SYMRADAR’s effectiveness by comparing it with four state-of-the-art patch verification techniques. SYMRADAR achieves the highest recall (100%) and specificity (78%) among the compared techniques, demonstrating its effectiveness in patch verification.
- (3) **Replication Package.** A replication package—including the SYMRADAR implementation and experimental scripts to reproduce our results—is available at:

<https://github.com/UNIST-LOFT/symradar>

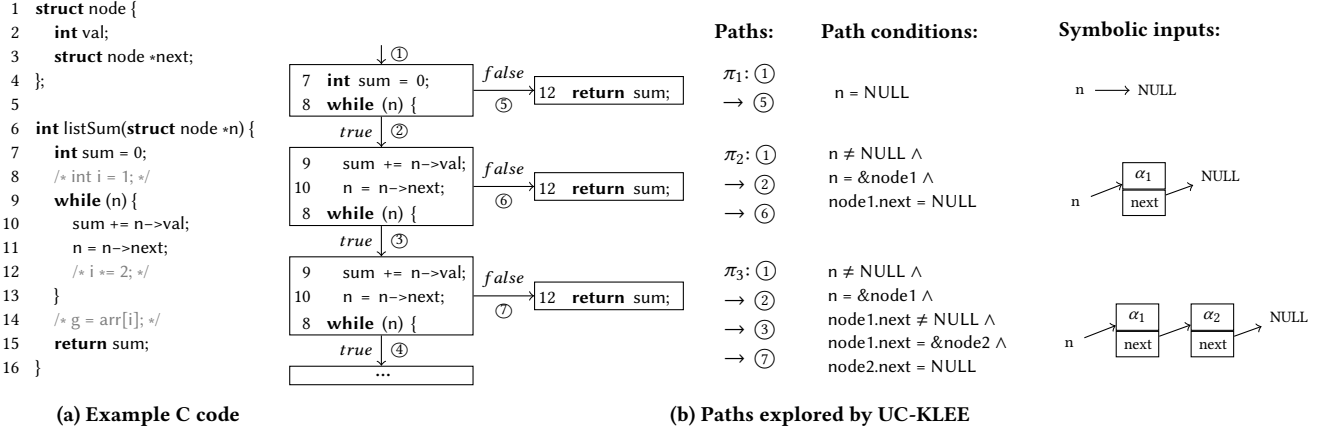


Figure 3: How UC-KLEE [43], a representative UC-SE tool, symbolically executes the listSum function.

2 Background

2.1 Automated Vulnerability Repair (AVR)

Figure 2 illustrates the typical workflow of AVR used in many AVR tools [18, 28, 48, 64].¹ These tools take as input a vulnerable program p and a crash-inducing input i_c , such as a Proof of Concept (PoC). In this paper, we use the term PoC to refer to a crash-inducing input. A PoC can be obtained externally, for example, from a fuzzer.

Given p and i_c , the first step of AVR is to generate a set of “plausible” patches,² all of which avoid the crash when tested with i_c .³ Since the number of plausible patches is often large, many AVR tools perform a patch verification step to filter out incorrect patches [18, 28, 48, 64]. Only the remaining patches are considered as candidates for the correct patch. Optimal patch verification should filter out all incorrect patches while retaining the correct one. However, the patch verification capability of existing AVR tools is limited and often fails to filter out many incorrect patches, as shown in our evaluation (see § 6). As a result, these tools rely on patch selection heuristics such as ranking. In this work, we show that more effective and trustworthy patch verification is possible.

2.2 Under-Constrained Symbolic Execution

We develop our approach based on under-constrained symbolic execution (UC-SE). UC-SE performs symbolic execution directly on the target function. Figure 3 illustrates how UC-KLEE [43], a representative UC-SE tool, symbolically executes the target function, listSum. This function computes the sum of the values in a linked list, with its head pointed to by the input parameter n . For now, please ignore the three commented lines.

UC-SE explores all possible inputs exhaustively, up to a certain bound. In our example, UC-SE considers multiple linked lists of different lengths (i.e., 0, 1, and 2), as shown on the right side of Figure 3. To achieve this, UC-SE employs a technique known as lazy

initialization [24]. Initially, the input parameter n is “uninitialized.” When n is accessed at line 9, UC-KLEE “lazily” initializes it to either NULL (in symbolic execution path π_1) or a new object of type node (in paths π_2 and π_3). Similarly, when $n \rightarrow \text{next}$ is accessed at line 11 during the first loop iteration, UC-KLEE initializes it to either NULL (in path π_2) or a new node object (in path π_3). To limit the exploration space, UC-KLEE bounds the length of these lazy initialization chains; this technique is known as k -bounding [13, 14]. In our example, the bound k is set to 2.

Limitations of UC-SE. Now suppose we uncomment the three commented lines in Figure 3. If the linked list has more than 30 nodes, the multiplication at line 12 causes an overflow, and executing $\text{arr}[i]$ at line 14 results in unsafe memory access via a negative array index. While this example is artificial, it illustrates the limitations of UC-SE. If a bound is not large enough, UC-SE cannot reproduce the overflow. Even with a sufficiently large bound, it generates many smaller linked lists before encountering the overflow. In practice, a small bound is typically used. For example, UC-KLEE uses a bound of 9 by default. In the next section, we describe how we perform bounded verification with inputs that are more relevant to the PoC input, instead of wasting time on irrelevant inputs.

3 PoC-Centered Bounded Verification

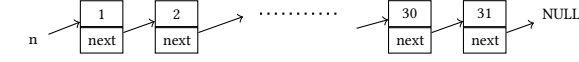
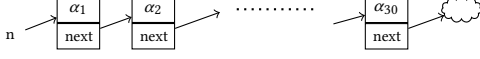
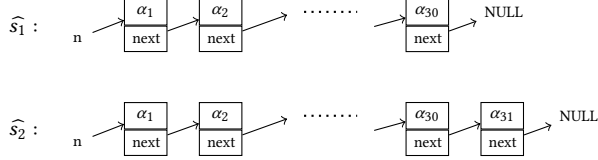
In this section, we present our approach, SYMRADAR. Unlike existing UC-SE, which explores the input space starting from an empty state, SYMRADAR is designed to explore the input space surrounding a particular input such as PoC input. To achieve this, SYMRADAR performs the following three steps: (1) concrete snapshot extraction (§ 3.1), (2) abstract snapshot construction (§ 3.2), and (3) patch verification (§ 3.3).

To illustrate these steps, consider a scenario where executing a main function with a certain user input i_c constructs a linked list containing 31 nodes, which is then passed to the function listSum as an argument. The listSum function is shown in Figure 3(a). Running this program with input i_c causes a crash at line 14 due to an out-of-bounds array access. Suppose the following patch is applied to line 14 to fix this crash:

¹Some works [10, 16, 65] describe only the patch generation step, but they can be extended to support the patch verification step as well.

²In APR, patches that pass the available tests are called plausible patches [34].

³A set of plausible patches can be maintained either explicitly [48] or implicitly [64], for example, using constraints that the patches must satisfy.

(a) Step 1: Taking a concrete snapshot s (b) Step 2: Constructing the abstract snapshot \hat{s} for s 

(c) Step 3: Performing patch verification. Executing the patched listSum function with the abstract snapshot \hat{s} results in two extended inputs, \hat{s}_1 and \hat{s}_2 , where the symbolic object cloud in \hat{s} is initialized to NULL and a new node, respectively. In this example, the lazy initialization bound is set to 1.

Figure 4: Three steps of SYMRADAR explained with the linked list example shown in Figure 3

```

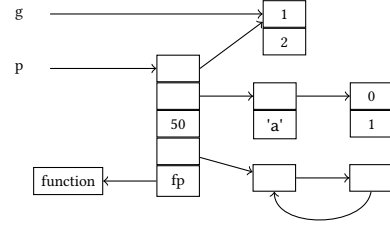
- g = arr[i];
+ if (i >= 0) g = arr[i];

```

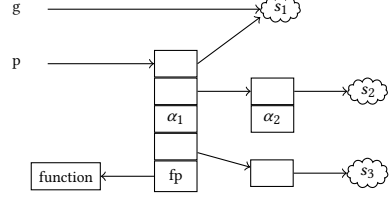
Using SYMRADAR, we perform the following three steps to verify this patch. First, we extract a concrete snapshot s at the entry point of the listSum function, where the patch is applied. Figure 4(a) shows the concrete snapshot s . Next, we construct an abstract snapshot \hat{s} from s , as shown in Figure 4(b). Notice that the last node of the linked list is replaced with an uninitialized pointer, denoted by cloud . In addition, all primitive type values, such as 1, 2, and 30, are replaced with symbolic variables, α_1 , α_2 , and α_{30} . Finally, we verify the patch by performing symbolic execution with the abstract snapshot \hat{s} . To accomplish this, we first run the original buggy listSum function symbolically using the abstract snapshot \hat{s} as the initial input. Similar to UC-KLEE (see § 2.2), we lazily initialize cloud during symbolic execution when it is accessed. Figure 4(c) shows two extended inputs, \hat{s}_1 and \hat{s}_2 , where the symbolic object cloud in \hat{s} is initialized to NULL and a new node containing α_{31} , respectively. In this example, the lazy initialization bound is set to 1. Next, we run the patched function with these obtained inputs. For both inputs, neither the crash nor the regression error occurs, so the patch is considered correct.

3.1 Concrete Snapshot Extraction

A concrete snapshot at a program location l is the complete program state at l including the stack, heap, and global variables. Given a patch applied to a function f in a vulnerable program p , we extract the concrete snapshot at the entry point of f by running p with a given PoC input. To perform this extraction, we use the symbolic execution tool KLEE [7]. Since KLEE maintains the program state during execution, extracting the snapshot incurs almost no



(a) A concrete snapshot example



(b) Abstract snapshot of the concrete snapshot shown in (a)

Figure 5: An example of abstract snapshot construction

additional overhead. Note that a concrete snapshot is extracted only once. Afterward, we perform patch verification by directly executing the patched function. We run the program on KLEE with the concrete PoC input and extract the program state at the entry point of f . If f is called multiple times, we only use the concrete snapshot from the final call to f that causes the crash.

3.2 Abstract Snapshot Construction

To explore the input space surrounding the extracted concrete snapshot s of function f , we construct an abstract snapshot \hat{s} from s . To achieve this, we first collect objects that are reachable from the following root nodes in s : (1) function f 's parameters, and (2) global variables.⁴ Take an object graph shown in Figure 5(a) as an example, where p and g denote a function parameter and a global variable, respectively. In the figure, each object is depicted as a stack of one or more rectangles, where each rectangle represents a field in the object. Each field is either a primitive value (e.g., 0, 1, 2, 50, and 'a') or a pointer to another object, including a function pointer fp .

Figure 5(b) shows the corresponding abstract snapshot, which is constructed as follows:

- Leaf objects, i.e., objects that do not point to other objects, are replaced with symbolic objects, denoted by cloud ; see s_1 and s_2 in the figure. Alias relations are preserved, as shown with s_1 .
- Primitive values contained in non-leaf objects are replaced with symbolic variables; see α_1 and α_2 .
- When searching for leaf objects, we ignore cyclic edges; see s_3 . This allows us to add symbolic objects that would otherwise be omitted.

⁴To limit the number of global variables considered, our current implementation only include those that are read while executing the program with the PoC. This implementation choice is orthogonal to the abstract snapshot construction algorithm.

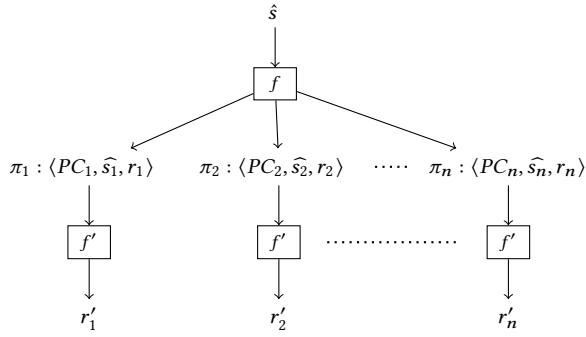


Figure 6: How SYMRADAR performs patch verification. We compare the execution result r_i of the original function f with the execution result r'_i of the patched function f' for each symbolic execution path π_i .

- We do not symbolize the objects pointed to by function pointers, as symbolizing them only leads to unresolved function calls; see fp and function in the figure.

Finally, if a root node is a variable of a primitive type, we replace it with a symbolic variable.

3.3 Patch Verification

Once an abstract snapshot is constructed, we perform patch verification. Figure 6 illustrates the process. We begin by executing the original function f symbolically, using the abstract snapshot \hat{s} as the initial input. Similar to UC-KLEE, symbolic objects in \hat{s} are initialized lazily when they are accessed during symbolic execution.

Symbolically executing f with \hat{s} produces a set of symbolic execution paths, $\pi_1, \pi_2, \dots, \pi_n$. Each path π_i is associated with a triple, $\langle PC_i, \hat{s}_i, r_i \rangle$, where:

- PC_i : the path condition of π_i .
- \hat{s}_i : the extended input observed at the end of π_i , reflecting how the initial abstract snapshot \hat{s} is extended during the symbolic execution via lazy initialization. See Figure 4(c) for an example.
- r_i : the execution result in path π_i , which can be either a crash or normal termination.

We perform patch verification for each path π_i . Note that each pair (PC_i, \hat{s}_i) represents a distinct input. We symbolically execute the patched function f' using this pair: \hat{s}_i is used as the initial input, and the initial path condition is set to PC_i .⁵ After executing f' , we monitor the execution result r'_i , which can be either a crash or normal termination. Because the symbolic execution of the patched function may explore multiple paths satisfying PC_i , multiple results $\{r'_i\}$ may be obtained.

Finally, we compare the results r_i and r'_i for each path π_i . If the patched function produces multiple results $\{r'_i\}$, we compare r_i with each r'_i . We determine whether the patch is safe or unsafe for each path π_i based on the rubric shown in Table 1. We consider the patch to be correct if it is labeled as “Safe” for all investigated paths. We below describe each case:

Case	r_i	r'_i	Condition	Classification
1	Crash	Normal	-	Safe
2	Crash C	Crash	$C = C_{PoC}$	Unsafe
3	Crash C	Crash	$C \neq C_{PoC}$	Safe
4	Normal	Crash	$C = C_{PoC}$	Unsafe
5	Normal	Crash	$C \neq C_{PoC}$	Safe
6	Normal	Normal	-	Check regression

Table 1: Classification rubric for execution results of each symbolic execution path. C_{PoC} denotes the crash observed in the original program when it is executed with the PoC input.

- **Case 1:** If a crash occurs in the original function f but not in the patched function f' , it indicates that the patch is safe for this path.
- **Cases 2 and 3:** If a crash occurs in both f and f' , we check whether the crash is the same as C_{PoC} , i.e., the crash observed in the original program when it is executed with the PoC input. If so, we consider the patch unsafe for this path; otherwise, we consider it safe for this path. The rationale behind this heuristic is that a crash different from C_{PoC} may result from an infeasible input.
- **Cases 4 and 5:** This corresponds to the case where f terminates normally, but f' crashes. Employing a similar heuristic as in Case 2 and 3, we check whether the crash is the same as C_{PoC} . If so, we consider the patch unsafe for this path; otherwise, we consider it safe for this path. The rationale is similar to before: a crash different from C_{PoC} may result from an infeasible input.
- **Case 6:** If both f and f' terminate normally, we check whether the patch introduces a regression error. If so, we consider the patch unsafe for this path; otherwise, it is considered safe.

Our heuristic used for Cases 2 and 3 is similar to differential assertion checking (DAC) [29]. To reduce false alarms caused by infeasible inputs, DAC ignores the inputs for which P fails the assertion. Similarly, we ignore inputs for which f crashes—except for the target crash C_{PoC} , which we intend to fix with the provided patch. We extend this heuristic to Cases 4 and 5 where we ignore the inputs for which f' crashes with a different reason than C_{PoC} . Overall, we focus on checking whether the vulnerability exposed by the PoC is fixed by the patch. In § 6.2, we evaluate the effectiveness of this heuristic.

In Case 6, we check for a regression error. While the specific method used to check for regression errors is orthogonal to our approach, our current implementation supports the following two methods: (1) comparing the outputs of the original and patched functions, and (2) comparing the sequence of branches taken at the patched conditions between the original and patched functions. For the former method, the outputs we consider include the return value of the function and writes to heap and global variables, similar to [35, 57]. Meanwhile, the latter is applicable only to patches that modify the conditional expressions of the original function or add a guarded statement, such as `if (size < 0) return NULL;`.⁶ Many security patches fall into this category, as they often return an error code or NULL when

⁵When executing the original function f , we set the initial path condition to *True*.

⁶The original function is assumed to have `if (0) return NULL;`.


an invalid input is detected. We track the sequence of branches taken in both the original and patched conditions during symbolic execution. If the sequences differ, we consider this a regression error, similar to [17, 40].

4 Implementaion and Optimizations

4.1 Implementation

We implemented SYMRADAR on top of KLEE [7], a well-known symbolic execution tool. SYMRADAR supports all three steps, concrete snapshot extraction, abstract snapshot construction, and patch verification. As mentioned in § 3.1, KLEE internally maintains the program state. While executing the program on KLEE with the provided PoC input, SYMRADAR stores the program state at the entry point of the patched function f as a file. The current implementation takes a snapshot at every entry point of f for implementation simplicity, although it is possible to take a snapshot only at the last call to f before the crash occurs.

Once the concrete snapshot is stored to a file, SYMRADAR loads the file and construct the object graph of the concrete snapshot. It then traverses the object graph using breadth-first search and constructs an abstract snapshot by applying the rules described in § 3.2.

Finally, SYMRADAR performs patch verification by executing the original and patched functions symbolically. We implemented the lazy initialization algorithm [24] described in § 3.2 on top of KLEE. Supporting lazy initialization requires type information for symbolic objects. For example, in our linked list example shown in Figure 4, SYMRADAR needs to know that the symbolic object, , has the type struct node in order to create a new node of that type. Since type information is not available in the program state maintained by KLEE, we retrieve it by analyzing the LLVM bytecode generated from the source code of the vulnerable program. Type information can be found in the GetElementPtr instructions within the LLVM bytecode. Our tool is publicly available at <https://github.com/UNIST-LOFT/symradar>.

4.2 Optimizations

We implemented several optimizations to verify patches with inputs likely to be relevant to the problematic crash, while avoiding irrelevant ones.

4.2.1 PoC-Oriented Concretization. Consider the following function under verification: `void foo(int fd, ...) { ... read(fd, ...); ...}` where `read` is a system call. Since `fd` has a primitive type, SYMRADAR replaces it with a symbolic variable during the abstract snapshot construction. When encountered with a system call such as `read`, KLEE—the underlying symbolic execution engine of SYMRADAR—concretizes the symbolic variable `fd` to a concrete value. However, since the concrete value of `fd` lost when the abstract snapshot is constructed, relying on the concretization of KLEE cannot access the file intended to be read, and as a result, fails to reproduce the crash when the original vulnerable function is executed with the abstract snapshot. To address this issue, we bookmark the concrete value of `fd` observed in the concrete snapshot, and use this value when concretization occurs for `fd`.

4.2.2 Order-Preserving Lazy Initialization. Consider a function accessing a buffer. Suppose that variables `start` and `end` point to the start and end of the buffer, respectively, when the function is executed with the provided PoC input. Now, consider lazily initializing these variables during symbolic execution. The original UC-SE allows `start` to point to a memory address that is larger than the address of `end`, which results in infeasible inputs. To avoid this, SYMRADAR preserves the relationship between `start` and `end` when lazily initializing them.

4.2.3 Static Analysis for Function Pointers. While performing lazy initialization, a new function pointer that does not exist in the concrete snapshot may be created. The original lazy initialization cannot handle this situation, as it does not know which function should be assigned to the function pointer. To address this, we modify the lazy initialization algorithm in two ways. First, when constructing an abstract snapshot, we do not symbolize function pointers and instead preserve the concrete functions they point to, as explained in § 3.2. Second, during symbolic execution, if a function pointer is encountered that was not present in the concrete snapshot, we initialize it to its type-compatible function identified through static analysis—using standard address-taken analysis and type-compatibility checks.

5 Evaluation Setup

We evaluate the verification performance of SYMRADAR through experiments. Specifically, we ask the following research questions:

- **RQ1:** How effective is SYMRADAR for patch verification? Specifically, how well does SYMRADAR filter out incorrect patches while preserving the correct ones?
- **RQ2:** How is the effectiveness of SYMRADAR compared to other state-of-the-art patch verification methods?
- **RQ3:** How is the performance of SYMRADAR affected by the verification bound and the heuristics used?
- **RQ4:** What is the runtime performance of SYMRADAR?
- **RQ5:** How does SYMRADAR perform on the patches that are not included in the benchmark used in the previous RQs?

5.1 Benchmarks

The main focus of this study is on patch verification for vulnerability patches. SYMRADAR can be applied to any vulnerability patch modifying a single function, which is the target of most AVR tools [18, 28, 48, 64]. As illustrated in Figure 2, AVR tools internally maintain patch spaces from which a patch that passes verification is produced as output. The larger the patch space, the more likely it is to include a correct patch. However, a larger patch space also contains more incorrect patches, thereby making the verification task more challenging.

For our evaluation, we use the patch space of CPR [48]. Among the existing AVR tools, CPR’s patch space is largest, providing a challenging benchmark for patch verification. Furthermore, CPR performs PoC-centered patch verification, making it comparable to our function-level PoC-centered verification technique. All 28 vulnerable programs in the CPR benchmark are written in C, and each has an associated PoC available in the benchmark.

Table 2: Evaluation results of SYMADAR and the compared techniques

Program	Bug ID	Patches		CPR [†]		SPIDER		VULNFix [†]		UC-KLEE		SYMADAR	
		C	I	FN	FP	FN	FP	FN	FP	FN	FP	FN	FP
coreutils	gnubug-25003	1	138	0	119	0	69	0	131	0	122	0	18
coreutils	gnubug-25023	1	43	0	43	0	43	0	27	0	3	0	0
coreutils	bugzilla-19784	1	3	0	3	0	0	0	1	0	3	0	3
coreutils	bugzilla-26545	1	966	0	890	1	0	n.a.	n.a.	0	4	0	4
jasper	CVE-2016-8691	1	90	0	66	0	90	1	0	0	45	0	31
jasper	CVE-2016-9387	0	45	0	11	0	45	n.a.	n.a.	0	40	0	0
binutils	CVE-2017-15025	1	89	0	89	0	89	0	2	0	89	0	89
binutils	CVE-2018-10372	1	1	0	1	0	1	0	1	0	1	0	1
libjpeg	CVE-2012-2806	1	3	0	3	0	3	0	3	0	3	0	3
libjpeg	CVE-2017-15232	1	510	0	510	1	0	0	248	0	248	0	75
libjpeg	CVE-2018-14498	1	89	0	87	1	0	1	0	0	89	0	89
libjpeg	CVE-2018-19664	1	89	0	89	0	89	1	0	0	89	0	2
libtiff	CVE-2014-8128	1	1	0	1	0	1	n.a.	n.a.	0	1	0	1
libtiff	CVE-2016-10094	1	89	0	52	0	55	1	3	0	89	0	87
libtiff	CVE-2016-3186	1	89	0	89	0	89	n.a.	n.a.	0	3	0	14
libtiff	CVE-2016-3623	1	90	0	69	1	0	0	90	0	45	0	21
libtiff	CVE-2016-5314	1	138	0	136	0	138	n.a.	n.a.	0	138	0	4
libtiff	CVE-2016-5321	1	1	0	1	0	1	0	1	0	1	0	1
libtiff	CVE-2016-9273	1	9	0	8	1	0	1	0	0	9	0	2
libtiff	CVE-2017-7595	1	90	0	90	0	90	1	0	0	90	0	90
libtiff	CVE-2017-7601	1	63	0	63	0	63	1	0	0	63	0	21
libtiff	bugzilla-2611	1	89	0	78	0	89	0	80	0	89	0	29
libtiff	bugzilla-2633	1	89	0	89	0	89	0	1	0	15	0	45
libxml2	CVE-2012-5134	1	1	1	0	0	1	0	1	0	1	0	1
libxml2	CVE-2016-1838	1	138	0	138	0	138	1	0	1	12	0	13
libxml2	CVE-2016-1839	1	45	0	45	0	45	0	1	1	4	0	1
libxml2	CVE-2017-5969	1	13	0	13	1	0	0	13	0	13	0	13
Total	-	26	3011	1	2783	6	1228	8	603	2	1309	0	658
Recall				96%		77%		62%		92%		100%	
Specificity				8%		59%		66%		57%		78%	
Balanced Accuracy				52%		67%		64%		74%		89%	

C (Correct): Number of correct patches (i.e., positives). **I (Incorrect):** Number of incorrect patches (i.e., negatives).

FN (False Negative): Number of correct patches that are incorrectly classified as incorrect.

FP (False Positive): Number of incorrect patches that are incorrectly classified as correct.

[†]: For CPR and VULNFix, we evaluate the performance of their patch verification modules.

n.a.: VULNFix is failed to be run due to its implementation issues.

We evaluate SYMADAR using all 28 programs in the CPR benchmark.⁷ For these programs, CPR generates a total of 3,037 unique plausible patches,⁸ all of which avoid the crash when tested with the given PoC inputs. We apply SYMADAR to these 3,037 patches to check their correctness. To answer RQ5, we collect separate patches for 9 additional vulnerable programs outside the CPR benchmark, as described in § 6.4.

⁷CPR is not applicable to two FFmpeg programs, which are excluded from our evaluation.

⁸We removed semantically equivalent duplicate patches.

5.2 Compared Patch Verification Techniques

We compare SYMADAR with the following patch verification techniques: (1) CPR [48], (2) VULNFix [64], (3) SPIDER [35], and (4) UC-KLEE [43]. CPR is an AVR tool that performs PoC-centered patch verification at the system level using concolic execution. VULNFix performs patch verification using both system-level fuzzing and function-level fuzzing, with the latter performed by mutating the program state at the entry point of the patched function. SPIDER is a state-of-the-art static-analysis-based patch verification tool. UC-KLEE is our baseline approach for bounded patch verification.

Since UC-KLEE implementation is not publicly available, we implement a UC-KLEE mode within SYMRADAR. The main difference between SYMRADAR and UC-KLEE is the initial state of symbolic execution: while SYMRADAR uses the abstract snapshot of the PoC input, UC-KLEE uses an uninitialized input state.

5.3 Evaluation Metrics

Our dataset is skewed towards incorrect patches, reflecting the sparsity of correct patches and the abundance of incorrect patches in the patch space. The goal of patch verification is to filter out as many incorrect patches as possible while preserving the correct ones. Accordingly, we report the following metrics:

- **Recall:** The ratio of correctly identified correct patches (true positives) to the total number of correct patches.
- **Specificity:** The ratio of correctly identified incorrect patches (true negatives) to the total number of incorrect patches.
- **Balanced Accuracy:** The average of recall and specificity. This metric is typically used when the dataset is imbalanced, as in our case.

In our dataset, each subject has a different number of correct and incorrect patches. Reporting true positives and true negatives for each subject is not informative. Instead, we report false negatives and false positives, which are the number of correct patches incorrectly classified as incorrect and the number of incorrect patches incorrectly classified as correct, respectively.

5.4 Experimental Environment

We set a 12-hour timeout for each patch verification task across all tools, considering the usage scenario of an overnight run. For SYMRADAR and UC-KLEE, which use lazy initialization with k -bounding, we use a k value of 3 by default, but also report results for $k = 6$ and $k = 9$. All experiments are performed on a machine equipped with an AMD EPYC processor running at 2.45 GHz and 1024 GB of RAM.

6 Evaluation Results

Table 2 shows the evaluation results of SYMRADAR and the four compared techniques. The left two columns present the information about the subjects, including the names of the programs and their bug IDs. The next two columns, C and I, show the number of correct patches and the number of incorrect patches. Recall that all patches avoid the crash when tested with the given PoC input. The remaining columns show the results of patch verification with SYMRADAR and other techniques. Based on the results, we answer the three research questions below.

6.1 RQ1 and RQ2: Verification Effectiveness

Before describing the results, we briefly review the evaluation context. Our evaluation simulates the internal process of an AVR tool (see Figure 2). An AVR tool first identifies a set of plausible patches that prevents the crash when tested with the provided PoC. In the next step, it filters out incorrect patches among the collected plausible patches. SYMRADAR removes incorrect patches only when it finds a concrete evidence of their incorrectness: an input that leads to unsafe execution of the patched program (see Table 1). Ideally,

all incorrect patches should be filtered out, while preserving the correct one. Considering the sparsity of correct patches, it is crucial to avoid discarding them.

As indicated by the recall value, SYMRADAR preserves all the correct patches in our benchmark. It is the only tool that achieves 100% recall, meaning that all the correct patches are correctly classified as correct. While CPR also shows a high recall (96%), it exhibits the lowest specificity (8%) among all the tools, indicating that it fails to filter out many incorrect patches. This is due to the fact that CPR performs concolic execution at the system level, and as a result, making it difficult even to reach the patched code.

Now, we compare SYMRADAR with the remaining tools one by one. UC-KLEE is closest to SYMRADAR in the sense that both tools perform function-level symbolic execution using lazy initialization. The main difference is in that only SYMRADAR performs PoC-centered symbolic execution. UC-KLEE also achieves a decent recall (92%), although it is lower than SYMRADAR. Its weakness is in specificity, which is only 57%, meaning that it fails to filter out many incorrect patches. In comparison, SYMRADAR achieves a much higher specificity (78%), demonstrating the efficacy of PoC-centered symbolic execution.

VULNFix also exhibits lower recall and specificity than SYMRADAR. The performance of VULNFix varies significantly across subjects. For example, in bugzilla-19784, VULNFix filters out all incorrect patches except one, whereas in gnuvuln-25003, it filters out only 7 out of 138 incorrect patches. In comparison, SYMRADAR performs more consistently across subjects, highlighting the advantages of our systematic PoC-centered input space exploration over VULNFix’s mutation-based approach. Another issue with VULNFix is in its lowest recall (62%) among all the tools, indicating that it often fails to preserve the correct patch. Since VULNFix randomly mutates the program state at the entry point of the patched function, it can easily generate infeasible inputs that lead to rejecting the correct patch. In comparison, SYMRADAR systematically explores the input space around the PoC input, which is more likely to lead to feasible inputs that preserve the correct patch.

Lastly, SPIDER also shows a low recall (77%) and specificity (59%) than SYMRADAR. As typical in static analysis, SPIDER performs well on some subjects (e.g., CVE-2016-5321) but performs poorly on others (e.g., CVE-2016-1838). Its performance varies greatly across subjects, depending on how well the static analysis rules align with the patch under verification.

RQ1 and RQ2: SYMRADAR achieves the best performance in terms of recall (100%) and specificity (78%) among all the tools. Its balanced accuracy is 89%, which is also the highest among all the tools, demonstrating the well-balanced performance of SYMRADAR.

6.2 RQ3: Ablation Study

In our main evaluation, we set the bound of lazy initialization to 3, as mentioned in § 5.4. With RQ3, we investigate the impact of the size of the bound. We also investigate the impact of the heuristic we used to determine whether a patch is correct or not (see § 3.3). We conduct an ablation study by varying the bound and the heuristic.

Table 3 shows the results of the ablation study. The second column shows the bound of lazy initialization, which is set to 3, 6, or 9.

Table 3: Ablation Study

Tool	Bound	Heuristic	FN	FP	Recall	Specificity	B.Acc
SYMADAR	3	Y	0	658	100%	78%	89%
		N	7	617	73%	80%	76%
	6	Y	1	691	96%	77%	87%
		N	6	665	77%	78%	77%
	9	Y	1	697	96%	77%	87%
		N	7	668	73%	78%	75%
UC-KLEE	3	Y	2	1309	92%	57%	74%
		N	6	1270	77%	58%	67%
	6	Y	2	1327	92%	56%	74%
		N	6	1064	77%	65%	71%
	9	Y	2	1326	92%	56%	72%
		N	6	1065	77%	65%	71%

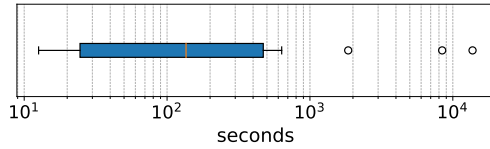


Figure 7: Time distribution of SYMADAR for detecting incorrect patches.

The third column shows whether the heuristic is used or not. The remaining columns show the results of patch verification, including the number of false negatives (FN), the number of false positives (FP), recall, specificity, and balanced accuracy (B.Acc).

The results show that the size of the bound does not significantly affect the performance of SYMADAR. For example, the recall for each bound is 100%, 96%, and 96%, respectively. Similarly, the specificity for each bound is 78%, 77%, and 77%, respectively. The performance of UC-KLEE also does not significantly change with the size of the bound.

Meanwhile, the heuristic has a significant impact on the performance of SYMADAR. When the heuristic is not used, the recall drops to 73% from 100% (for bound 3), while the specificity marginally increases from 78% to 80%. Overall, the balanced accuracy drops from 89% to 76%. Similar trends are observed for the other bounds as well.

Lastly, we evaluate the impact of static analysis for function pointers (see § 4.2.3). When this static analysis is disabled, specificity drops from 78% to 74%, while recall remains unchanged. This indicates that our static analysis helps in correctly identifying incorrect patches.

RQ3: SYMADAR’s performance is not significantly affected by the size of the bound of lazy initialization. However, the heuristic used to determine whether a patch is correct or not has a significant impact on the performance of SYMADAR.

6.3 RQ4: Runtime Performance

Figure 7 shows the time distributions for detecting incorrect patches. SYMADAR detects incorrect patches in between 12 seconds and 3.8 hours, with a median detection time of 2.2 minutes and a third

quartile time of 7.9 minutes. This implies that the evaluation results in Table 2 can be obtained within a time budget of 3.8 hours, and that 75% of the incorrect patches in the benchmark can be detected in less than 8 minutes.

RQ4: SYMADAR detects 90% of the incorrect patches in 3 minutes and 95% within 2.3 hours, indicating that a much shorter timeout than 12 hours is sufficient in practice.

6.4 RQ5: Generalizability

Benchmark. To evaluate the generalizability of SYMADAR, we require a new benchmark that is distinct from the CPR benchmark used in our main evaluation. To this end, we constructed a new set of patches by applying an AVR (Automated Vulnerability Repair) tool—other than CPR—to a set of vulnerable programs not included in the CPR benchmark. Specifically, we applied SAN2PATCH [27], a recent AVR tool leveraging LLMs (Large Language Models), to the VulnLoc benchmark [2]. Since the VulnLoc benchmark overlaps with the CPR benchmark, we applied SAN2PATCH to the 18 vulnerable programs not included in the CPR benchmark.

To obtain multiple plausible (i.e., PoC-avoiding) patches, we allow SAN2PATCH to invoke the LLM 50 times per vulnerable program. As SYMADAR is designed to verify a single patched function, we specified in the LLM prompt of SAN2PATCH to modify only one function. Under this setup, SAN2PATCH successfully generated plausible patches for 9 out of the 18 vulnerable programs.

The first three columns of Table 4 show the information about the vulnerable programs and the collected plausible patches. In total, we collected 90 unique plausible patches—21 correct and 69 incorrect ones. We determined the correctness of each patch by manually investigating whether it is semantically equivalent to the developer patch.

Results. Table 4 presents the evaluation results of SYMADAR and the compared techniques on the new benchmark. The overall trends of the results are similar to those observed with the CPR benchmark. SYMADAR achieves the highest recall (100%), the second-highest specificity (74%) after SPIDER (83%), and the highest balanced accuracy (87%). Although SPIDER shows the highest specificity, its recall is lowest (48%), indicating that it classifies the majority of correct patches as incorrect. Unlike SPIDER, whose performance varies significantly across benchmarks, SYMADAR consistently achieves well-balanced, high performance on both the CPR benchmark and the new benchmark, demonstrating its generalizability.

RQ5: SYMADAR consistently achieves well-balanced, high performance on both benchmarks (100% recall in both benchmarks; 78% and 74% specificity in the CPR and new benchmarks, respectively; 89% and 87% balanced accuracy in the CPR and new benchmarks, respectively), demonstrating its generalizability.

7 Discussion

Why SYMADAR outperforms UC-KLEE. SYMADAR outperforms UC-KLEE in both recall and specificity. In particular, the gap in specificity is significant, with SYMADAR achieving 78% while

Table 4: Evaluation results of SYMRADAR and the compared techniques for the additional benchmark.

Program	Bug ID	Patches		SPIDER		VULNFix [†]		UC-KLEE		SYMRADAR	
		C	I	FN	FP	FN	FP	FN	FP	FN	FP
binutils	CVE-2017-6965	1	3	0	3	1	2	0	3	0	0
jasper	CVE-2016-9557	6	9	6	0	0	13	0	9	0	5
libming	CVE-2016-9264	3	3	3	0	0	1	0	1	0	3
libtiff	CVE-2016-9532	1	30	0	1	0	30	0	6	0	0
libtiff	CVE-2017-5225	3	13	0	3	1	1	0	10	0	10
libtiff	CVE-2017-7599	2	0	0	2	2	0	0	0	0	0
libtiff	CVE-2017-7600	2	0	0	2	2	0	0	0	0	0
zziplib	CVE-2017-5975	2	4	2	0	n.a.	n.a.	0	4	0	0
zziplib	CVE-2017-5976	1	7	0	1	1	0	0	0	0	0
Total	-	21	69	11	12	7	47	0	33	0	18
Recall				48%		63%		100%		100%	
Specificity				83%		28%		52%		74%	
Balanced Accuracy				65%		45%		76%		87%	

C (Correct): Number of correct patches (i.e., positives). **I (Incorrect):** Number of incorrect patches (i.e., negatives).

FN (False Negative): Number of correct patches that are incorrectly classified as incorrect.

FP (False Positive): Number of incorrect patches that are incorrectly classified as correct.

[†]: For VULNFix, we evaluate the performance of their patch verification modules.

n.a: VULNFix is failed to be run due to its implementation issues.

UC-KLEE only achieves 57%. This is because SYMRADAR performs PoC-centered symbolic execution, which allows it to explore the input space around the PoC input. In contrast, UC-KLEE performs lazy initialization starting from the empty input state, which is likely to generate many inputs irrelevant to the patch under verification.

To investigate this difference between SYMRADAR and UC-KLEE more concretely, we count the number of symbolic execution paths that reach the patch location. We found that in only 28.6% of the paths explored by UC-KLEE, the patch location is reached. In contrast, SYMRADAR reaches the patch location in 66.5% of the paths explored. This indicates that UC-KLEE spends much of its time exploring irrelevant paths, resulting in lower specificity. In contrast, by focusing on the input space around the PoC input, SYMRADAR explores more relevant paths.

Cost of Concrete Snapshot Extraction. SYMRADAR extracts concrete snapshots by executing the original program with the PoC input. This incurs an initial cost, although this is a one-time cost. In our experiments, it took on average 15 minutes to extract the snapshot for each subject. However, the fact that the extracted snapshot can be shared across multiple patches modifying the same function amortizes this cost.

Limitations of SYMRADAR. When abstracting the snapshot, the current implementation of SYMRADAR does not symbolize global variables that are not read while executing the original program with the PoC input, as explained in § 3.2. While this is to limit the number of global variables to be symbolized, as a result, SYMRADAR cannot treat such global variables symbolically when they are used in a patched function.

SYMRADAR is built on top of KLEE [7], inheriting its limitations. For instance, a crash reported by a sanitizer such as AddressSanitizer [3] may not be reproducible in KLEE when the same PoC input is used. This is due to KLEE’s limited capability for error detection (e.g., KLEE invokes simplified models for system calls, which may miss certain errors), which is an orthogonal issue to our approach.

8 Threats to Validity

We acknowledge the following threats to the validity of our study. First, caution should be exercised when generalizing our results to other AVR tools and benchmarks. Second, in our experiments, we use specific k -bounding values (3, 6, and 9) and heuristics, and different configurations might yield different results. However, according to our ablation study (see § 6.2), neither the size of the bound nor the heuristic significantly affects the performance of SYMRADAR.

9 Related Work

9.1 Automated Vulnerability Repair

Many AVR tools have been proposed to automatically generate patches for vulnerable programs. SENX [22] uses a patch template suitable for the identified vulnerability and generates a patch that passes the given crash-inducing input. EXTRACTFix [18] similarly exploits the identified vulnerability type to generate a patch. Furthermore, unlike SENX, it also conducts bounded verification for the generated patch. CPR [48] similarly generates patches and conducts patch verification. However, as shown in this paper, their patch verification is not sufficiently effective. VULNFix [64] conducts fuzzing at the patch location to work around the limitations of system-level

fuzzing, as discussed in § 7. Another AVR tool, CRASHREPAIR [49], similarly performs snapshot-based fuzzing for patch verification. However, unlike SYM-RADAR that performs bounded verification around the crash-inducing input, these fuzzing approaches provides little evidence on the patch correctness.

There are also AVR tools based on static analysis. These tools generate patches that eliminate warnings produced by static analyzers. While they typically fix specific types of vulnerabilities that can be detected by static analyzers, such as memory leaks and use-after-frees (examples include FOOTPATCH [55], MEMFIX [31], SAVER [20], and EFFFIX [63]), there is also an AVR tool that supports user-provided custom specifications like PROVENFIX [51]. Although it is often claimed that the patches generated by these tools are guaranteed to be correct, such claims should be taken with caution. For instance, FOOTPATCH uses a static analyzer, INFER [4], to detect and fix memory leaks, but the patches it generates can introduce use-after-free errors. It is also noteworthy that industry-grade static analyzers, such as INFER [4] (used in FOOTPATCH) and PULSE [5] (used in EFFFIX), are neither sound nor complete [1]. As a result, AVR tools built on top of such static analyzers may miss vulnerabilities that can be manifested by a PoC input or produce incorrect patches.

Recently, deep-learning-based techniques have also been proposed for AVR, such as VREPAIR [10], VULREPAIR [16], VULMASTER [65], and SAN2PATCH [27]. These tools focus on patch generation using neural models, but lack dedicated patch verification mechanisms beyond testing with provided PoC inputs and regression tests.

9.2 Patch Verification and Testing

Symbolic Execution. Independent of AVR, symbolic execution has been widely used for patch verification and testing. KATCH [36] finds a system-level input that reaches the patched program location by using symbolic execution. Palikareva et al. [40] proposed a symbolic-execution-based approach that finds an input that takes different branches between the original and patched programs. ODIT [45] performs differential testing between the original and patched programs using UC-SE. VERIBIN [57] performs UC-SE at the binary level. Unlike SYM-RADAR, these symbolic-execution-based approaches either use system-level symbolic execution (KATCH and [40]), which causes scalability issues, or UC-SE without considering the crash-inducing input (ODIT and VERIBIN), limiting their effectiveness in patch verification.

Deep Learning. Recently, deep-learning techniques [8, 9, 11, 15, 33, 52] have been utilized to detect vulnerable functions. These techniques typically involve training a neural model with a large dataset of functions labeled as vulnerable or safe. A recent trend in this area is to enhance classification accuracy by training models with diverse semantic information about programs. For example, PDBERT [33] incorporates control and data dependencies of programs, while DEEPDFA [52] simulates data-flow analysis on neural graph networks. However, a recent empirical study [44] has pointed out a key limitation of these deep-learning models for vulnerability detection: they classify functions as vulnerable or safe in isolation without considering their calling contexts. This is problematic since the same function can be vulnerable in one calling context but safe in another.

Static Analysis. The calling-context issue also persists in static analyzers. To consider the calling context, static analyzers typically perform context-sensitive analysis. For example, k -call-site-sensitive analysis [41] distinguishes a function’s calling contexts based on its last k call sites on the call stack. However, to ensure scalability, a small k (typically $k \leq 2$) is often used, even though in practice, the calling context of a vulnerable function can be much deeper. As a result, static analyzers typically approximate the calling context of a function, which can lead to imprecise analysis results. In contrast, SYM-RADAR’s PoC-centered snapshots can capture the problematic calling context of a vulnerable function, regardless of call depth.

Function-Level Concolic Execution. Tools such as CUTE [46] and PEX [54] perform concolic execution directly on the target function, similar to under-constrained symbolic execution. In theory, these approaches can perform similarly to SYM-RADAR, if a user provides an adequate test driver that constructs the calling context corresponding to the PoC-centered abstract snapshot. However, in practice, writing or extracting such a test driver is non-trivial. SYM-RADAR, by contrast, eliminates this burden by automatically constructing PoC-centered abstract snapshots.

9.3 Automated Program Repair

AVR is a special case of APR (Automated Program Repair) [18, 25, 26, 47, 58, 61]. Most APR tools are test-driven, meaning that they generate patches that pass a given test suite. Various approaches have been proposed to generate patches, including genetic programming [30], template-based synthesis [32], constraint solving [37, 38, 62], and learning-based generation [66]. APR tools may generate incorrect patches that merely pass the given test suite, known as overfitting patches [50] or plausible patches [42]. To address this issue, various patch classification techniques—which classify patches as correct or incorrect—have been proposed, including machine learning-based classification [60], execution-profile-based classification [59], and fuzzing-based test augmentation [23, 53]. Although SYM-RADAR is designed for AVR, it can be generalized to verify functions patched by APR tools if equipped with a suitable postcondition for the function under verification.

10 Conclusion

In this paper, we have proposed a novel patch verification tool, SYM-RADAR. To achieve high effectiveness for patch verification, SYM-RADAR performs under-constrained symbolic execution (UC-SE) in the vicinity of the PoC input. We demonstrated the effectiveness and efficiency of SYM-RADAR through experimental evaluation on real-world vulnerabilities and a large set of patches. Overall, we have shown that it is possible to verify patches effectively while providing a high degree of confidence in their correctness.

Acknowledgements

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2023-00213434, RS-2021-NR060080) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00337414).

References

- [1] 2016. Unsoundness and incompleteness of Infer. <https://github.com/facebook/infer/issues/427>
- [2] 2024. VulnLoc Benchmark. <https://github.com/nus-apr/vulnloc-benchmark>
- [3] 2025. AddressSanitizer. <https://github.com/google/sanitizers/wiki/AddressSanitizer>
- [4] 2025. Infer. <https://fbinfer.com/>
- [5] 2025. Pulse. <https://fbinfer.com/docs/checker-pulse/>
- [6] Josiah Bates. 2019. 2017 Equifax data breach. <https://time.com/5634465/equifax-125-data-breach-how-to/> Accessed: 2025-04-15.
- [7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. 209–224.
- [8] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. 2021. BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection. *Information and Software Technology* 136 (Aug. 2021), 106576. doi:10.1016/j.infsof.2021.106576
- [9] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, Xiaolei Liu, Xingwei Lin, and Wei Liu. 2024. Snopy: Bridging Sample Denoising with Causal Graph Learning for Effective Vulnerability Detection. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Sacramento CA USA, 606–618. doi:10.1145/3691620.3695057
- [10] 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 (Jan. 2023), 147–165. doi:10.1109/TSE.2022.3147265
- [11] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Transactions on Software Engineering and Methodology* 30, 3 (July 2021), 1–33. doi:10.1145/3436877
- [12] Dominic Conlon. 2021. Class action in British Airways data breach. <https://irglobal.com/article/class-action-in-british-airways-data-breach/>
- [13] Xianghua Deng, Jooyong Lee, and Robby. 2006. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, Tokyo, 157–166. doi:10.1109/ASE.2006.26
- [14] Xianghua Deng, Jooyong Lee, and Robby. 2012. Efficient and formal generalized symbolic execution. *Automated Software Engineering* 19, 3 (Sept. 2012), 233–301. doi:10.1007/s10515-011-0089-9
- [15] 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)*. IEEE, Montreal, QC, Canada, 60–71. doi:10.1109/ICSE.2019.00024
- [16] Michael Fu, Chakkrit Tantithamthavorn, and Trung Le. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. (2022), 13.
- [17] 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*. ACM, Beijing China, 8–18. doi:10.1145/3293882.3330558
- [18] 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* 30, 2 (April 2021), 1–27. doi:10.1145/3418461
- [19] Andy Greenberg. 2018. The Untold Story of NotPetya, the Most Devastating Cyberattack in History. <https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/>
- [20] 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*. ACM, Seoul South Korea, 271–283. doi:10.1145/3377811.3380323
- [21] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. <http://arxiv.org/abs/2303.18184> arXiv:2303.18184 [cs].
- [22] 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, San Francisco, CA, USA, 539–554. doi:10.1109/SP.2019.00071
- [23] Elkan Ismayilzada, Md Mazba Ur Rahman, Dongsun Kim, and Jooyong Yi. 2024. Poracle: Testing Patches under Preservation Conditions to Combat the Overfitting Problem of Program Repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (Feb. 2024), 1–39. doi:10.1145/3625293
- [24] Saffraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Hubert Garavel, and John Hatcliff (Eds.). Vol. 2619. Springer Berlin Heidelberg, Berlin, Heidelberg, 553–568. doi:10.1007/3-540-36577-X_40 Series Title: Lecture Notes in Computer Science.
- [25] YoungJae Kim, Seunghoon Han, Askar Yeltayuly Khamit, and Jooyong Yi. 2023. Automated Program Repair from Fuzzing Perspective. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle WA USA, 854–866. doi:10.1145/3597926.3598101
- [26] YoungJae Kim, Yechan Park, Seunghoon Han, and Jooyong Yi. 2024. Enhancing the Efficiency of Automated Program Repair via Greybox Analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Sacramento CA USA, 1719–1731. doi:10.1145/3691620.3695602
- [27] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and Jiwon Yoon. 2025. Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis. In *34th USENIX Security Symposium*. 4401–4419.
- [28] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*. ACM, Porto de Galinhas Brazil, 103–111. doi:10.1145/3664646.3664770
- [29] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, Saint Petersburg Russia, 345–355. doi:10.1145/2491411.2491452
- [30] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. doi:10.1109/TSE.2011.104
- [31] 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*. ACM, Lake Buena Vista FL USA, 95–106. doi:10.1145/3236024.3236079
- [32] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 31–42. doi:10.1145/3293882.3330577
- [33] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. 2024. Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. doi:10.1145/3597503.3639142
- [34] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, Austin Texas, 702–713. doi:10.1145/2884781.2884872
- [35] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: Enabling Fast Patch Propagation In Related Software Repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1562–1579. doi:10.1109/SP40000.2020.00038
- [36] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, Saint Petersburg Russia, 235–245. doi:10.1145/2491411.2491438
- [37] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 448–458. doi:10.1109/ICSE.2015.63
- [38] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, Austin Texas, 691–701. doi:10.1145/2884781.2884807
- [39] Martin Monperrus. 2019. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan. 2019), 1–24. doi:10.1145/3105906
- [40] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, Austin Texas, 1181–1192. doi:10.1145/2884781.2884845
- [41] M Pnueli and Micha Sharir. 1981. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications* (1981), 189–234.
- [42] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, Baltimore MD USA, 24–36. doi:10.1145/2771783.2771791
- [43] David A Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium*. 49–64.
- [44] Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 388–410.
- [45] Richard Rutledge and Alessandro Orso. 2022. Automating Differential Testing with Overapproximate Symbolic Execution. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Valencia, Spain, 256–266. doi:10.1109/ICST53961.2022.00035
- [46] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. 263–272.

- [47] Arooba Shahoor, Askar Yeltayuly Khamit, Jooyong Yi, and Dongsun Kim. 2023. Leakpair: Proactive Repairing of Memory Leaks in Single Page Web Applications. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Luxembourg, Luxembourg, 1175–1187. doi:10.1109/ASE56229.2023.00097
- [48] 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*. ACM, Virtual Canada, 390–405. doi:10.1145/3453483.3454051
- [49] Ridwan Shariffdeen, Christopher S. Timperley, Yannic Noller, Claire Le Goues, and Abhik Roychoudhury. 2025. Vulnerability Repair via Concolic Execution and Code Mutations. *ACM Transactions on Software Engineering and Methodology* 34, 4 (May 2025), 1–27. doi:10.1145/3707454
- [50] 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*. ACM, Bergamo Italy, 532–543. doi:10.1145/2786805.2786825
- [51] Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. 2024. ProveNFix: Temporal Property-Guided Program Repair. 1, FSE (2024), 226–248.
- [52] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. doi:10.1145/3597503.3623345
- [53] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 981–992. doi:10.1145/3324884.3416532
- [54] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–White Box Test Generation for .NET. In *Tests and Proofs*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Bernhard Beckert, and Reiner Hähnle (Eds.). Vol. 4966. Springer Berlin Heidelberg, Berlin, Heidelberg, 134–153. doi:10.1007/978-3-540-79124-9_10 Series Title: Lecture Notes in Computer Science.
- [55] 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*. ACM, Gothenburg Sweden, 151–162. doi:10.1145/3180155.3180250
- [56] Katharine Viner. 2020. Prosecutors open homicide case after cyber-attack on German hospital. <https://www.theguardian.com/technology/2020/sep/18/prosecutors-open-homicide-case-after-cyber-attack-on-german-hospital>
- [57] Hongwei Wu, Jianliang Wu, Ruoyu Wu, Ayushi Sharma, Aravind Machiry, and Antonio Bianchi. 2025. VeriBin: Adaptive Verification of Patches at the Binary Level. In *Proceedings 2025 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. doi:10.14722/ndss.2025.230359
- [58] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair Via Zero-Shot Learning. <http://arxiv.org/abs/2207.08281> arXiv:2207.08281 [cs].
- [59] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 789–799. doi:10.1145/3180155.3180182
- [60] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2022. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Transactions on Software Engineering* 48, 8 (Aug. 2022), 2920–2938. doi:10.1109/TSE.2021.3071750
- [61] Jooyong Yi. 2020. On the Time Performance of Automated Fixes. In *Proceedings of 6th International Conference in Software Engineering for Defence Applications*, Paolo Ciancarini, Manuel Mazzara, Angelo Messina, Alberto Sillitti, and Giancarlo Succi (Eds.). Vol. 925. Springer International Publishing, Cham, 312–324. doi:10.1007/978-3-030-14687-0_28 Series Title: Advances in Intelligent Systems and Computing.
- [62] Jooyong Yi and Elkhan Ismayilzade. 2022. Speeding up constraint-based program repair using a search-based technique. *Information and Software Technology* 146 (June 2022), 106865. doi:10.1016/j.infsof.2022.106865
- [63] Yuntong Zhang, Andreea Costea, Ridwan Shariffdeen, Davin McCall, and Abhik Roychoudhury. 2025. EffFix: Efficient and Effective Repair of Pointer Manipulating Programs. *ACM Transactions on Software Engineering and Methodology* 34, 3 (March 2025), 1–27. doi:10.1145/3705310
- [64] 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*. ACM, Virtual South Korea, 691–702. doi:10.1145/3533767.3534387
- [65] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. doi:10.1145/3597503.3639222
- [66] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens Greece, 341–353. doi:10.1145/3468264.3468544