Patch Validation in Automated Vulnerability Repair

Zheng Yu[†§], Wenxuan Shi[§], Xinqian Sun[†§], Zheyun Feng[‡], Meng Xu[†], Xinyu Xing[§]
[§]*Northwestern University* [†]*University of Waterloo*, [‡]*University of New Hampshire*
{zheng.yu,wenxuan.shi,xinqian.sun,xinyu.xing}@northwestern.edu,
meng.xu.cs@uwaterloo.ca, Zheyun.Feng@unh.edu

ABSTRACT

Automated Vulnerability Repair (AVR) systems, especially those leveraging large language models (LLMs), have demonstrated promising results in patching vulnerabilities—that is, if we trust their patch validation methodology. Ground-truth patches from human developers often come with new tests that not only ensure mitigation of the vulnerability but also encode extra semantics such as root cause location, optimal fix strategy, or subtle coding styles or conventions. And yet, none of the recent AVR systems verify that the auto-generated patches additionally pass these new tests (termed as PoC$^+$ tests). This is a subtle yet critical omission.

To fill this gap, we constructed a benchmark, PVBench, with 209 cases spanning 20 projects. Each case includes basic tests (functional tests before the patch and the PoC exploit) as well as the associated PoC$^+$ tests. Evaluated on three state-of-the-art AVR systems, we find that over 40% of patches validated as correct by basic tests fail under PoC$^+$ testing, revealing substantial overestimation on patch success rates. Analyzing these patches that are falsely labeled as correct, we suggest that AVR tools should improve in three critical areas: root cause analysis, adherence to program specifications, and capturing developer intention.

1 INTRODUCTION

Patch validation—ensuring that a patch effectively addresses security vulnerabilities while preserving functional integrity—is crucial in software development. While traditionally applied to human-written patches [25, 71], patch validation has become increasingly important for automated vulnerability repair (AVR) systems [17, 28, 45], where it serves to evaluate auto-generated patches. Hence, the reliability of AVR effectiveness evaluation therefore depends on the accuracy and comprehensiveness of their underlying patch validation methodologies.

In an abstract view, an AVR tool takes, at minimum, a vulnerable codebase C and a proof-of-concept (PoC) input *poc* that demonstrates the existence of a vulnerability in C, and produces a patch $\hat{p}$ that is supposed to fix the vulnerability. Many AVR tools [10, 15, 73] that achieved good performance in the pre-LLM era focused exclusively on vulnerabilities exhibiting fixed patterns, limiting their generalizability to other vulnerability classes. As code generation and completion capabilities advance through machine learning techniques, particularly large language models (LLMs), researchers are increasingly exploring their application in AVR [17, 28, 45]. Several recent studies [23, 40, 80] have demonstrated promising repair rates far exceeding best AVR tools before the LLM era.

Regarding validating the generated patch $\hat{p}$, our survey of the recent literature reveals three primary methodologies adopted in previous AVR works, including *manual comparison* that relies on expert assessment [24, 40, 73], *similarity metrics* that employ similarity-based scoring [5, 8, 86], and *test suite validation* that executes automated tests to validate patch correctness [23, 45, 80, 84]. Both manual comparison and similarity metrics compare against a ground-truth patch p which is typically developer-written and committed into the codebase as the official bugfix. While manual comparison can provide high accuracy in principle, it lacks scalability for large-scale evaluation [26, 79]. Similarity-based metrics offer automation but have been shown to inadequately correlate with actual patch effectiveness [84], as patches achieving high similarity scores may still fail to address the vulnerability.

Test suite-based validation, on the other hand, does not necessarily require a ground-truth patch p to compare $\hat{p}$ with. Instead, it assumes that the codebase C comes with a comprehensive test suite T that pins down the intended behaviors of the program. Therefore, if the patched codebase ($\hat{p}$ applied to C) passes all tests in T and additionally mitigates *poc*, then $\hat{p}$ is correct. Perhaps due to its close resemblance to typical CI/CD pipelines in mature codebases when bugfixes are introduced, test suite-based validation has emerged as the predominant method in AVR research and practice, adopted by the majority of state-of-the-art tools including PatchAgent [80], San2Patch [23], SWE-Agent [76] and others [45, 58, 83, 85].

However, while an AVR tool should never has access to the ground-truth patch p when generating $\hat{p}$, it does not implies that we should evaluate the AVR tool completely ignoring p. In fact, based on the generally accepted principle that *new code should have a new test* we expect new tests t associated with the ground-truth patch p in mature open-source projects—effectively, this means an updated test suite T' in the patched codebase. So this raises a question: **shall we evaluate $\hat{p}$ with T' instead of T and *poc*?**

This question seems trivial at first glance, as it is tempting to assume that t and *poc* are equally useful in evaluating $\hat{p}$ (hence T combined with *poc* is effectively the same as T'). However, as shown in §3, a patch $\hat{p}$ that mitigates *poc* and passes T does not mean that it fixes the vulnerability in the way intended by the developer, and such intention can be encoded in the new test t associated with the patch p from the developers. This validation gap raises a critical question: **does the use of T and *poc* validation substantially overestimate the performance of AVR tools due to their inability to guarantee patch correctness?**

We name the new test t written by developers for the official patch p as PoC$^+$ in this paper to honor the fact that t is often derived from the *poc* but potentially encodes more semantics in terms of how developers intend to fix the vulnerability, and we argue that the auto-generated patch $\hat{p}$ of AVR tools should be evaluated using T', including the PoC$^+$. We constructed a dataset called PVBench, comprising 209 cases across 20 open-source projects. Each case includes both basic tests (T and *poc*) and PoC$^+$ tests (t). Using PVBench,

Table 1: Taxonomy of Patch Validation Methodologies Used in Previous AVR Works

Method	Category	Papers	Key Characteristics
Manual Comparison	Limited Scope	Conch [73], FixMorph [56], SkyPort [59], TSB-Port [78], PortGPT [29]	Check if patch is semantically equivalent to ground truth; The AVR tool is constrained to specific bug types (e.g., null dereference) or scenarios (e.g., backporting) only.
	General Scope	Senx [18], APPatch [40], VRpilot [24]	Check if patch fixes the bug without breaking code functionality; The AVR tool handles various vulnerability types without constraints.
Similarity Metrics	Similarity Scoring	VulMaster [86], VulRepair [8], VRepair [5]	Automated evaluation using common code similarity metrics: EM, BLEU-4, and CodeBLEU metrics.
Test Suite Validation	PoC-only Testing	ExtractFix [10], CodeRover-S [84], SAVER [15], VFix [75], CrashFixer [33]	Validate patches by executing PoC to confirm vulnerability mitigation; focuses on security-specific validation but may overlook functional correctness.
	Full Testing	CrashRepair [58], CPR [57], Fix2Fit [9], PatchAgent [80], San2Patch [23], WilliamT [85], VulnFix [83], Zero-Shot [45]	Validate patches by executing both PoC tests and the program's existing functional test suites; ensures patches fix vulnerabilities while maintaining program functionality (test coverage dependent).

we re-evaluate three state-of-the-art AVR tools, PatchAgent [80], San2Patch [23], and SWE-Agent [76, 82]. Our results reveal that over 40% of patches validated as correct by basic tests failed when evaluated against PoC$^+$ tests, corresponding to a high false discovery rate in statistical terms. This stark contrast exposes a critical weakness in test suite-based evaluation: commonly used validation methods substantially overestimate AVR tool effectiveness.

To assess whether PoC$^+$ tests serve as a reliable patch validation method, we manually compared patches that passed PoC$^+$ tests against developer-written patches. We found that over 70% of PoC$^+$-passing patches achieve semantic equivalence with the developer's patch, demonstrating PoC$^+$'s effectiveness in capturing developer intent and the inherent repair logic. The remaining patches exhibit issues such as suboptimal complexity or other quality concerns, which we detail in §6. Additionally, we analyze patches that fail PoC$^+$ tests and summarize the failure reasons in §7.

To understand how PoC$^+$ tests are commonly created by developers (and why they encode more program semantics than the *poc*), we surveyed the PoC$^+$ tests in PVBench and identified three categories based on their validation mechanisms: *Output Checking*, *Intermediate Checking*, and *Self Checking*. Across all categories, developers transform the original PoC by capturing the expected behavior of the patched program, whether through output comparison, intermediate state assertions, or embedded runtime checks, and encoding these expectations into the test specification. Unlike a PoC, which only observes whether the program crashes, a PoC$^+$ test encodes richer program semantics by explicitly specifying the expected correct behavior. This additional semantic information enables more precise patch validation: a patch that merely suppresses a crash without restoring correct functionality will fail the PoC$^+$ test, whereas it might pass a crash-only PoC check.

In summary, our work makes four main contributions:

- We propose PoC$^+$ tests as an improved patch validation method for evaluating AVR tools and introduce PVBench to investigate their advantages over traditional test suite-based validation. We also describe our validation methodology and the process for producing these tests.
- We evaluate three state-of-the-art LLM-based AVR systems on PVBench, revealing that over 40% of patches validated as correct by basic tests fail when subjected to PoC$^+$ tests, demonstrating substantial overestimation in current evaluation practices for modern AVR tools.
- We validate the reliability of PoC$^+$ tests by comparing patches that pass them against developer patches. Over 70% achieve semantic equivalence with developer patches, while the remainder exhibit performance issues or other concerns. In other words, PoC$^+$ does not fully capture every detail in developers' intention, but is still an important step forward.
- We provide a comprehensive analysis of validation inadequacies by systematically categorizing falsely correct patches into three categories which might help guide APR tools towards better repair process.

To foster further research, we have released the PVBench and evaluation artifacts at https://anonymous.4open.science/r/PVBench-91BC. Additional details are provided in Appendix B.

2 BACKGROUND: PATCH VALIDATION IN AVR

This section surveys patch validation methods used in previous AVR works (Table 1), examining their practices and potential limitations.

Manual Comparison. This approach relies on human experts to assess patch quality through direct comparison with developer-provided patches. *Limited scope systems* target specific vulnerability types or constrained scenarios (e.g., Conch [73] focuses on null pointer dereferences, while TSBPORT [78], SKYPORT [59], PortGPT [29], and FixMorph [56] address backporting tasks). *General scope systems* such as APPatch [40], VRpilot [24], and Senx [18] handle diverse vulnerability types, making evaluation more challenging since correct patches may differ structurally from reference solutions. Unfortunately, none of these works disclose the specific criteria examiners use during comparison, particularly when evaluating functionally correct but structurally different patches.

Limitations: Manual comparison provides arguably the highest level of accuracy, this approach is typically confined to scenarios with limited solution spaces. For instance, Conch [73] focuses specifically on null dereference vulnerabilities that require straightforward null check additions, while three other studies [56, 59, 78] examine backporting scenarios where patches are adapted from

mainline to stable versions. These backporting studies reveal that most patches require only minimal modifications—TSBPORT [78] found that 81% of backported patches involve location or namespace changes only—making manual verification feasible even for backporting tasks on large codebases (e.g., Linux kernel). For the other works that target general vulnerability repair [18, 24, 40], they all acknowledge that manual review is expensive and typically sample only a subset of cases for evaluation, highlighting the critical need for scalable automated patch validation methods.

Similarity Metrics. These approaches employ automated scoring without human intervention, typically combining three metrics: Exact Match (EM) [52] for binary character-level comparison, BLEU-4 [43] for token-level n-gram precision, and CodeBLEU [53] which extends BLEU with AST-based syntactic and data-flow semantic analysis. Representative systems include VulMaster [86], VulRepair [8], and VRepair [5].

Limitations: While code similarity metrics can be an automated way of measuring patch quality, they often fail to provide reliable indicators of patch effectiveness, as patches with high similarity scores may still fail when executed against PoCs [84]. This occurs because even minor syntactic differences—such as incorrect variable names, missing edge case handling, or subtle logic errors—can render functionally similar-looking code completely ineffective. Critical factors like proper variable scoping, correct API usage, and precise conditional logic placement are often overlooked by similarity metrics despite being essential for patch functionality.

Test Suite-Based Validation. This widely adopted method validates patches through automated test execution. *PoC testing* focuses on exploit validation, exemplified by CodeRover-S [84], ExtractFix [10], SAVER [15], and VFix [75], though it may overlook functional correctness. *Comprehensive testing* combines PoC validation with functional test suites in the pre-patched codebase, as adopted by PatchAgent [80], San2Patch [23], CrashRepair [58], William [85], Zero-Shot [45], VulnFix [83], CPR [57], and Fix2Fit [9].

Limitations: Test suite-based validation in AVR remains contentious due to debating *overfitting* concerns, where patches may pass tests without addressing root causes. While early studies argued AVR tools are susceptible to overfitting [61, 67, 74], recent work [42, 47, 72] suggests modern LLM-based AVR tools exhibit less project-specific fitting due to training on diverse codebases. However, existing research fails to explain why patches pass tests yet miss root causes, and largely excludes memory-unsafe languages like C/C++ despite well-studied vulnerability patterns [62, 64, 81]. This knowledge may bias LLMs toward symptom-fixing rather than addressing root causes as shown in §3.

State-of-the-practice. Despite these potential problems, most modern AVR research continues to rely on test suite-based validation to decide patch correctness. This is likely due to its fully automated nature [17, 28] and its resemblance on how a manually written patch is applied on a production-grade codebase, which typically, and minimally, requires that all new code pushed to production must pass the CI/CD pipeline [12] which runs all existing tests. However, test suite-based validation makes a strong assumption on the comprehensiveness of existing test suite: the test suite should cover *all* intended behaviors of the program, which is unlikely to be true even for mature codebases [19]. The practical implications of this assumption are illustrated in our motivating example (§3),

```
PHP_FUNCTION(range) {
  struct zval *user_start = /* Extract start argument */;
  struct zval *user_end = /* Extract end argument */;
  // Extract type from arguments
  uint8_t start_type = Z_TYPE_P(user_start);
  uint8_t end_type = Z_TYPE_P(user_end);
  /* If the range is given as strings,
     generate an array of characters. */
  if (start_type >= IS_STRING || end_type >= IS_STRING) {
    // VULNERABLE: condition fails when start_type=5 (IS_DOUBLE)
    // and end_type=7 (IS_ARRAY) because 5+7 = 2*6 (IS_STRING)
    if (start_type + end_type < 2*IS_STRING) {
      // ... handle mixed type inputs and convert to numeric
      goto handle_numeric_inputs;
    }
    // TYPE CONFUSION OCCURS HERE:
    //   When the vulnerable condition fails, we reach this point
    //   with non-string types, but try to access them as strings
    unsigned char low = Z_STRVAL_P(user_start)[0];
    unsigned char high = Z_STRVAL_P(user_end)[0];
    // ... character range generation logic
    return;
  }
handle_numeric_inputs:
  if (start_type == IS_DOUBLE || end_type == IS_DOUBLE) {
    // ... process numeric ranges (floats)
```

Figure 1: Type Confusion Bug in PHP Project

```
@@ -2924,8 +2924,8 @@ PHP_FUNCTION(range)
 /* If the range is given as strings,
    generate an array of characters. */
 if (start_type >= IS_STRING || end_type >= IS_STRING) {
-  if (start_type + end_type < 2*IS_STRING) {
+  if (start_type < IS_STRING || end_type < IS_STRING) {
     if (start_type < IS_STRING) {
       if (end_type != IS_ARRAY) {
         php_error_docref(NULL, E_WARNING, "...");
```

(a) Developer Patch

```
@@ -2960,6 +2960,14 @@ PHP_FUNCTION(range)
   }

   /* Generate array of characters */
+  if (Z_TYPE_P(user_start) != IS_STRING) {
+    zend_argument_value_error(1, "must be a string");
+    RETURN_THROWS();
+  }
+  if (Z_TYPE_P(user_end) != IS_STRING) {
+    zend_argument_value_error(2, "must be a string");
+    RETURN_THROWS();
+  }

   unsigned char low = Z_STRVAL_P(user_start)[0];
   unsigned char high = Z_STRVAL_P(user_end)[0];
```

(b) AVR-Generated Patch

Figure 2: Two distinct patch strategies for the type confusion vulnerability in PHP's `range()` function.

which demonstrates how flawed patches can bypass current test suites. Our quantitative analysis in §5 further reveals the extent to which test suite-based validation influences AVR performance.

3 MOTIVATION: PoC$^+$ TEST

This section presents a motivating example of a minimum viable yet incorrect patch that would pass the conventional test suite-based patch validation. We subsequently introduce PoC$^+$ tests and demonstrate their effectiveness in exposing the flaws of such patches.

3.1 Plausible Patch

We use a type confusion vulnerability (Issue #13094 [22]) in the PHP interpreter to demonstrate plausible patches. As depicted in Figure 1,

```
---TEST---
GH-13094 (range(9.9, '0') causes segmentation fault)
---FILE---
<?php
var_dump(range(9.9, '0'));
?>
---EXPECT---
array(10) {
  [0]=>float(9.9)
  [1]=>float(8.9)
  [2]=>float(7.9)
  [3]=>float(6.9)
  [4]=>float(5.9)
  [5]=>float(4.9)
  [6]=>float(3.9000000000000004)
  [7]=>float(2.9000000000000004)
  [8]=>float(1.9000000000000004)
  [9]=>float(0.9000000000000004)
}
```

Figure 3: An PoC$^+$ test derived from the PoC for PHP issue #13094, validating correct behavior for a crashing input.

this code demonstrates a type confusion vulnerability in PHP's `range()` function that occurs due to flawed type checking logic. The bug arises when the function receives arguments of specific type combinations, such as a double (floating-point number) and an array. The first condition (line 9) correctly identifies that at least one argument appears to be string-like since `IS_ARRAY(7)` ≥ `IS_STRING(6)`, so the code enters the string-handling branch. However, the inner arithmetic condition (line 12) unexpectedly fails when for example, `IS_DOUBLE(5)` + `IS_ARRAY(7)` = 12. When this condition fails, the code skips the proper type conversion logic and incorrectly attempts to access the non-string data using string accessor macros (line 19), which cause the program crash. Two patches exist to address this vulnerability: one crafted by the developer and another representing a plausible solution generated by AVR tools.

Developer Patch. The developer patch (Figure 2a) addresses the bug by correcting the flawed arithmetic condition, ensuring that mixed-type inputs are properly redirected to the numeric handling branch instead of entering the string processing path.

AVR-Generated Patch. The patch (Figure 2b) generated by AVR tool takes a defensive programming approach by adding explicit type validation at the point of string access. It inserts runtime checks to verify that both arguments are actually strings before attempting to dereference them using `Z_STRVAL_P()`, throwing appropriate error messages if non-string types are encountered.

Comparison. When evaluated against the existing PHP functional test suite and PoC exploits, both patches successfully pass all tests and effectively mitigate the vulnerability. Automated test suite-based validation would therefore conclude that both patches are correct. However, manual inspection reveals fundamental differences in their control flow behavior. The developer patch conditionally redirects mixed-type inputs to follow the numeric conversion path. In contrast, the AVR-generated patch immediately terminates execution with an error for any mixed-type input. This divergence in control flow indicates that the patches are not functionally equivalent—one may introduce unintended behavioral changes, which are not covered by any of the existing tests.

3.2 PoC$^+$ Tests Reveal the Difference

To find evidence that the AVR-generated patch is incorrect, we observe that the developers not only provided the patch but also created a new test when fixing the bug, as is typical practice [35], and the test case is shown in Figure 3. This test case follows the PHP Test (PHPT) format. The format is a structured test specification that consists of several sections: the `–TEST–` section provides a descriptive name for the test case, the `–FILE–` section contains the actual PHP code to be executed, and the `–EXPECT–` section defines the exact output that should be produced. Given that the PHP code within this new test case closely resembles the original PoC, we refer to it as a PoC$^+$ test. When we run the PoC$^+$ tests on the program with the AVR-generated patch applied, we obtain the following output:

```
Fatal error: Uncaught ValueError: range():
Argument #1 ($start) must be a valid string in /test.php:2
Stack trace:
#0 /test.php(2): range(9.9, '0')
#1 {main}
  thrown in /test.php on line 2
```

This output differs significantly from the expected output. According to the PHP specification [48], the `range()` function is designed to be permissive with respect to argument types: when given a mixture of numeric and string arguments, it performs type coercion and generates a numeric range if either argument is numeric. For example, a call such as `range(9.9, "0")` is valid and expected to produce a descending array of floating-point numbers, as demonstrated in the expected output of the PoC$^+$ test case. The developer's patch preserves this intended behavior by redirecting mixed-type arguments to the numeric range generation logic. In contrast, the AVR-generated patch breaks this specification: by enforcing that both arguments must be strings in the string-handling branch, it rejects mixed-type inputs and raises a runtime error, thereby violating the established PHP semantics. Thus, the AVR-generated patch is incorrect, as it introduces a change that deviates from the PHP language specification. The PoC$^+$ test automatically exposes this deviation, thus demonstrating the value of such tests for automated patch validation. In other words, when evaluating an AVR system, the generated patch should be validated against the post-patch test suite instead of the pre-patch test suite.

4 PoC$^+$ TEST DATASET & VALIDATION

This section presents the PVBench dataset and the validation method of different PoC$^+$ tests.

4.1 PVBench Overview

Vulnerability Statistic. PVBench provides a benchmark by incorporating 209 real-world vulnerabilities from 20 open-source projects. These projects represent widely-used systems with extensive codebases and robust test suites, ensuring the vulnerabilities reflect security issues encountered in production environments. As shown in Table 2, PVBench covers a diverse range of 12 CWEs, with memory safety issues being most prevalent, including NULL Dereference, UAF, and Heap Overflow, alongside control flow vulnerabilities such as Reachable Assertion and various data handling issues including Integer Overflow.

Selection Criteria. We identify open-source projects and vulnerabilities for PVBench by following a systematic workflow. First, we identified open-source projects based on GitHub star counts,

Table 2: Overview of Projects and Vulnerabilities in PVBench

Project	LoC	#	Test	Project	LoC	#	Test
php [49]	1390.2K	43	18.7K	vim [36]	564.2K	11	5.2K
cpython [51]	745.9K	33	48.6K	hdf5 [63]	1334.4K	8	0.6K
llvm [31]	8980.4K	26	128.7K	exiv2 [6]	93.5K	7	0.3K
v8 [14]	6225.6K	24	53.7K	wabt [68]	514.9K	5	1.1K
libxml2 [13]	200.4K	19	3.3K	hermes [7]	590.0K	4	2.3K
icu [65]	1241.5K	15	2.0K	pcap++ [44]	160.0K	3	0.3K
quickjs [3]	78.8K	2	79.7K	libtiff [30]	109.0K	1	0.2K
mruby [37]	152.4K	2	1.7K	jasper [20]	5.5K	1	0.2K
jq [21]	4.7K	2	0.9K	simdjson [60]	547.5K	1	0.1K
htslib [55]	108.3K	1	0.4K	wireshark [70]	6088.9K	1	0.1K
Total Vulnerabilities: 209							

CWE	#	Description	CWE	#	Description
CWE-476	52	NULL Dereference	CWE-670	3	Incorrect Control Flow
CWE-617	40	Reachable Assertion	CWE-415	3	Double Free
CWE-122	34	Heap Overflow	CWE-704	3	Type Confusion
CWE-416	32	Use After Free	CWE-457	1	Uninitialized Memory
CWE-190	26	Integer Overflow	CWE-362	1	Race Condition
CWE-121	13	Stack Overflow	CWE-369	1	Divide by Zero

Table 3: PoC$^+$ Test Category Distribution by Projects

Category	Projects
Output Checking	exiv2, hermes, htslib, jasper, libxml2, php jq, llvm-project, simdjson, wabt, wireshark
Intermed. Checking	hdf5, icu, pcapplusplus, libtiff
Self Checking	cpython, mruby, quickjs, v8, vim

prioritizing those with substantial vulnerability histories that are well-documented through GitHub issues or dedicated vulnerability tracking systems. This process led to the selection of 20 high-quality projects. Subsequently, we conducted a manual review of vulnerabilities reported in these projects over the past ten years, collecting cases that satisfy the following three requirements:

Reproducibility: Each vulnerability must include build scripts and at least one PoC to enable compilation and reproduction.

Test Coverage: Each vulnerability must be accompanied by a functional test suite and PoC$^+$ tests developed by the maintainers.

Functional Preservation: The vulnerability have been fixed and the set of functionalities remain unchanged before and after applying the fix, ensuring that patches only resolve security vulnerabilities rather than adding a new feature. This requirement is necessary because otherwise PoC$^+$ tests may validate functionality that does not exist in the unpatched program.

4.2 How Developers Create PoC$^+$ Tests

The PoC$^+$ test is derived from a PoC for the vulnerability. Unlike common PoCs, which typically only observe if the program crashes, the PoC$^+$ test performs more comprehensive validations to determine if the program behaviors are as expected. These validations include observing the output content or other intermediate running results. Based on the specific behaviors validated by the PoC$^+$, we classified the PoC$^+$ tests into three categories: *Output Checking*, *Intermediate Checking* and *Self Checking*. The distribution of the three categories across the 20 projects is shown in Table 3.

```
class issue_2377_buffer_overflow(metaclass=CaseMeta):
    filename = "$data_path/issue_2377_poc.mp4"
    commands = ["$exiv2 $filename"]
    retval   = [253]
    stderr   = ["$filename: No Exif data found in the file\n"]
    stdout   = ["File name: $filename\nFile size: 225 Bytes\n..."]
```

```
// CHECK-LABEL: func.func @nested_muli() -> i32 {
// CHECK:         %[[VAL_0:.*]] = "test.constant"() ...
// CHECK:         %[[VAL_1:.*]] = arith.muli %[[VAL_0]], ...
func.func @nested_muli() -> (i32) {
  %0 = "test.constant"() {value = 0x7fffffff} : () -> i32
  %1 = "test.constant"() {value = -2147483648} : () -> i32
  ...
}
```

```
<!-- oss-fuzz-51295_0.xsd (input) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="e" substitutionGroup="e"/>
</xs:schema>

<!-- oss-fuzz-51295_0.err (expected error output) -->
element decl. 'e': The element declaration 'e' defines
a circular substitution group to element declaration 'e'.
```

Figure 4: PoC$^+$ Test Examples for Output Checking

Output Checking. This category applies to programs that process external input files or byte streams and produce observable output. During patch validation, the PoC$^+$ test executes the program with the AVR-generated patch applied, feeds it the original PoC input, and compares the actual output against the expected output stored as part of the test. The structure of *Output Checking* varies by project. As illustrated in the motivation example (Figure 3), PHP's PoC$^+$ tests use the PHPT format. Figure 4 demonstrates three other representative formats: Exiv2 employs a Python-based framework specifying the input file, command, expected return value, and stderr/stdout content; LLVM uses inline CHECK directives to verify generated intermediate representation; and libxml2 separates input XML schemas from expected error messages in distinct files. Despite these differences, all formats share a common structure of defining inputs and expected outputs.

The testcase production process for this category is straightforward: compile the program with the correct patch applied, execute it with the PoC as input to capture the expected output, and format the result according to the project's test framework conventions. We observed that several projects in PVBench, including Hermes, libxml2, LLVM, PHP, and Wabt, have implemented automated scripts using similar methodologies. The locations of these production scripts are provided in Table 7.

Intermediate Checking. This category applies to vulnerabilities in library codebases where the original PoC is a source file (i.e., a harness) that invokes a sequence of API functions. During the patch validation stage, the PoC$^+$ test executes a modified harness that encodes expected intermediate results and determines success based on the return value. Consider the PoC $\rightarrow$ PoC$^+$ transformation for a bug in the HDF5 library, as illustrated in Figure 5. The original PoC consists of a sequence of API calls. The PoC$^+$ test modifies this harness by inserting `CHECK` and `VERIFY` macros to assert expected intermediate behavior at each step. Specifically, the bug-triggering call is checked by the assertion `VERIFY(ret, FAIL, ...)`. This structure ensures that the patched function correctly detects the

```c
// Original PoC for hdf5 bug
space_id = H5Screate_simple(1, dims, NULL);
H5Sselect_hyperslab(space_id, ...);
H5Sset_extent_none(space_id);
H5Sget_select_hyper_blocklist(space_id, ...); // Vulnerability Triggered Here
H5Sclose(space_id);
```

```c
static void poc_plus_test(void) {
  hsize_t dims[]  = {10}; /* ... initialization ... */
  space_id = H5Screate_simple(1, dims, NULL);
  CHECK(space_id, H5I_INVALID_HID, "H5Screate_simple");
  ret = H5Sselect_hyperslab(space_id, ...);
  CHECK(ret, FAIL, "H5Sselect_hyperslab");
  ret = H5Sset_extent_none(space_id);
  CHECK(ret, FAIL, "H5Sset_extent_none");
  ret = H5Sget_select_hyper_blocklist(space_id, ...);
  VERIFY(ret, FAIL, "H5Sget_select_hyper_blocklist");
  ret = H5Sclose(space_id);
  CHECK(ret, FAIL, "H5Sclose");
}
```

Figure 5: PoC$^+$ Test Example for Intermediate Checking

invalid input state and returns the appropriate error code (`FAIL`) rather than crashing, thereby validating the patch.

Developers produce these test cases by instrumenting API calls to capture both return values and pointer-based outputs at runtime. Since C lacks multiple return values, APIs often use pointer arguments for additional outputs. Checking logic is then inserted at an appropriate abstraction level to validate these captured values.

Self Checking. This category targets interpreter programs, such as Python, JavaScript engines, or Ruby interpreters, where vulnerabilities are triggered by executing code written in the interpreted language. The goal of PoC$^+$ construction is to transform the original PoC script into a self-validating test that explicitly verifies expected runtime behavior. Rather than merely observing whether the interpreter crashes, the PoC$^+$ test embeds assertions within the interpreted program itself to confirm that the patched interpreter properly handles previously vulnerable inputs by raising appropriate exceptions, producing specific error messages, or returning expected values. This transformation requires understanding the correct post-patch behavior and inserting corresponding exception handlers and assertions to validate the interpreter's response. Consider the example transformation for a CPython bug illustrated in Figure 6. The original PoC consists of simple function calls that trigger a crash in the unpatched interpreter. The PoC$^+$ test transforms this into a self-validating script by wrapping each vulnerable call in a `self.assertRaises(TypeError)` context manager, verifying both that the expected exception type is raised. This structure ensures that the patched interpreter correctly detects malformed input and raises appropriately typed exceptions.

To produce self-checking tests, developers first execute the original PoC on the patched interpreter to observe the corrected behavior, whether it raises a specific exception, returns a particular value, or outputs an error message. They may also transform the PoC to capture all possible error paths, ensuring comprehensive coverage of the fix. Developers then wrap vulnerable code paths in appropriate assertion constructs. Since the interpreters in PVBench are all about dynamic languages, developers may also verify side effects—such as changes to global state, object attributes, or resource handles—to confirm that the interpreter maintains correct behavior beyond just the immediate return value.

```python
# Original PoC for CPython sys bug
import sys
sys.remote_exec(0, None)
```

```python
# PoC+ test with self-test
import sys
with self.assertRaises(TypeError):
    sys.remote_exec(0, None)
with self.assertRaises(TypeError):
    sys.remote_exec(0, 123)
```

Figure 6: PoC$^+$ Test Example for Self Checking

5 QUANTIFYING OVERESTIMATION

In this section, we examine the extent to which conventional test suite-based validation overestimates AVR system effectiveness.

5.1 Methodology

We evaluated three state-of-the-art AVR systems: PatchAgent [80], San2Patch [23], and SWE-Agent[76] on PVBench to quantify the extent to which test suite-based validation overestimate AVR system performance. All tools use the LLM-based approach. Since the original SWE-Agent does not support C/C++ programs, we use its multi-language version [82]. For each tool, we conducted experiments with two different large language models: GPT-4.1 version *gpt-4.1-2025-04-14* [41] and Claude-4 Sonnet version *claude-sonnet-4-20250514* [1]. To account for the non-deterministic nature of LLM-based patch generation, we executed each configuration, i.e., a (AVR tool, LLM version) pair, five times on every test case, resulting in 1045 (209×5) patch attempts per each configuration for all 209 vulnerabilities, where each vulnerability will receive at most 30 (6 configurations $\times$ 5 attempts) patches.

Our experiment employs a two-stage validation framework designed to expose the limitations of conventional validation approaches. In Stage 1 (Basic Validation), generated patches are validated using the conventional approach: verifying PoC mitigation and executing the project's existing functional test suite. Patches passing both criteria are classified as "correct" under conventional validation. In Stage 2 (PoC$^+$ Validation), patches deemed correct in Stage 1 are further evaluated using PoC$^+$ tests to assess whether the patch also conforms to the semantics or developers' intention encoded in the PoC$^+$ test. This stage reveals false positives, i.e., patches that appear correct under basic tests but fail on PoC$^+$ tests.

5.2 General Results

Our experiment results reveal a substantial gap between conventional test suite-based validation and rigorous PoC$^+$ test validation. As shown in Table 4, PatchAgent with GPT-4.1 generated 798 patches that passed basic tests among 1045 executions (209 vulnerabilities $\times$ 5 attempts), achieving an initial success rate of 76.4%. Similarly, PatchAgent with Claude Sonnet-4 achieved an initial success rate of 83.5% However, when these seemingly correct patches were subjected to PoC$^+$ test validation, the success rates dropped dramatically to 44.5% and 50.1%, respectively, exposing false discovery rates (FDR) of 41.7% and 40.1%.

San2Patch demonstrated lower overall patch generation rates but similar validation reliability issues, with initial success rates of

Table 4: Performance of AVR tools under different validation. init: patches passing basic tests; poc+: patches also passing PoC$^+$ tests; FDR: false discovery rate, i.e., the fraction of initially validated patches that fail subsequent testing.

Tool	Model	init	poc+	FDR
PatchAgent	Sonnet-4	83.5%	50.1%	40.1% (350/873)
	GPT-4.1	76.4%	44.5%	41.7% (333/798)
San2Patch	Sonnet-4	41.3%	20.7%	49.8% (215/432)
	GPT-4.1	37.9%	19.6%	48.2% (191/396)
SWE-Agent	Sonnet-4	29.0%	19.6%	32.3% (98/303)
	GPT-4.1	14.4%	8.3%	41.3% (63/150)
Overall		47.1%	27.1%	42.3% (1250/2952)

37.9% (GPT-4.1) and 41.3% (Sonnet-4) declining to 19.6% and 20.7% under PoC$^+$ validation, resulting in FDRs of 48.2% and 49.8%. SWE-Agent exhibited the poorest performance with initial success rates of only 14.4% (GPT-4.1) and 29.0% (Sonnet-4), further declining to 8.3% and 19.6% respectively under PoC$^+$ validation. The full results are provided in Figure 13.

Across all configurations, our evaluation consistently reveals an FDR around 40%, i.e., about 40% of patches that pass basic tests fail on PoC$^+$ tests. To analyze the repair outcomes for different vulnerabilities in our experiments. Figure 7 illustrates the distribution of vulnerabilities based on the ratio of correct patches to false positive patches generated, revealing four distinct behavioral patterns:

Repair-Resistant Vulnerabilities. 6.70% of vulnerabilities cluster at the origin (no patch passes any tests), representing cases where AVR tools completely fail to generate any patches that pass basic validation. These vulnerabilities consistently challenge all tested AVR configurations, indicating limitations in current tools.

High-Success Repair Targets. 31.10% of vulnerabilities demonstrate excellent repairability, generating correct patches without producing any FPs. The distribution peaks at 12.44%, representing repair scenarios with 19-24 correct patches and zero FPs, while an additional 6.70% produce 7-12 correct patches with perfect accuracy.

False-Positive Prone Vulnerabilities: 28.24% of vulnerabilities generate exclusively false positive patches while producing zero genuinely correct patches that pass comprehensive testing. The distribution peaks at 9.09% for cases producing 7-12 false positives with no correct patches. These scenarios are particularly concerning for real-world deployment, as AVR tools consistently generate plausible but fundamentally flawed fixes, creating a dangerous illusion of successful repair that could introduce new security risks.

Mixed-Performance Vulnerabilities: 33.96% of vulnerabilities exhibit mixed repair patterns across intermediate performance regions, generating varying combinations of correct and incorrect patches in low-to-moderate quantities.

The vulnerability distribution reveals a complex, multi-modal pattern demonstrating varied AVR tool performance across real-world vulnerabilities. Rather than showing a continuous distribution, the data exhibits distinct clustering with 6.7% producing no patches, 31.10% achieving strong repair outcomes, and 28.24% generating only false positives. This clustering pattern suggests that repairability is predominantly an intrinsic characteristic of

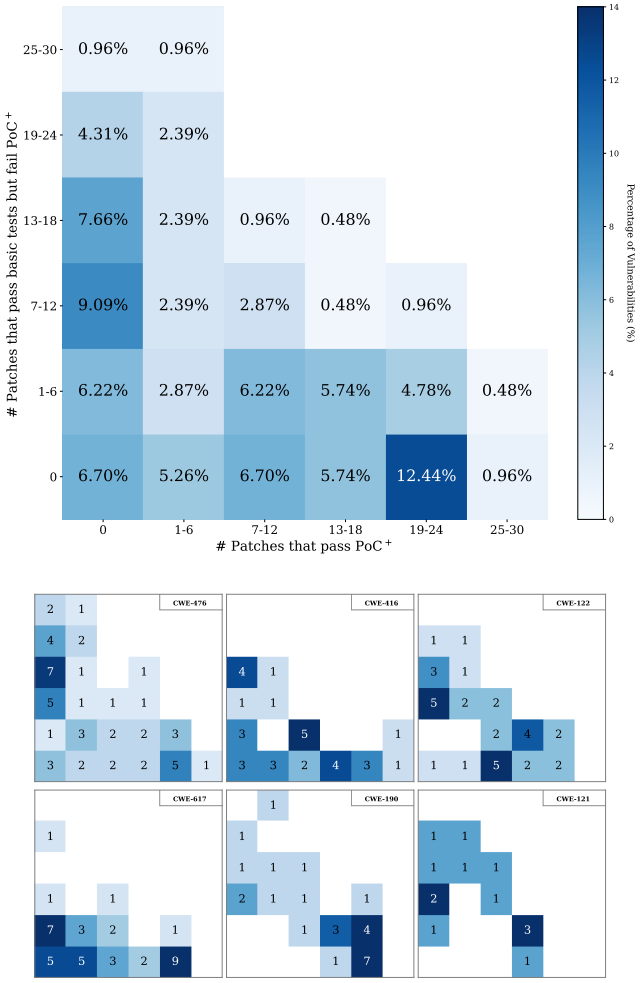

Figure 7: Distribution of vulnerabilities: The x-axis shows #patches passing PoC$^+$ tests, while the y-axis shows #patches passing basic tests but failing PoC$^+$ tests.

the vulnerability itself rather than a consequence of specific tool configurations or parameter choices.

To understand these clustering patterns by CWE category, we conducted detailed analysis on six prevalent ($\geq$ 10 samples) CWE types in our dataset. CWE-617 (Reachable Assertion) emerges as the most resistant vulnerability type to automated repair, likely because LLMs lack extensive training data on developer-introduced assertions compared to common memory errors. However, when patches are successfully generated for CWE-617, they demonstrate higher reliability than other vulnerability types. Among memory errors, NULL pointer dereference (CWE-476) and stack-based buffer overflow (CWE-121) both exhibit high false discovery rates. This may occur because these vulnerabilities typically have standardized repair methods, leading LLMs to generate superficial fixes—null checks for CWE-476 and boundary checks for CWE-121—rather than addressing the underlying architectural issues. Use After Free

Table 5: Manual Categorization of Patches that Passed PoC$^+$ Tests Across Different AVR Tools: The table shows the distribution of four quality categories Results are compared across three tools using two LLMs .

Category	PatchAgent		San2Patch		SWE-Agent		Total
	Sonnet-4	GPT-4.1	Sonnet-4	GPT-4.1	Sonnet-4	GPT-4.1	
Semantic Equivalent	396 (75.72%)	339 (72.90%)	156 (71.89%)	150 (73.17%)	159 (77.56%)	66 (75.86%)	1266 (74.38%)
Performance Issue	25 (4.78%)	24 (5.16%)	1 (0.46%)	3 (1.46%)	3 (1.46%)	0 (0.00%)	56 (3.29%)
Suboptimal Repair	59 (11.28%)	61 (13.12%)	28 (12.90%)	27 (13.17%)	20 (9.76%)	13 (14.94%)	208 (12.22%)
Check Circumvention	43 (8.22%)	41 (8.82%)	32 (14.75%)	25 (12.20%)	23 (11.22%)	8 (9.20%)	172 (10.11%)
Total	**523 (100%)**	**465 (100%)**	**217 (100%)**	**205 (100%)**	**205 (100%)**	**87 (100%)**	**1702 (100%)**

(CWE-416) and Heap-based Buffer Overflow (CWE-122) demonstrate more reliable repair outcomes. Integer overflow (CWE-190) also employs general fix methods but shows high reliability, possibly because upgrading to larger data types is a common practice in real-world development.

6 RELIABILITY OF PoC$^+$ TEST

In this section, we examine the reliability of PoC$^+$ tests as a validation method for evaluating patch correctness.

6.1 Methodology

To assess the reliability of PoC$^+$ tests and identify potential issue of using them for AVR tools evaluation, we conducted a manual review of all patches that passed PoC$^+$ tests by comparing them with developer patches. The results are presented in Table 5. We classified these patches into four categories based on their semantic closeness to the developer patches. Two patches are considered semantically equivalent if they satisfy the following criteria:

- Both patches target the same function
- Both patches have identical time and space complexity
- The logic implemented by both patches is functionally equivalent with no unintended side effects

For patches that do not achieve semantic equivalence, we proceed with further categorization. If we find that a patch uses ad-hoc methods to bypass security or functional checks, we classify it as an *Check Circumvention*, as such approaches are in fact incorrect implementations despite passing all tests, including PoC$^+$. For patches with correct functionality but different performance characteristics, we evaluate whether the AVR-generated solution exhibits worse time or space complexity compared to the developer solution. Such cases are classified as having *performance issues*. Finally, for patches that are functionally correct and have same algorithm complexity with developers patch, we assess quality by examining whether the developer's patch is more obvious for correctness compared to the AVR-generated patch. We classify the AVR-generated patch as having *Suboptimal Repairs*.

We implemented an inter-rater reliability process [40, 78] where each author independently categorized the patches, and then cross-validated results and discussed any discrepancies until reaching consensus. **Our findings show that more than 70% of the patches maintain semantic equivalence with developer patches**, demonstrating the high reliability of the PoC$^+$ tests. The subsequent sections examine the remaining patch categories in detail.

```c
static int template_clear(TemplateObject *self) {
    Py_CLEAR(self->literal);
    for (Py_ssize_t i = 0, n = Py_SIZE(self); i < n; i++)
        // BUG: items[i].literal may be uninitialized
        Py_CLEAR(self->items[i].literal);
    return 0;
}
static PyObject *sre_template_impl(..., PyObject *template) {
    // template is a list containing interleaved
    // literal strings (str or bytes) and group indices (int)
    // [literal1, group1, literal2, ..., literalN].
    Py_ssize_t n = /* Extract number of groups */;
    // Allocate array but items[].literal not initialized
    TemplateObject *self = PyObject_GC_NewVar(TemplateObject, ..., n);
    self->literal = Py_NewRef(PyList_GET_ITEM(template, 0));
    // FIX OPTION 1: Initialize all literal fields to NULL
    // This prevents crashes since Py_CLEAR safely handles NULL
    // for (Py_ssize_t i = 0; i < n; i++)
    //     self->items[i].literal = NULL;
    for (Py_ssize_t i = 0; i < n; i++) {
        Py_ssize_t index = // Extract (i+1)-th group number;
        if (index < 0) {
            // FIX OPTION 2: Set object size to track
            // how many items were successfully initialized.
            // template_clear will only clean initialized items.
            // Py_SET_SIZE(self, i);
            goto bad_template;
        }
        // Normal case: would initialize items[i].literal here...
    }
    return (PyObject*) self;
bad_template:
    PyErr_SetString(PyExc_TypeError, "invalid template");
    Py_XDECREF(self); // Triggers template_clear() cleanup
    return NULL;
}
```

Figure 8: Example of Performance Issue

6.2 Performance Issue

Performance issues arise when patches are functionally correct but employ repair strategies with suboptimal algorithmic complexity compared to developer solutions. To evaluate performance differences, we analyze the complexity of the extra operations required specifically for the repair, rather than comparing the overall program complexity. This approach isolates the computational overhead introduced by different repair strategies. If we find that the time or space complexity of extra operations for repairing the bug by the AVR tools is higher than the developer's, we categorize the patch as having a performance issue. It is important to note that categorizing a patch as having a performance issue does not necessarily indicate that it is better or worse than the developer's

solution, as the project may not be particularly sensitive to performance considerations. This category simply identifies the difference in computational efficiency between repair approaches.

To illustrate this category, we examine a case from CPython involving an uninitialized memory access vulnerability. Figure 8 demonstrates how two algorithmically distinct repair strategies can address the same bug with different time complexity for their repair operations. The issue occurs when `PyObject_GC_NewVar()` allocates memory for a `TemplateObject` but leaves the `items[i].literal` fields uninitialized. If an error condition arises during the initialization loop (such as encountering a negative index), the code jumps to the error handler, which calls `Py_XDECREF(self)` and subsequently triggers `template_clear()`. This cleanup function blindly iterates through all allocated items and calls `Py_CLEAR()` on potentially uninitialized `literal` fields, causing crashes when the macro attempts to decrement reference counts on garbage memory values.

Two distinct repair strategies emerge to address this vulnerability, which differ significantly in their algorithmic complexity. The AVR-suggested fix takes a defensive programming approach by proactively initializing all `literal` fields to `NULL` immediately after allocation. This repair strategy has $O(n)$ complexity. In contrast, the developer patch employs a tracking-based strategy using `Py_SET_SIZE(self, i)` to record how many items have been successfully initialized. This repair approach has $O(1)$ complexity.

```
@@ -729,10 +729,14 @@ void add_class_vars(...)
    }
    prop = NULL;
    if (statics && (info->flags & ACC_STATIC) != 0) {
-       prop = &ce->static_members_table[info->offset];
+       if (ce->static_members_table &&
+           info->offset < ce->static_members_count) {
+           prop = &ce->static_members_table[info->offset];
+       }
    } else if (!statics && (info->flag & ACC_STATIC) == 0) {
-       prop = &property_table[info->offset];
+       if (property_table && info->offset < ce->property_count) {
+           prop = &property_table[info->offset];
+       }
      }
      if (!prop) {
         continue;
```

(a) AVR-Generated Patch

```
@@ -724,7 +724,8 @@ void add_class_vars(...)
    if (((info->flags & ACC_PROTECTED) &&
         !check_protected(info->ce, scope)) ||
         ((info->flags & ACC_PRIVATE) &&
-          info->ce != scope)) {
+          info->ce != scope) ||
+         (info->flags & ACC_VIRTUAL)) {
          continue;
    }
    prop = NULL;
```

(b) Developer Patch

Figure 9: Example of Suboptimal Repair

6.3 Suboptimal Repair

Suboptimal repair represents patches that are functionally correct and maintain the same algorithmic complexity as developer solutions, but exhibit inferior implementation quality that makes their correctness less apparent. With no intuitive justification on why code changes are made at specific locations, this type of patch eventually hurts the maintainability of the overall codebase. Patches in this category typically fall into two patterns.

First, AVR-generated patches often address vulnerabilities at later stages in the execution flow rather than preventing the underlying issue at its source. Similar to the incorrect root cause example in our motivating case (§3), developer patches typically fix vulnerabilities at creation sites where corrupted data structures are initially formed, while AVR tools apply defensive measures at usage sites where problems manifest. The prevention-oriented approach is more intuitively correct because it detects problems at their source and makes the code more self-documenting.

Second, developer solutions often encode richer semantic information and domain-specific knowledge compared to AVR-generated alternatives. To illustrate this pattern, consider a heap out-of-bounds vulnerability in PHP's class variable handling shown in Figure 9. The PoC of the vulnerability defines a class with virtual members, but the handling function incorrectly forgets to handle this case, causing heap out-of-bounds access. The AVR-generated patch applies defensive bounds checking by validating array pointers and offsets before access, essentially forcing a fix through defensive programming. In contrast, developer's patch adds a single condition `(info->flags & ACC_VIRTUAL)` to exclude virtual properties from processing, demonstrating semantic understanding that virtual properties should not be handled in this context. While both patches prevent the crash, developer's solution encodes the actual business logic—virtual properties are conceptually different and require separate handling—making it obviously better despite being difficult to distinguish through automated testing. This semantic richness reflects deep understanding of PHP's object model and makes the code more maintainable.

6.4 Check Circumvention

Check circumvention represent patches that attempt to bypass security or functional checks rather than addressing the underlying root cause of bugs—another way of suppressing an error! These patches prioritize immediate test passage over principled problem resolution, often employing workarounds that circumvent protective mechanisms or safety validations. Such approaches fundamentally misunderstand the purpose of security checks and program invariants, treating them as obstacles to avoid rather than indicators of deeper logical issues. Our analysis identified two primary patterns of check circumvention. First, for out-of-bounds access vulnerabilities, AVR patches frequently resort to excessive memory over-allocation, attempting to prevent crashes by allocating significantly larger buffer sizes rather than determining and allocating the exact memory size the program requires or correcting the faulty indexing logic. Second, for reachable assertion failures, patches commonly either remove assertion statements entirely or artificially manipulate variables immediately before assertions to force conditions to evaluate as true, effectively disabling safety checks without understanding why the assertions were violated.

7 FALSE POSITIVE ANALYSIS

We conducted a systematic analysis of all false positives from our PVBench experiments (§5.2) to understand the reasons why these

Table 6: Categorization of FP Patches: The table presents a breakdown of FP patches across three AVR systems using two LLMs.

Category	PatchAgent		San2Patch		SWE-Agent		Total
	Sonnet-4	GPT-4.1	Sonnet-4	GPT-4.1	Sonnet-4	GPT-4.1	
Incorrect Root Cause	144 (41.14%)	118 (35.44%)	113 (52.56%)	81 (42.41%)	37 (37.76%)	22 (34.92%)	515 (41.18%)
Specification Violation	189 (54.00%)	200 (60.06%)	93 (43.26%)	97 (50.79%)	61 (62.24%)	40 (63.49%)	680 (54.38%)
Poor Code Practice	17 (4.86%)	15 (4.50%)	9 (4.19%)	13 (6.81%)	0 (0.00%)	1 (1.59%)	55 (4.40%)
Total	**350 (100%)**	**333 (100%)**	**215 (100%)**	**191 (100%)**	**98 (100%)**	**63 (100%)**	**1250 (100%)**

```
@@ -2234,6 +2234,10 @@ PySequence_Count(...) {
 PySequence_Contains(PyObject *seq, PyObject *ob)
 {
+    if (seq == NULL) {
+        null_error();
+        return -1;
+    }
     PySeqMethods *sqm = Py_TYPE(seq)->tp_as_sequence;
     if (sqm != NULL && sqm->sq_contains != NULL) {
```

(a) AVR-Generated Patch

```
@@ -5083,19 +5083,17 @@ ast_type_init(...) {
     PyObject *key, *value, *fields, *attributes = NULL;
-    if (PyObj_GetOptionalAttr((PyObject*)Py_TYPE(self),
-                          state->_fields, &fields) < 0) {
+    fields = PyObj_GetAttr((PyObject*)Py_TYPE(self),
+                           state->_fields);
+    if (fields == NULL)
         goto cleanup;
-    if (fields) {
-        numfields = PySequence_Size(fields);
-        if (numfields == -1)
-            goto cleanup;
-        remaining_fields = PySet_New(fields);
-    }
-    else
-        remaining_fields = PySet_New(NULL);
+    numfields = PySequence_Size(fields);
+    if (numfields == -1)
+        goto cleanup;
+    remaining_fields = PySet_New(fields);
     if (remaining_fields == NULL)
         goto cleanup;
```

(b) Developer Patch

Figure 10: Incorrect Root Cause Example

patches pass the basic tests but fails PoC$^+$. In particular, we manually compared those patches with developer patches and classified them into three categories: patches that incorrectly identify the root cause of the issue, patches that violate project specifications, and patches that use poor coding practices. Table 6 presents the distribution of these categories across all tested configurations. Our analysis reveals that *Specification Violation* represents the most prevalent failure mode, accounting for approximately 55.57% of all false positive cases. This is followed by *Incorrect Root Cause* at 39.88%, while *Poor Code Practice* constitute the smallest category.

7.1 Incorrect Root Cause

We classify patches into this category when the generated patch modifies code in a different function from the developer patch, indicating fundamental misunderstanding of the true location of vulnerability. This represents the most severe type of analytical failure, where AVR tools fix symptoms close to the point of failure rather than addressing the underlying cause that eventually leads the vulnerability. To illustrate this failure mode, consider a NULL pointer dereference vulnerability in AST module of Python interpreter demonstrated by this PoC exploit:

```
import ast; del ast.AST._fields; t = ast.AST(arg1=123)
```

When `ast.AST._fields` is deleted and a new AST object is subsequently created with `ast.AST(arg1=123)`, the program crashes with a NULL pointer dereference. Figure 10 shows how the AVR tools and developer approach this problem with fundamentally different philosophies. The AVR-generated patch applies a band-aid solution by adding a defensive NULL check in `PySequence_Contains()` where the crash occurs. While this mutes the crash, it fails to address why the NULL condition arose in the first place. In contrast, the developer's patch identifies the root cause in `ast_type_init()`, where the initialization code incorrectly uses `PyObject_GetOptionalAttr()` to retrieve `_fields`. By changing to `PyObject_GetAttr()`, the patch enforces stricter validation during object creation.

The developer's approach is grounded in the AST specification, which explicitly states that *"Each concrete class has an attribute* `_fields` *which gives the names of all child nodes"* [50]. This makes `_fields` a mandatory component of the AST object model, not an optional one. The human patch recognizes this specification requirement and prevents invalid objects from being created, while the AVR patch merely handles the consequences of allowing such invalid objects to exist. This example demonstrates how incorrect root cause identification leads to patches that may prevent immediate crashes but fail to address the underlying design violation, potentially leaving the system vulnerable to related issues.

7.2 Specification Violation

We classify patches into this category when the generated patch correctly identifies the problematic code location but produces modifications that violate established software specifications, programming language standards, or documented functional requirements. These violations typically manifest as changes to input validation logic, return value semantics, or control flow that contradict to properties or contracts communicated otherwise. As demonstrated in §3 with the PHP `range()` function, the AVR-generated patch enforced strict type checking by rejecting mixed-type inputs with runtime errors, while the PHP language specification requires permissive type coercion that allows mixed numeric and string arguments to generate valid numeric ranges. This violation demonstrates how patches can on one hand fix vulnerabilities while on the other hand fundamentally breaking expected program functionality.

Patches in specification violation category demonstrate partial understanding of the vulnerability context but fail to respect the intended functionality of the target system overall. This failure mode

```
@@ -661,8 +661,12 @@ getArgumentAccessInfo(...) {
  auto TypeSize = DL.getTypeStoreSize(Ty);
  if (!TypeSize.isScalable() && Offset) {
    int64_t Size = TypeSize.getFixedValue();
-   return ConstantRange(APInt(64, *Offset, true),
-                        APInt(64, *Offset + Size,
-                        true));
+   if (Size > 0) {
+     int64_t End = *Offset + Size;
+     if (End > *Offset)
+       return ConstantRange(APInt(64, *Offset, true),
+                            APInt(64, End, true));
+   }
  }
  return std::nullopt;
};
```

(a) AVR-Generated Patch

```
@@ -661,8 +661,13 @@ getArgumentAccessInfo(...) {
  auto TypeSize = DL.getTypeStoreSize(Ty);
  if (!TypeSize.isScalable() && Offset) {
    int64_t Size = TypeSize.getFixedValue();
-   return ConstantRange(APInt(64, *Offset, true),
-                        APInt(64, *Offset + Size,
-                        true));
+   APInt Low(64, *Offset, true);
+   bool Overflow;
+   APInt High = Low.sadd_ov(APInt(64, Size, true),
+                            Overflow);
+   // Bail if the range overflows signed 64-bit int.
+   if (Overflow)
+     return std::nullopt;
+   return ConstantRange(Low, High);
  }
  return std::nullopt;
};
```

(b) Developer Patch

Figure 11: Domain Ignorance Example

indicates that AVR systems need improved adherence to program specifications, requiring better integration of a knowledge base that captures intended program behaviors, and potential constraint validation during patch generation.

7.3 Poor Code Practice

We classify patches into this category when the generated patch correctly identifies the vulnerability location and maintains specification compliance but still fails PoC$^+$ due to poor code quality or violation of established coding practices. These patches demonstrate adequate analytical capabilities of LLM but reveal insufficient technical domain knowledge or adherence to project-specific design principles. This failure mode indicates that AVR systems need enhanced code generation capabilities by incorporating better understanding of platform-specific requirements, software engineering best practices, and sometimes even intricacies in compiler behaviors. We provide two examples to illustrate patches in this category. One example involves an AVR-generated patch that violates C/C++ programming standards (i.e., causing undefined behavior), which is shown in Figure 11. The vulnerability occurs when computing memory access ranges in the `FunctionAttrs()` optimization pass, where `*Offset + Size` can overflow and wrap around.

The AVR patch in Figure 11a demonstrates overflow detection by adding a simple comparison `End > *Offset` to check for overflow. However, this approach is flawed because it relies on undefined behavior—when signed integer overflow occurs in C++, the comparison itself invokes undefined behavior, and aggressive

```
  GET_NODE(sxe, node);
+ if (!node) {
+   /* avoid null dereference */
+   return &EG(err_zval);
+ }
  php_libxml_invalidate_node_from_doc(node->doc);
  if (node) {
       if (attribs) {
```

Figure 12: Logic Shortcuts Example

compiler optimizations may eliminate the check entirely, assuming that signed overflow never occurs. The patch creates a false sense of security while potentially being optimized away at compile time. In contrast, the developer patch in Figure 11b correctly uses LLVM's `APInt::sadd_ov` method, which is specifically designed for overflow-safe arithmetic operations. This approach uses well-defined overflow detection mechanisms that cannot be optimized away by compilers, properly handling the edge case by returning `std::nullopt` when overflow is detected. The developer solution demonstrates deep understanding of both the vulnerability context and the platform-specific requirements for reliable overflow detection in optimized code.

The other example demonstrates how a AVR-generated patch disregards the control flow logic carefully maintained by developers. This example encompasses patches that introduce logically destructive code structures that break the developer's intended design patterns. As illustrated in Figure 12, the generated patch adds an early null check that immediately returns when `node` is null, bypassing the developer's carefully structured conditional logic that was designed to handle null cases gracefully within the existing control flow (the matching `else` block for the `if` condition at line 7). This approach creates destructive code in the later `if(node)` check, as the condition will always be true since null values have already been filtered out by the early return, and also violates the developer's intent to maintain a unified error handling strategy throughout the function. The correct approach respects the original design by moving the `php_libxml_invalidate_node_from_doc` call inside the existing `if(node)` conditional, preserving the developer's intended control flow while eliminating the vulnerability without introducing structural inconsistencies.

8 IMPLICATIONS FOR AVR RESEARCH

Our evaluation on PVBench reveals that conventional test suite-based validation methods may substantially overestimate the effectiveness of AVR tools, with over 40% of patches that pass basic tests failing to pass PoC$^+$ tests. Our further analysis of these false positive cases suggests that the primary cause is specification violations, where patches produce modifications that violate established software specifications or documented behavioral requirements. These findings suggest two main implications for AVR research.

AVR research could benefit from adopting more reliable validation methodologies during evaluation beyond simply running PoC exploits and existing functional tests. The discovery of such high false discovery rates (40%+) across all tested state-of-the-art AVR systems indicates that current validation practices may create an illusion of effectiveness. This systematic overestimation could undermine confidence in deployment decisions

and suggests that some published success rates might be inflated. To address this potential gap, future AVR evaluation frameworks could incorporate multi-layered validation approaches, including developer-authored functional tests like PoC$^+$ tests, manual comparison against developer patches to assess semantic equivalence and formal verification techniques. Research might prioritize developing automated methods to generate or identify comprehensive test suites that capture true functional requirements, while AVR benchmarks could consider including more rigorous validation criteria that help ensure patches meet both security and specification requirements for potential production deployment.

Current AVR approaches that rely primarily on codebase and vulnerability information as input may be insufficient for generating production-ready patches. Our categorical analysis in §7 reveals that modern LLM-based AVR tools often struggle to understand program specifications, API semantics, and behavioral requirements that appear essential for correct repairs. The prevalence of specification violations suggests that many critical requirements are documented in external sources rather than being easily inferable from code alone. AVR systems could benefit from incorporating information from project documentation, API specifications, coding guidelines, comments, and other textual resources that capture intended program behavior and constraints. Future AVR research might explore methods to automatically extract and leverage such documented requirements, develop techniques to integrate natural language specifications with code analysis, and create approaches that can reason about both explicit code patterns and implicit behavioral expectations described in documentation. This potential shift toward incorporating documented knowledge alongside code analysis could represent an important evolution for generating patches that respect both syntactic correctness and documented program intentions.

9 DISCUSSION

Scope and Limitations. While our study provides insights into the limitations of conventional test suite-based patch validation, several constraints limit the generalizability of our findings. First, PVBench focuses exclusively on C/C++ programs across 20 open-source projects, which may not represent the full spectrum of programming languages and software domains where AVR tools are applied. Different languages have varying features (e.g., type systems) and testing cultures that could influence both vulnerability patterns and validation effectiveness. Second, improved patch validation methods do not directly enhance the effectiveness of AVR tools. Although some prior work suggests that better patch validation can provide more informative feedback to improve AVR systems, our test generation approach still relies on developer-provided correct patches rather than generating tests from scratch. Consequently, our method is primarily suited for evaluation purposes rather than integration into an end-to-end repair pipeline. We leave the generation of PoC$^+$ tests from scratch for future work.
Automated Software Engineering. Building PVBench required substantial effort due to challenges such as resolving complex dependencies and aggregating information from multiple sources. This labor-intensive process consumed considerable time, limiting both dataset scale and diversity. Our experience highlights the critical need for automated software engineering. Recent advances have begun to address these challenges: CompileAgent[16] automates repository-level compilation with LLM agents, while ExecutionAgent[4] can automatically executes functional test suites. Automated dataset construction systems like SWE-smith[77] and SWE-rebench[2] have demonstrated the ability to create datasets an order of magnitude larger than manual collections.

10 RELATED WORK

Patch Equivalence. Among prior works that use test suite-based methods to evaluate the patch correctness rate of their tools, some [23, 45] also employ manual comparison to count the number of generated patches that are semantically equivalent to developer patches—establishing a lower bound of the actual patch success rate. Empirical studies have shown that approximately 25% of correct AVR patches are syntactically different but semantically equivalent to developer patches [66]. Recent advances in automated semantic equivalence assessment combine syntactic and semantic similarity metrics with test coverage analysis [11], while sophisticated approaches leverage program invariants and pre-trained language models for semantic reasoning about patch equivalence [27]. Research on program equivalence for adaptive vulnerability repair has formalized the patch equivalence problem using test-equivalence relations, proposing efficient algorithms for partitioning patches into equivalence classes based on runtime behavior [69]. Our work provides a better approach to calculate a tighter upper bound of patch success rate, while the lower bound is still defined by equivalence.
Formal Verification. Formal verification approaches offer mathematically rigorous alternatives to test-suite based patch validation, providing stronger guarantees about patch correctness through specification-based reasoning and constraint solving. SemFix [38] formulates repair requirements as constraints solved by SMT solvers, providing formal guarantees about patch correctness beyond test adequacy. Subsequent advances in scalable semantics-based repair using symbolic execution with Z3 SMT solver have demonstrated practical applications of formal methods to multiline vulnerability repair [34]. Contract-based repair approaches use formal specifications such as pre- and postconditions for both patch generation and validation [46], while sound and complete mutation-based vulnerability repair provides theoretical guarantees of soundness and completeness through bounded model and SAT/SMT [54]. Advanced approaches integrate formal verification throughout the repair process, using constraint solving with mathematical foundations such as Farkas lemma for patch synthesis [39], while modular program verifiers [32] enable property-specific validation.

11 CONCLUSION

Our study suggests that current AVR evaluation practices may benefit from more comprehensive validation approaches, as we observe that over 40% of patches deemed correct by conventional test-suite methods fail when evaluated against PoC$^+$ tests across three state-of-the-art LLM-based AVR systems. Our proposed PoC$^+$ test demonstrates promising results, with over 70% of passing patches achieving semantic equivalence with developer solutions, suggesting this methodology could serve as a valuable complement to

existing validation practices. These findings encourage future research to consider incorporating more rigorous quality assessment approaches and including specification information in AVR.

REFERENCES

[1] Anthropic. 2024. Anthropic. https://www.anthropic.com/.

[2] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. arXiv:2505.20411 [cs.SE] https://arxiv.org/abs/2505.20411

[3] Fabrice Bellard. 2024. QuickJS: Small JavaScript Engine. https://bellard.org/quickjs/.

[4] Islem Bouzenia and Michael Pradel. 2025. You name it, I run it: An LLM agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1054–1076.

[5] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

[6] Exiv2. 2024. Exiv2: Image Metadata Library and Tools. https://exiv2.org/.

[7] facebook. 2024. Hermes: JavaScript Engine for React Native. https://hermesengine.dev/.

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

[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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 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 Trans. Softw. Eng. Methodol.* 30, 2, Article 14 (Feb. 2021), 27 pages. https://doi.org/10.1145/3418461

[11] Ali Ghanbari and Andrian (Andi) Marcus. 2023. Shibboleth: Hybrid Patch Correctness Assessment in Automated Program Repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 166, 4 pages. https://doi.org/10.1145/3551349.3559519

[12] GitHub. 2024. GitHub Actions Documentation. https://docs.github.com/en/actions.

[13] GNOME. 2024. libxml2: XML Parsing and Manipulation Library. http://xmlsoft.org/.

[14] Google. 2024. V8 JavaScript Engine. https://v8.dev/.

[15] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 271–283. https://doi.org/10.1145/3377811.3380323

[16] Li Hu, Guoqiang Chen, Xiuwei Shang, Shaoyin Cheng, Benlong Wu, Gangyang Li, Xu Zhu, Weiming Zhang, and Nenghai Yu. 2025. CompileAgent: Automated Real-World Repo-Level Compilation with Tool-Integrated LLM-based Agent System. arXiv:2505.04254 [cs.SE] https://arxiv.org/abs/2505.04254

[17] Yiwei Hu, Zhen Li, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. 2025. SoK: Automated Vulnerability Repair: Methods, Tools, and Assessments. arXiv:2506.11697 [cs.SE] https://arxiv.org/abs/2506.11697

[18] 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 Computer Society, Los Alamitos, CA, USA, 539–554. https://doi.org/10.1109/SP.2019.00071

[19] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 955–963. https://doi.org/10.1145/3338906.3340459

[20] JasPer. 2024. JasPer: Image Processing Library. https://jasper-software.github.io/jasper/.

[21] jqlang. 2024. jq: Command-line JSON Processor. https://jqlang.github.io/jq/.

[22] KidFlo. 2024. range(9.9, '0') causes segmentation fault. https://github.com/php/php-src/issues/13094.

[23] 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 (USENIX Security 25)*. USENIX Association, Seattle, WA, 4401–4419.

[24] 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* (Porto de Galinhas, Brazil) *(AIware 2024)*. Association for Computing Machinery, New York, NY, USA, 103–111. https://doi.org/10.1145/3664646.3664770

[25] Wei Le and Shannon D. Pattison. 2014. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1047–1058. https://doi.org/10.1145/2568225.2568304

[26] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 524–535. https://doi.org/10.1109/ICSE.2019.00064

[27] Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D. Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. 2023. Invalidator: Automated Patch Correctness Assessment Via Semantic and Syntactic Reasoning. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3411–3429. https://doi.org/10.1109/TSE.2023.3255177

[28] Ying Li, Faysal hossain shezan, Bomin wei, Gang Wang, and Yuan Tian. 2025. SoK: Towards Effective Automated Vulnerability Repair. arXiv:2501.18820 [cs.CR] https://arxiv.org/abs/2501.18820

[29] Zhaoyang Li, Zheng Yu, Jingyi Song, Meng Xu, Yuxuan Luo, and Dongliang Mu. 2026. PORTGPT: Towards Automated Backporting Using Large Language Models . In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 643–662. https://doi.org/10.1109/SP63933.2026.00034

[30] LibTIFF Contributors. 2024. LibTIFF: TIFF Library and Utilities. http://libtiff.org/.

[31] LLVM Project. 2024. LLVM: The LLVM Compiler Infrastructure. https://llvm.org/.

[32] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. *ACM SIGPLAN Notices* 47, 10 (2012), 133–146.

[33] Alex Mathai, Chenxi Huang, Suwei Ma, Jihwan Kim, Hailie Mitchell, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. 2025. CrashFixer: A crash resolution agent for the Linux kernel. arXiv:2504.20412 [cs.SE] https://arxiv.org/abs/2504.20412

[34] 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* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[35] Qing Mi and Jacky Keung. 2016. An empirical analysis of reopened bugs based on open source projects. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (Limerick, Ireland) *(EASE '16)*. Association for Computing Machinery, New York, NY, USA, Article 37, 10 pages. https://doi.org/10.1145/2915970.2915986

[36] Bram Moolenaar. 2024. Vim: The Ubiquitous Text Editor. https://www.vim.org/.

[37] mruby Contributors. 2024. mruby: Lightweight Ruby Implementation. https://mruby.org/.

[38] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781.

[39] Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. 2019. Automatic Program Repair Using Formal Verification and Expression Templates. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 70–91.

[40] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching. arXiv:2408.13597 [cs.CR] https://arxiv.org/abs/2408.13597

[41] OpenAI. 2024. OpenAI API Models Documentation. https://platform.openai.com/docs/models.

[42] Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 440–452. https://doi.org/10.1145/3650212.3652140

[43] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[44] PcapPlusPlus Contributors. 2024. PcapPlusPlus: Network Packet Processing Library. https://pcapplusplus.github.io/.

[45] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large

Language Models . In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2339–2356. https://doi.org/10.1109/SP46215.2023.10179420

[46] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 427–449. https://doi.org/10.1109/tse.2014.2312918

[47] Justyna Petke, Matias Martinez, Maria Kechagia, Aldeida Aleti, and Federica Sarro. 2024. The Patch Overfitting Problem in Automated Program Repair: Practical Magnitude and a Baseline for Realistic Benchmarking. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 452–456. https://doi.org/10.1145/3663529.3663776

[48] PHP Documentation Group. 2024. PHP:range Manual. https://www.php.net/manual/en/function.range.php.

[49] PHP Group. 2024. PHP: Hypertext Preprocessor. https://www.php.net/.

[50] Python Software Foundation. 2024. ast — Abstract Syntax Trees. https://docs.python.org/3/library/ast.html.

[51] Python Software Foundation. 2024. CPython: The Reference Implementation of Python. https://github.com/python/cpython.

[52] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. arXiv:1606.05250 [cs.CL] https://arxiv.org/abs/1606.05250

[53] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[54] Bat-Chen Rothenberg and Orna Grumberg. 2016. Sound and Complete Mutation-Based Program Repair. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 593–611.

[55] Samtools Contributors. 2024. HTSlib: High-Throughput Sequencing Data Library. https://www.htslib.org/.

[56] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated patch backporting in Linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 633–645. https://doi.org/10.1145/3460319.3464821

[57] 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 390–405. https://doi.org/10.1145/3453483.3454051

[58] 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 (2025), 1–27.

[59] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1993–2010. https://www.usenix.org/conference/usenixsecurity22/presentation/shi

[60] simdjson. 2024. simdjson: Parsing Gigabytes of JSON per Second. https://simdjson.org/.

[61] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[62] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1275–1295. https://doi.org/10.1109/SP.2019.00010

[63] The HDF Group. 2024. HDF5: High-Performance Data Management Suite. https://www.hdfgroup.org/solutions/hdf5/.

[64] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 862–880. https://doi.org/10.1109/SP54263.2024.00210

[65] Unicode Consortium. 2024. ICU: International Components for Unicode. https://icu.unicode.org/.

[66] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2019. How Different Is It Between Machine-Generated and Developer-Provided Patches? An Empirical Study on The Correct Patches Generated by Automated Program Repair Techniques. arXiv:1906.03447 [cs.SE] https://arxiv.org/abs/1906.03447

[67] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2021. Automated patch correctness assessment: how far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3324884.3416590

[68] WebAssembly. 2024. WABT: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt.

[69] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE '13)*. IEEE Press, Piscataway, NJ, USA, 356–366. https://doi.org/10.1109/ASE.2013.6693094

[70] Wireshark Foundation. 2024. Wireshark: Network Protocol Analyzer. https://www.wireshark.org/.

[71] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. 2023. Mitigating Security Risks in Linux with KLAUS: A Method for Evaluating Patch Correctness. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4247–4264. https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yuhang

[72] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, Piscataway, NJ, USA, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[73] Yunlong Xing, Shu Wang, Shiyu Sun, Xu He, Kun Sun, and Qi Li. 2024. What IF Is Not Enough? Fixing Null Pointer Dereference With Contextual Check. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 1367–1382. https://www.usenix.org/conference/usenixsecurity24/presentation/xing-yunlong

[74] 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* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 789–799. https://doi.org/10.1145/3180155.3180182

[75] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 512–523. https://doi.org/10.1109/ICSE.2019.00063

[76] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] https://arxiv.org/abs/2405.15793

[77] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling Data for Software Engineering Agents. arXiv:2504.21798 [cs.SE] https://arxiv.org/abs/2504.21798

[78] Su Yang, Yang Xiao, Zhengzi Xu, Chengyi Sun, Chen Ji, and Yuqing Zhang. 2023. Enhancing OSS Patch Backporting with Semantics. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2366–2380. https://doi.org/10.1145/3576915.3623188

[79] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26 (2021), 1–38.

[80] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. 2025. PatchAgent: A Practical Program Repair Agent Mimicking Human Expertise. In *34rd USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 4381–4400.

[81] Zheng Yu, Ganxiang Yang, and Xinyu Xing. 2024. ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 7177–7193. https://www.usenix.org/conference/usenixsecurity24/presentation/yu-zheng

[82] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. 2025. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. arXiv:2504.02605 [cs.SE] https://arxiv.org/abs/2504.02605

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

[84] Yuntong Zhang, Jiawei Wang, Dominic Berzin, Martin Mirchev, Dongge Liu, Abhishek Arya, Oliver Chang, and Abhik Roychoudhury. 2024. Fixing Security Vulnerabilities with AI in OSS-Fuzz. arXiv:2411.03346 [cs.CR] https://arxiv.org/

abs/2411.03346

[85] Han Zheng, Ilia Shumailov, Tianqi Fan, Aiden Hall, and Mathias Payer. 2025. Fixing 7,400 Bugs for 1$: Cheap Crash-Site Program Repair. https://arxiv.org/abs/2505.13103 arXiv:2505.13103 [cs.SE]

[86] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 88, 13 pages. https://doi.org/10.1145/3597503.3639222

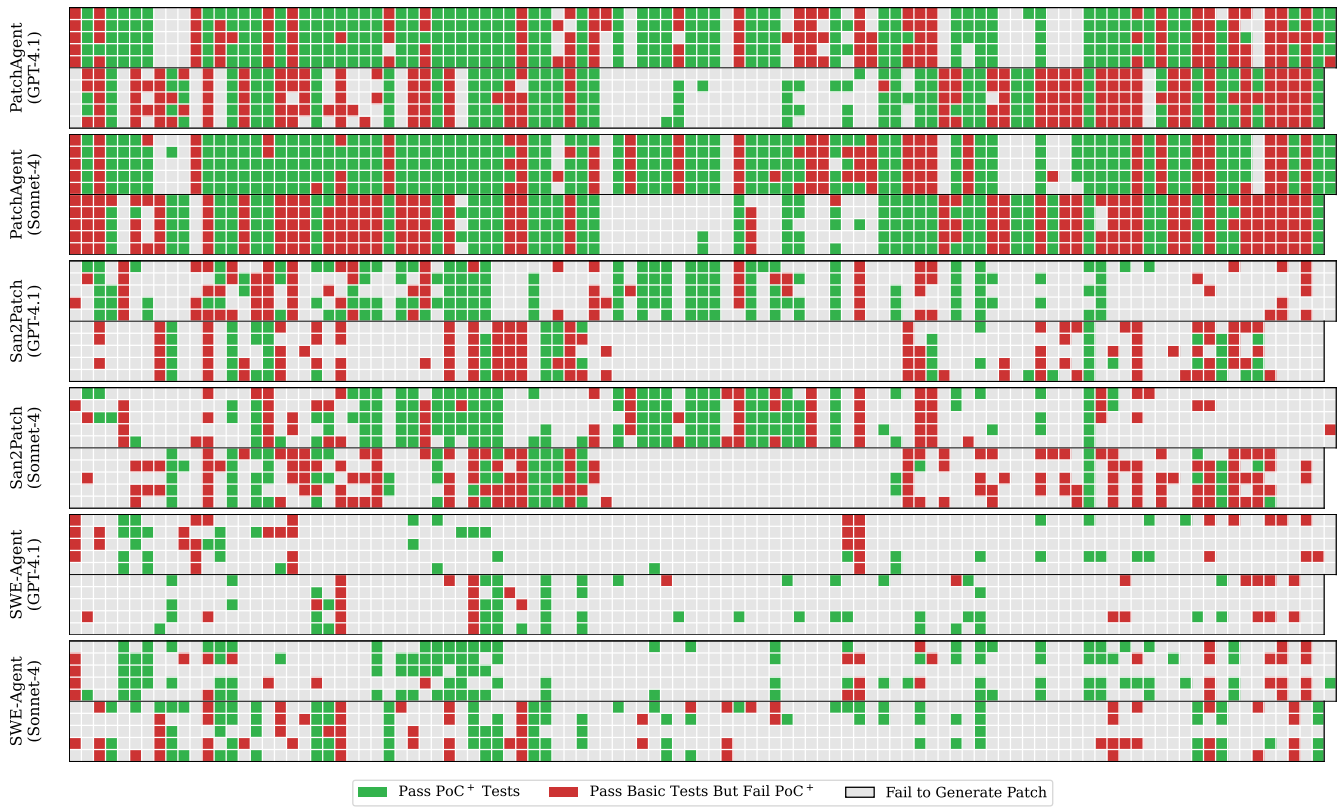

Figure 13: Comprehensive evaluation results across 209 vulnerability cases.

A APPENDIX

Table 7 presents the paths to automated test generation scripts for projects that natively support PoC$^+$ generation, as evaluated in the *Output Checking* category. Figure 13 presents the comprehensive results of our evaluation across all 209 vulnerability cases. The visualization employs a grid-based format where each agent-model combination is represented by a 10×105 matrix. Due to the odd number of vulnerability cases (209), the data is split into two unequal halves: the top 5 rows display results for vulnerabilities 1-105, while the bottom 5 rows show results for vulnerabilities 106-209. Each column in the grid corresponds to a single vulnerability case, with 5 cells per column representing independent experimental runs. The color coding follows our evaluation criteria: green cells indicate successful patches that pass both basic functionality tests and PoC$^+$ tests, red cells represent patches that pass basic tests but fail the more stringent PoC$^+$ requirements, and gray cells denote cases where the agent failed to generate any viable patch.

B OPEN SCIENCE.

To support the evaluation of this paper's contributions and facilitate reproducibility, we provide all artifacts at https://anonymous.4open.science/r/PVBench-91BC. The repository contains the PVBench dataset and the original experimental data from evaluating three state-of-the-art agents on PVBench.

Table 7: Script Path of Projects that Support PoC$^+$ Generation

Project	Automated Test Generation Scripts Path
Hermes	`utils/updateErrorTest.py`
libxml2	`codegen/genTestApi.py`
	`xstc/fixup-tests.py`
LLVM	`llvm/utils/update_analyze_test_checks.py`
	`llvm/utils/update_any_test_checks.py`
	`llvm/utils/update_cc_test_checks.py`
	`llvm/utils/update_llc_test_checks.py`
	`llvm/utils/update_mca_test_checks.py`
	`llvm/utils/update_mir_test_checks.py`
	`llvm/utils/update_test_checks.py`
	`llvm/utils/update_test_prefix.py`
PHP	`scripts/dev/bless_tests.php`
Wabt	`test/run-tests.py`
	`test/update-spec-tests.py`