Pre-print – Extended version of the poster paper accepted at the 41st ACM/SIGAPP Symposium on Applied Computing (SAC) Smarter Engineering - Building AI and Building with AI (SEAI) 2026

Evaluating LLMs for One-Shot Patching of Real and Artificial Vulnerabilities

Aayush Garg
aayush.garg@list.lu
Luxembourg Institute of Science and Technology
Luxembourg

Zanis Ali Khan
zanis-ali.khan@list.lu
Luxembourg Institute of Science and Technology
Luxembourg

Renzo Degiovanni
renzo.degiovanni@list.lu
Luxembourg Institute of Science and Technology
Luxembourg

Qiang Tang
qiang.tang@list.lu
Luxembourg Institute of Science and Technology
Luxembourg

ABSTRACT

Automated vulnerability patching is crucial for software security, and recent advancements in Large Language Models (LLMs) present promising capabilities for automating this task. However, existing research has primarily assessed LLMs using publicly disclosed vulnerabilities, leaving their effectiveness on related artificial vulnerabilities largely unexplored. In this study, we empirically evaluate the patching effectiveness and complementarity of several prominent LLMs, such as OpenAI's GPT variants, LLaMA, DeepSeek, and Mistral models, using both real and artificial vulnerabilities. Our evaluation employs Proof-of-Vulnerability (PoV) test execution to concretely assess whether LLM-generated source code successfully patches vulnerabilities. Our results reveal that LLMs patch real vulnerabilities more effectively compared to artificial ones. Additionally, our analysis reveals significant variability across LLMs in terms of overlapping (multiple LLMs patching the same vulnerabilities) and complementarity (vulnerabilities patched exclusively by a single LLM), emphasizing the importance of selecting appropriate LLMs for effective vulnerability patching.

1 INTRODUCTION

Automated vulnerability patching has become increasingly significant in software security due to the escalating number and complexity of software vulnerabilities discovered annually. Recently, **Large Language Models *(LLMs)*** including ***OpenAI***'s ***GPT*** variants, ***LLaMA***, ***DeepSeek***, and ***Mistral***, have emerged as promising tools for automating vulnerability patching. These models have shown notable capability in generating syntactically and semantically coherent software patches [10, 43]. However, the effectiveness of these models in patching vulnerabilities, particularly across diverse vulnerability types and their variants, remains understudied.

Previous studies evaluating LLM-based patching have primarily focused on known, i.e., publicly disclosed vulnerabilities sourced from public vulnerability databases [2, 21, 32] or real-world software repositories [23, 24, 36]. These evaluations typically consider syntactic correctness or employ similarity-based metrics such as *CodeBLEU* [38] and *CrystalBLEU* [8]. While these metrics provide insights into patch quality, they do not directly indicate whether the generated patches **effectively eliminate** vulnerabilities [25, 29, 30].

Moreover, a significant limitation in existing literature is the scarcity of research investigating whether LLMs can generalize their vulnerability patching capabilities beyond these known vulnerabilities [41]. *Garg et al.*[15] recently augmented vulnerability dataset with their generated **artificial vulnerabilities**. We considered these artificial cases in our study to augment our evaluation and to test LLMs' ability to patch both real and artificial vulnerabilities. This enabled us to perform an assessment of the generalizability and robustness of LLM-based vulnerability patching [46].

Our paper empirically investigates the effectiveness and complementarity of several prominent LLMs in automated vulnerability patching, specifically focusing on both real vulnerabilities and their corresponding artificial vulnerabilities. We structure our investigation around two research questions (RQs):

> **RQ1: How effective are LLMs in patching real vulnerabilities vs. artificial vulnerabilities?**

RQ1 investigates the extent to which LLMs can successfully patch real vulnerabilities compared to their artificial counterparts, quantitatively comparing their effectiveness across these vulnerability categories.

> **RQ2: How complementary and overlapping are LLMs in patching real vulnerabilities vs. artificial vulnerabilities?**

RQ2 explores how frequently multiple LLMs successfully patch the same vulnerabilities (**overlapping**), and how often vulnerabilities are successfully patched exclusively by a single LLM (**complementary**). By evaluating this distribution across real and artificial vulnerabilities, we assess whether different LLMs consistently patch the same vulnerabilities or contribute uniquely by patching distinct vulnerabilities.

To perform this evaluation, we employed **15** real vulnerabilities and their **41** artificial counterparts (vulnerabilities). We applied a uniform prompting strategy across all evaluated LLMs to generate patches. Evaluation of these patches was performed using PoV test executions, ensuring rigorous and concrete validation of the effectiveness of the generated patches.

Our contributions are summarized as follows:

(1) A systematic evaluation of the effectiveness of prominent LLMs in patching real and artificial vulnerabilities, measured using Proof-of-Vulnerability (PoV) test execution.

(2) An analysis of the extent to which LLM-generated patches overlap across models or complement each other, identifying cases where specific vulnerabilities are patched either by multiple LLMs or uniquely by individual LLMs.

2 BACKGROUND

In this section, we present the necessary background, including prior works on LLM-based vulnerability patching.

2.1 Large Language Model *(LLM)*-based Automated Vulnerability Patching

Traditional vulnerability-patching methods in industry often rely on static analysis tools and manual expert fixes. Static analyzers can flag common weaknesses and even suggest remedial code patterns (e.g., adding input validations), though these suggestions are typically limited to well-known bug patterns [17]. General automated program repair tools have also been evaluated on security bugs, but their success is modest [9]. For instance, Bui et al. [1] found that state-of-the-art automated vulnerability patching tools could produce testable patches for only about **20%** of real-world vulnerabilities, and under half of those patches (~44%) actually fixed the issue without breaking functionality. This amounts to less than 10% of vulnerabilities being fully remedied by traditional APR techniques. In addition, domain-specific patch generation has seen some success in narrow contexts, e.g., automatic insertion of security checks to fix Android component-hijacking flaws, effectively patching such vulnerabilities in multiple real-world apps [45]. These targeted solutions, however, are limited to specific vulnerability types.

LLMs have demonstrated significant potential in program repair, particularly in automated vulnerability patching [10, 37, 43]. Given a vulnerable code snippet, these models generate a patched version that aims to eliminate security flaws while preserving functionality. Prior research has evaluated LLMs' ability to patch vulnerabilities based on datasets of publicly disclosed vulnerabilities [32, 36]. However, LLM-generated patches often vary in effectiveness, requiring validation through test cases to confirm whether the vulnerability has been successfully addressed [25, 30, 33].

Despite promising results, LLM-based patching faces challenges. First, LLMs may **overfit to known vulnerabilities**, producing patches that resemble existing solutions rather than addressing the underlying security flaw [10]. Second, **syntactic correctness does not guarantee functional correctness**, making execution-based evaluation crucial [5, 34, 35]. Lastly, the generalization of LLMs to variations of real vulnerabilities remains an open question [47], which this study seeks to investigate.

2.2 Artificial Vulnerabilities

Artificial vulnerabilities are intentionally introduced flaws in a system, such as source code, designed to mimic real security weaknesses [14, 15, 26]. These controlled weaknesses allow researchers and developers to **assess and refine detection and mitigation** strategies in a reproducible environment [12, 13, 44].

Several methodologies exist for **creating** artificial vulnerabilities. **Code injection** introduces malicious code to simulate attacks like SQL injection or cross-site scripting *(XSS)*, helping assess defensive measures [18]. Fault injection deliberately introduces errors to test system resilience and error-handling capabilities [19]. **Mutation testing**, on the other hand, modifies source code systematically to replicate common programming mistakes, evaluating the effectiveness of testing strategies [16, 20, 22]. For example, the *LAVA-M* [7] dataset systematically injects synthetic memory-safety bugs into C programs to create ground-truth vulnerability cases. These techniques collectively support the development of more secure and robust software systems.

2.3 Evaluation of LLM-Based Patching

Existing studies primarily evaluate LLM-based code generation using **code similarity metrics** such as ***CodeBLEU*** [38] and ***CrystalBLEU*** [8], which assess the syntactic and semantic resemblance between the generated code and the expected. However, these metrics may not accurately reflect the security efficacy of patches, as they do not account for the actual elimination of vulnerabilities [31].

To ensure that patches genuinely address security flaws, **test execution validation** is necessary [5, 6, 41]. This approach involves running **Proof-of-Vulnerability (*PoV*)** test cases against the patched code to verify that the specific vulnerabilities have been effectively mitigated. Such execution-based evaluations provide a more reliable assessment of the patch's effectiveness in real-world scenarios [27, 39]. *Wang et al.* [41] also demonstrated that without rigorous exploit-driven tests, an LLM-generated patch may give a **false sense of safety** where it can appear to fix the bug while the underlying vulnerability remains exploitable. Hence, in this study, we prioritize the *PoV* test results over similarity-based evaluations to assess the effectiveness of LLM-generated patches.

3 METHODOLOGY

This section describes our approach for evaluating LLM patching effectiveness, detailing the dataset, models, prompting strategy, and evaluation metrics.

3.1 Dataset and Vulnerabilities

Firstly, we utilized the ***Vul4J*** dataset [2], a carefully-curated benchmark of **reproducible** Java vulnerabilities drawn from open-source projects and covering distinct Common Weakness Enumeration (CWE) classes. For every entry Vul4J provides (i) the vulnerable revision (V_{vul}), (ii) the corresponding fixed revision (V_{fix}), (iii) the minimal human-authored patch, and crucially (iv) one or more **Proof-of-Vulnerability *(PoV)*** JUnit test cases. A PoV test is an executable exploit oracle: it exercises the buggy code so that the test **fails** on V_{vul} (e.g., by throwing an exception, timing out, or violating an assertion) and **passes** on V_{fix}. This PoV-test fail-to-pass flip allows automated repair systems to verify, with a single test run, whether a candidate patch truly eliminates the vulnerability. When no suitable test existed in the original project, *Bui et al.* [2] manually wrote one by following the exploit steps in the CVE report, thereby guaranteeing that every vulnerability in the dataset can be triggered, reproduced, and validated in a *Maven/Gradle* build.

Secondly, we augmented our evaluation with **artificially** generated vulnerabilities to broaden our test criteria. By including the artificial cases, we wanted to test LLMs' ability to patch both real and, artificial yet representative, vulnerabilities. Additionally,

```
private static void decompress
 (final InputStream in, final byte[] out)
 throws IOException {
  int position = 0;
  final int total = out.length;
  while (position < total) {
   final int n = in.read();



   if (n > 128) {
    final int value = in.read();
    for (int i = 0; i < (n & 0x7f); i++) {
     out[position++] = (byte) value; }
   } else {
    for (int i = 0; i < n; i++) {
    out[position++] = (byte) in.read();}
   }
  }
 }
```

(a) Real Vulnerability (CVE-2018-17201)

```
private static void decompress
 (final InputStream in, final byte[] out)
 throws IOException {
  int position = 0;
  final int total = out.length;
  while (position < total) {
   final int n = in.read();
   if (n < 0) {
    throw new ImageReadException("Error decompressing
          RGBE file"); }
   if (n > 128) {
    final int value = in.read();
    for (int i = 0; i < (n & 0x7f); i++) {
     out[position++] = (byte) value; }
   } else {
    for (int i = 0; i < n; i++) {
    out[position++] = (byte) in.read();}
   }
  }
 }
```

(b) Patched source code

```
private static void decompress
 (final InputStream in, final byte[] out)
 throws IOException {
  int position = 0;
  final int total = out.length;
  while (position < total) {
   final int n = in.read();
   if (n == 0) { // `<' modified to `=='
    throw new ImageReadException("Error decompressing
          RGBE file"); }
   if (n > 128) {
    final int value = in.read();
    for (int i = 0; i < (n & 0x7f); i++) {
     out[position++] = (byte) value; }
   } else {
    for (int i = 0; i < n; i++) {
    out[position++] = (byte) in.read();}
   }
  }
 }
```

(c) Artificial Vulnerability

Figure 1: Vulnerability CVE-2018-17201 (Fig. 1a) causes an infinite loop that can make the program hang, enabling a Denial-of-Service (DoS) attack. This issue is addressed by introducing a conditional exception using an "*if*" statement (Fig. 1b). However, the code-modification (Fig. 1c) alters the "*if*" condition, effectively nullifying the fix and reintroduces the vulnerability.

```
void addPathParam(String name, String value, boolean
       encoded) {
 if (relativeUrl == null) {
  throw new AssertionError(); }



 relativeUrl = relativeUrl.replace("{" + name + "}",
        canonicalizeForPath(value, encoded));


}
```

(a) Real Vulnerability (CVE-2018-1000850)

```
void addPathParam(String name, String value, boolean
       encoded) {
 if (relativeUrl == null) {
  throw new AssertionError(); }
 String replacement = canonicalizeForPath(value,
        encoded);
 String newRelativeUrl = relativeUrl.replace("{" +
        name + "}", replacement);
 if (PATH_TRAVERSAL
  .matcher(newRelativeUrl)
  .matches()) {
   throw new IllegalArgumentException(
   "@Path parameters shouldn't perform path traversal
         (`.' or `..'): " + value); }
 relativeUrl = newRelativeUrl;
}
```

(b) Patched source code

```
void addPathParam(String name, String value, boolean
       encoded) {
 if (relativeUrl == null) {
  throw new AssertionError(); }
 String replacement = canonicalizeForPath(value,
        encoded);
 String newRelativeUrl = relativeUrl.replace("{" +
        name + "}", replacement);
 if (PATH_TRAVERSAL
  .matcher(name)//passed argument changed
  .matches()) {
   throw new IllegalArgumentException(
   "@Path parameters shouldn't perform path traversal
         (`.' or `..'): " + value); }
 relativeUrl = newRelativeUrl;
}
```

(c) Artificial Vulnerability

Figure 2: Vulnerability CVE-2018-1000850 (Fig. 2a) enables path traversal and access to restricted directories. The patch adds a conditional check for '.' or '..' in "*newRelativeUrl*" (Fig. 2b). The code-modification (Fig. 2c) replaces "*newRelativeUrl*" with "*name*", nullifying the fix and reintroducing the vulnerability.

we wanted to observe whether LLMs handle them differently. To do so, we incorporated artificial vulnerabilities derived from the work of *Garg et al.*[15]. *Garg et al.* used ***CodeBERT***[11] to generate thousands of candidate artificial vulnerabilities and validated them with Vul4J's PoV tests. Only about 4% of the candidates reproduced the **same** PoV failures (including identical exception messages) observed for real Vul4J vulnerabilities. Those validated cases cover **15** real CVEs, i.e., roughly 55% of Vul4J's vulnerabilities.

To illustrate, **Figures 1 and 2** provide motivating examples of artificial and real vulnerabilities, originally presented by *Garg et al.*[15]. Figure 1 shows the code-modification that nullifies a patch for a high-severity (**CVSS 7.5**) infinite loop vulnerability (***CVE-2018-17201*** [4]) by altering a conditional check, effectively reintroducing the real vulnerability. Similarly, Figure 2 demonstrates the code-modification that reintroduces a directory traversal vulnerability (***CVE-2018-1000850*** [3]) by modifying an argument in a validation function.

Please **note** that we only considered these 4% artificial vulnerabilities from *Garg et al.*'s dataset that exhibit the **same** failure patterns as the real vulnerabilities. *Vul4J v1.1* currently contains 27 reproducible Java vulnerabilities; the 15 we study are those for which validated artificial vulnerabilities exist. This resulted in our dataset of **15** real vulnerabilities (from ***Vul4J***) and their corresponding **41** artificial vulnerabilities (from *Garg et al.*), for which we present the details (i.e., vulnerability's CVE ID, CWE details, severity, failed PoV tests, and the number of corresponding artificial cases) in **Table 1**.

3.2 LLMs Evaluated

We evaluate the patching capabilities of **14** large language models *(LLMs)* spanning different architectures and sizes. These include models from *OpenAI*'s GPT series, *Meta*'s *LLaMA* models, *DeepSeek*, and *Mistral*'s instruction-tuned variants. Below are the LLMs that we assess in our study.

(1) **DeepSeekR1 Qwen 32B**
(2) **GPT3.5 Turbo**
(3) **GPT3.5 Turbo 1106**
(4) **GPT3.5 Turbo Instruct**
(5) **GPT4**
(6) **GPT4 0613**
(7) **GPT4 Turbo**
(8) **GPT4o**
(9) **GPT4o Mini**
(10) **LLaMA 3.1 70B Instruct**
(11) **LLaMA 3.3 70B Instruct**
(12) **Mistral 7B v0.2 Instruct**
(13) **Mistral 7B v0.3 Instruct**
(14) **Mistral 8×7B v0.1 Instruct**

Table 1: Overview of real and artificial vulnerabilities

Real Vulnerability (CVE ID)	Vulnerability Class (CWE ID)	CWE Description (Vulnerability Cause)	Severity (0 – 10)	Failed Tests (PoV)	Artificial Vulnerabilities (#)
APACHE-COMMONS-001	CWE-noinfo	No information provided by NIST	NA	1	1
CVE-2013-5960	CWE-310	Cryptographic Issues	5.8	15	1
CVE-2014-4172	CWE-74	Improper neutralization of special elements in output used by a downstream component (‘Injection’)	9.8	1	10
CVE-2016-10006	CWE-79	Improper neutralization of input during web page generation (‘Cross-site scripting’)	6.1	1	1
CVE-2016-2162	CWE-79	Improper neutralization of input during web page generation (‘Cross-site scripting’)	6.1	1	1
CVE-2016-6802	CWE-284	Improper access control	7.5	3	1
CVE-2017-5662	CWE-611	Improper restriction of xml external entity reference	7.3	1	6
CVE-2018-1000089	CWE-532	Insertion of sensitive information into log file	7.4	1	7
CVE-2018-1000531	CWE-20	Improper input validation	7.5	1	1
CVE-2018-1000850	CWE-22	Improper limitation of a pathname to a restricted directory (‘Path traversal’)	7.5	3	1
CVE-2018-1000854	CWE-74	Improper neutralization of special elements in output used by a downstream component (‘Injection’)	9.8	1	1
CVE-2018-11771	CWE-835	Loop with unreachable exit condition (‘Infinite loop’)	5.5	2	2
CVE-2018-17201	CWE-835	Loop with unreachable exit condition (‘Infinite loop’)	7.5	1	2
CVE-2019-12402	CWE-835	Loop with unreachable exit condition (‘Infinite loop’)	7.5	1	1
HTTPCLIENT-1803	CWE-noinfo	No information provided by NIST	NA	1	5
Total Real Vulnerabilities				**15**	
Total Artificial Vulnerabilities				**41**	

These **14** LLMs were selected to cover a range of training paradigms, instruction-tuned capabilities, and underlying architectures, allowing for a comprehensive analysis of their effectiveness in vulnerability patching.

3.3 Prompting Strategy

To ensure a consistent evaluation across all models, we employed a standardized prompt designed to elicit vulnerability patches while minimizing extraneous modifications:

> `"You are a security expert who is good at static program analysis. The following SOURCE CODE, written in Java, contains a vulnerability. Please write the source code to fix this vulnerability. Do not make any other changes to the source code, and just reply with the final patched Java function. SOURCE CODE:` {source code}"

We **deliberately avoid advanced prompting/decoding** such as *self-consistency* [42] and *iterative self-feedback* (*Reflexion* [40], *Self-Refine* [28]), and evaluate each model in a **one-shot, prompt-only** setting to ensure comparability across models. Each LLM received only the vulnerable function's Java source code. No surrounding files, build scripts, or PoV tests were supplied. This prompt explicitly instructs the LLM to focus solely on patching the vulnerability without altering unrelated code segments. Each LLM was prompted separately for every vulnerability, ensuring that the responses were generated independently. The collected outputs were then assessed for their patching success across corresponding vulnerabilities.

3.4 Evaluation Process

The patches generated by the LLMs were compiled and executed against the Proof-of-Vulnerability (PoV) test cases. For every vulnerability, we programmatically replaced the original function with the model-generated version (patch), rebuilt the project, and executed the PoV suite. A patch was considered **successful** if it successfully compiled and passed all associated PoV tests. If a model's output resulted in compilation errors or failed the PoV test cases, it was considered unsuccessful.

3.5 Evaluation Metrics

To assess the patching capabilities of LLMs, we employ the following key metrics:

(1) **Patching Success**: The percentage of vulnerabilities successfully patched by each model, measured separately for real and artificial vulnerabilities, enabling a direct comparison between the two categories.
(2) **Overlapping Patching**: The proportion of vulnerabilities that were patched by multiple models, indicating common success cases.
(3) **Complementary Patching**: The proportion of vulnerabilities patched exclusively by a single model, highlighting the complementary contributions to patching.

These metrics directly address our research questions, providing insights into the effectiveness and complementarity of LLMs in patching both real and artificial vulnerabilities.

Table 2: Overview of LLMs in patching real and artificial vulnerabilities

Vuln.	DeepSeek R1 Qwen 32B	GPT 3.5 Turbo	GPT 3.5 Turbo 1106	GPT 3.5 Turbo Instruct	GPT 4	GPT 4 0613	GPT 4 Turbo	GPT 4o	GPT 4o Mini	LLaMA 3.1 70B Instruct	LLaMA 3.3 70B Instruct	Mistral 7B v0.2 Instruct	Mistral 7B v0.3 Instruct	Mistral 8x7B v0.1 Instruct
APACHE-COMMONS-001														
real														
artificial														
CVE-2013-5960														
real	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
artificial	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVE-2014-4172														
real														
artificial 01														
artificial 02														
artificial 03														
artificial 04														
artificial 05														
artificial 06														
artificial 07														
artificial 08														
artificial 09														
artificial 10														
CVE-2016-10006														
real														
artificial						✓		✓					✓	✓
CVE-2016-2162														
real														
artificial	✓												✓	
CVE-2016-6802														
real					✓		✓					✓		✓
artificial					✓		✓							✓
CVE-2017-5662														
real														
artificial 01														
artificial 02														
artificial 03														
artificial 04														
artificial 05														
artificial 06														
CVE-2018-1000089														
real														
artificial 01														
artificial 02														
artificial 03														
artificial 04														
artificial 05														
artificial 06														
artificial 07														
CVE-2018-1000531														
real	✓			✓	✓	✓		✓		✓	✓	✓	✓	✓
artificial														
CVE-2018-1000850														
real	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
artificial														
CVE-2018-1000854														
real	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
artificial														
CVE-2018-11771														
real													✓	
artificial 1														
artificial 2														
CVE-2018-17201														
real	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓		✓
artificial 1	✓	✓	✓			✓						✓		✓
artificial 2														
CVE-2019-12402														
real														
artificial														
HTTPCLIENT-1803														
real	✓		✓											✓
artificial 1	✓				✓				✓					✓
artificial 2	✓						✓			✓				✓
artificial 3	✓						✓			✓				
artificial 4	✓													✓
artificial 5	✓													

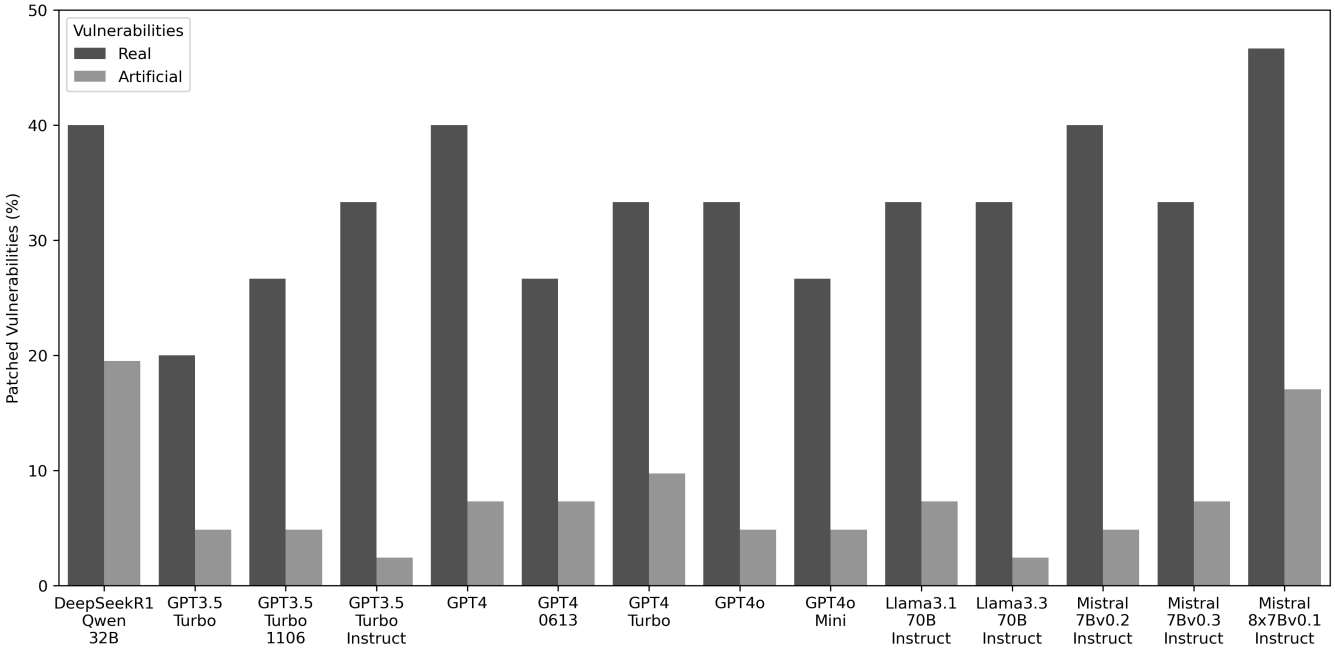

Figure 3: Patching effectiveness of LLMs for real and artificial vulnerabilities.

4 RESULTS

Table 2 provides an overview of different LLMs patching real and artificial vulnerabilities. Each section is grouped by CVE ID, covering both real and artificial vulnerabilities. The first column specifies the vulnerability type, distinguishing between real and artificial cases. The remaining columns represent different LLMs, where a checkmark (✓) indicates successful patching of the vulnerability by that model, while an empty cell denotes failure to patch, either due to a compilation error or failed test cases (Proof-of-Vulnerability). Moreover, the table enables a structured representation of how different models overlap or complement each other in their patching capabilities. The table reveals **three** trends:

(1) **Real vs. artificial.** Check-marks are still skewed toward the **real** CVEs, i.e., more than half (8/15, ∼53%) of the real vulnerabilities receive at least one patch, whereas fewer than one-quarter (10/41, ∼24%) of the artificial vulnerabilities are patched. Entire blocks of artificial cases such as all **ten** variations of *CVE-2014-4172* and **seven** of *CVE-2018-1000089*, remain completely blank in the table, conveying LLMs' **failure to patch** such cases.
(2) **LLM spread.** As per our results, no single vendor dominated. The best overall scores were achieved *DeepSeekR1 Qwen 32B* and *Mistral 8x7Bv0.1 Instruct* with 14 successful patches each, followed by the GPT4/Turbo with 9 patches, and LLaMA3.1 70B Instruct and Mistral 7B variants with 8 patches, followed by the rest.
(3) **Overlap.** When a real vulnerability was patched it was usually patched by **several** models, e.g., *CVE-2018-1000531*, *CVE-2018-1000850*, *CVE-2018-1000854* were patched by almost every LLM, while the few artificial vulnerabilities that were patched were usually fixed by only one model (i.e., model overlap is very low for these cases).

Illustrative Examples. We further inspected one success and one failure, for which, we provide the details below:

(1) **CVE-2013-5960 - patching-success.** *OWASP ESAPI* encrypted data with the legacy mode `AES/CBC/PKCS5Padding` and accepted message-authentication tags shorter than 128 bits. An attacker could therefore flip cipher-text bits and tamper with the decrypted message. The official patch makes **two** tiny edits: (i) change the constant that selects the cipher to `AES/GCM/NoPadding`, an authenticated-encryption mode; and (ii) hard-code the tag length check to 128 bits. Every LLM considered in our study reproduced those **exact** two lines. After applying the patch, the *Proof-of-Vulnerability (PoV)* test, which tries to modify a valid cipher-text and then decrypt it, **fails as expected**, confirming the tampering attack is blocked.
(2) **CVE-2017-5662 - patching-failure.** Apache Batik's XML parser resolved external entities by default, so a crafted SVG file could read and embed the contents of an arbitrary local file (e.g. `/etc/passwd`), an *XML external entity injection (XXE)* vulnerability. A **complete** repair needs two coordinated changes: (i) in code, disable external-entity resolution (e.g., call *setFeature(FEATURE_SECURE_PROCESSING, true)*); and (ii) in the build file, upgrade Batik to version ≥ 1.9, because older versions silently ignore that security flag in some pipelines. None of the LLMs produced both edits in the same patch where most toggled the parser flag but left the dependency unchanged. This likely reflects the fact that the required library upgrade resides outside the provided code snippet. When executed, the PoV tests failed (in contrast to passing-PoV-tests for a correct patch) with the malicious SVG still succeeding in printing the local file, hence indicating a failed patch.

Table 3: Overview of patched vulnerabilities by LLMs

Large Language Model *(LLM)*	**Overall Vulnerabilities Patched** *Count # (Percentage %)*	**Real Vulnerabilities Patched** *Count # (Percentage %)*	**Artificial Vulnerabilities Patched** *Count # (Percentage %)*
DeepSeekR1 Qwen 32B	14 (25.00%)	6 (40.00%)	8 (19.51%)
GPT3.5 Turbo	5 (8.93%)	3 (20.00%)	2 (4.88%)
GPT3.5 Turbo 1106	6 (10.71%)	4 (26.67%)	2 (4.88%)
GPT3.5 Turbo Instruct	6 (10.71%)	5 (33.33%)	1 (2.44%)
GPT4	9 (16.07%)	6 (40.00%)	3 (7.32%)
GPT4 0613	7 (12.50%)	4 (26.67%)	3 (7.32%)
GPT4 Turbo	9 (16.07%)	5 (33.33%)	4 (9.76%)
GPT4o	7 (12.50%)	5 (33.33%)	2 (4.88%)
GPT4o Mini	6 (10.71%)	4 (26.67%)	2 (4.88%)
LLaMA3.1 70B Instruct	8 (14.29%)	5 (33.33%)	3 (7.32%)
LLaMA3.3 70B Instruct	6 (10.71%)	5 (33.33%)	1 (2.44%)
Mistral 7Bv0.2 Instruct	8 (14.29%)	6 (40.00%)	2 (4.88%)
Mistral 7Bv0.3 Instruct	8 (14.29%)	5 (33.33%)	3 (7.32%)
Mistral 8x7Bv0.1 Instruct	14 (25.00%)	7 (46.67%)	7 (17.07%)

These examples show that LLMs handle single-location fixes quite well. However, their success rate drops when a vulnerability demands several interdependent edits scattered across the codebase.

RQ1: How effective are LLMs in patching real vulnerabilities vs. artificial vulnerabilities?

To assess how well LLMs patch real vulnerabilities compared to artificial vulnerabilities, we analyzed the proportion of vulnerabilities patched in both categories. **Figure 3** presents a comparative visualization of patching rates for real and artificial vulnerabilities across different models.

Table 3 presents the number and percentage of real and artificial vulnerabilities patched by each model. The results show that while LLMs successfully patch a subset of real vulnerabilities, their effectiveness in patching artificial vulnerabilities is consistently lower across all models, indicating a challenge in addressing these cases. Moreover, we found that, while some models perform better than others, none maintains consistent performance across both categories. All models show a noticeable drop in effectiveness on the artificial vulnerabilities. The gap between their effectiveness in patching real and artificial vulnerabilities suggests that artificial vulnerabilities pose a greater challenge for these models.

RQ2: How complementary and overlapping are LLMs in patching real vulnerabilities vs. artificial vulnerabilities?

To investigate how LLMs complement or overlap in their patching capabilities, we analyze the extent to which multiple models patch the same vulnerabilities vs. cases where a single model patches a vulnerability that no other model does.

Figure 4 presents a greyscale heat-map in which each cell is annotated as X (Y%), where X is the absolute number of vulnerabilities a given model patched in that category and Y is the corresponding share of that model's total patches. The figure presents our drawn observations mentioned below:

(1) **Substantial overlap on real-world vulnerabilities.** All models patch at least three of the fifteen real CVEs in common with another model, and the most overlapping, i.e., *Mistral 7B*, *DeepSeekR1 Qwen 32B*, and *GPT4*, overlap on 6-7 real CVEs (40-46.7% of their respective real-vulnerability fixes). Hence, when a real vulnerability is within reach of current LLMs, several models generally converge on the same repair.
(2) **Marked reduction in overlap on artificial variants.** For the 41 artificial vulnerabilities, the overlap count per model falls to the 1–4 range for most LLMs (2.4%–9.8%), with only *DeepSeekR1 Qwen 32B*, and *Mistral 8x7Bv0.1* reaching seven overlaps each (17%). Agreement among models is therefore far lower on artificial vulnerabilities, confirming the difficulty already highlighted in RQ1.
(3) **Scarcity of truly unique (complementary) patches.** Complementary successes, i.e., cases where a vulnerability is fixed by one model and by no other, are almost absent. Across all 56 vulnerabilities and 14 models, only two such instances appear: a single real vulnerability uniquely patched by Mistral 7Bv0.3 Instruct and one artificial vulnerability uniquely patched by DeepSeekR1 Qwen 32B.
(4) **Implications for ensemble strategies.** The combination of high overlap on real CVEs and negligible complementarity indicates limited marginal benefit from ensembling the considered LLMs in our study. The results indicate that most gains would come from whichever single model already attains the highest individual patching performance, rather than from additive coverage.

Collectively, these findings indicate that contemporary LLMs tend to **pile up** on the same and relatively tractable known vulnerabilities, and mostly **fail** to patch the artificial ones, with a seldom exception of unique patches that other LLMs miss.

5 THREATS TO VALIDITY

Construct Validity. The threat to construct validity lies in the measurement of patch effectiveness. We used *PoV* test cases provided by Vul4J[2], which may not comprehensively capture all security implications of the vulnerabilities. As a result, passing PoV tests does not necessarily imply complete vulnerability mitigation.

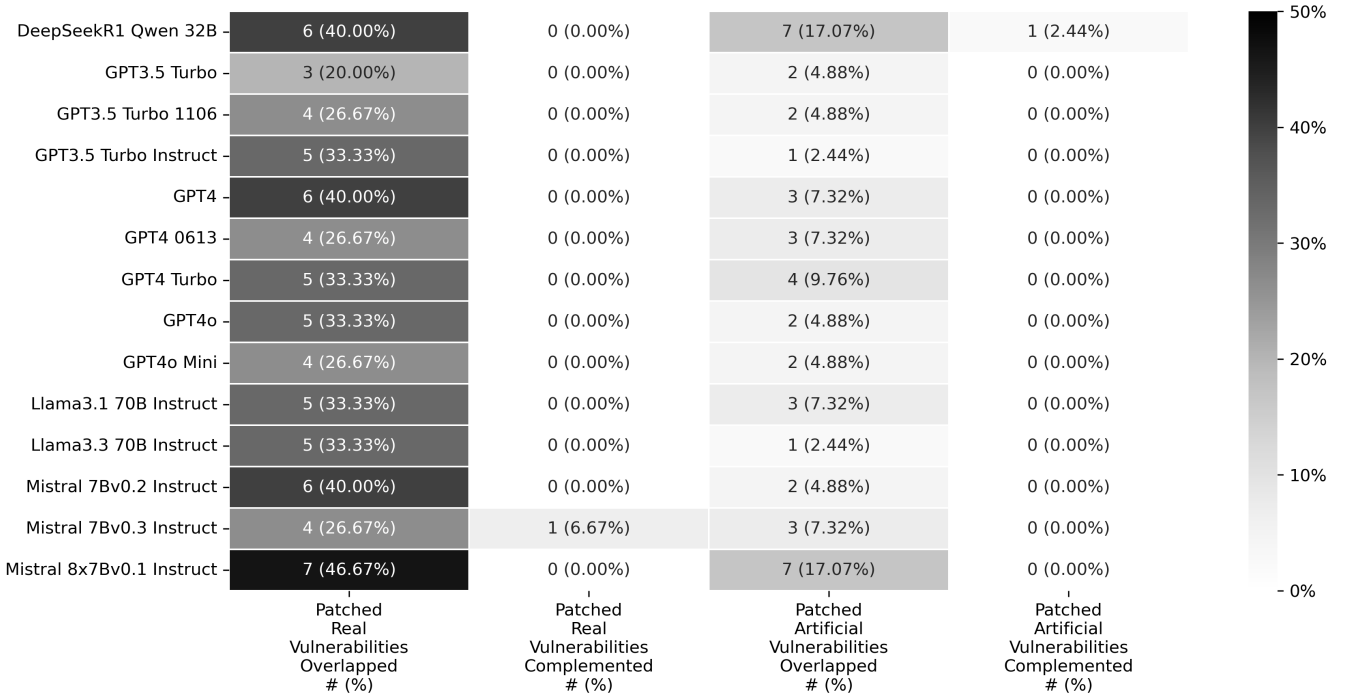

Figure 4: Overlapping and complementarity of LLMs in patching real and artificial vulnerabilities

Consequently, a patch that passes the existing PoV tests may still leave latent attack vectors unexercised, resulting in false negatives. Notably, *Wang et al.* [41] report that even the best LLM addressed *only 5 out of 23 real CVEs* (approximately 21.7%) under strict exploit-based validation. This suggests our results are consistent with low LLM patching rates found in other settings.

Another threat lies in the absence of a direct comparison with traditional or domain-specific patching solutions. We did not integrate baseline tools such as classical APR frameworks or static-analysis-driven fixers when evaluating the LLMs. This was intentional to maintain our study's focus on how different LLM-based approaches compare to one another, leaving cross-paradigm comparisons for future work.

Internal Validity. The threat to internal validity involves potential variations in LLM-responses due to their inherent stochasticity. To mitigate this, we used a consistent and standardized prompt across all experiments and evaluated each LLM independently. Running multiple generation trials per vulnerability could further mitigate randomness, but our single-prompt, single-run setup reflects a realistic one-shot usage scenario.

Another threat lies with our limited prompting strategy. Since our evaluation employs a standardized single prompt, this may not fully leverage the LLMs' potential. More sophisticated prompting strategies could improve patch-generation success and overall model performance. We chose not to explore those variations here in order to keep the evaluation consistent across models, leaving a study on advanced prompting techniques as future work.

External Validity. The threat to external validity involves the selection and representativeness of vulnerabilities included in our study. Although we considered a diverse set of CVEs and artificial vulnerabilities, our results might not generalize to other types of vulnerabilities or different scenarios.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented an empirical evaluation of Large Language Models (LLMs) in automated vulnerability patching. Our analysis, covering real and artificial vulnerabilities, highlights key insights into the capabilities and limitations of these models. Our results indicate that while LLMs demonstrate considerable effectiveness in patching real vulnerabilities, their performance significantly drops when addressing artificial vulnerabilities. This difference underscores a critical gap in their current ability to generalize patching strategies beyond previously encountered vulnerability patterns.

Additionally, our findings reveal that patching overlap among LLMs is notably higher for real vulnerabilities compared to artificial ones. This suggests that artificial vulnerabilities, despite exhibiting the same failure patterns, pose unique challenges that result in diverse patching outcomes across different LLMs.

In the future, we envision several avenues to extend this research. First, we plan to explore multi-model ensembling for patch generation, e.g., combining the suggestions of multiple LLMs or multiple independent executions to increase patch correctness. An ensemble could vote on or cross-verify patches, potentially overcoming individual model limitations, e.g., by voting or consensus among multiple LLM outputs. Secondly, we aim to investigate prompt tuning for this task. By tuning the prompts, we may significantly improve the models' understanding of security fixes. Such tailored prompting could reduce errors that occur in single-prompt mode. Thirdly, we would like to expand the dataset to include a larger

and more diverse set of vulnerabilities, covering additional vulnerability categories and languages (beyond those in the current *Vul4J* subset). This will require reliable Proof-of-Vulnerability (PoV) tests for every new case, which is a labor-intensive manual process involving a thorough analysis of the vulnerable program, its human patch, and crafting an exploit that fails only when the fix is correct. Creating these artifacts is beyond our current scope, yet it remains an important next step for a broader generalization. Finally, exploring a combination of approaches like integrating static analysis with LLM suggestions or an interactive human-in-the-loop patch validation is an exciting direction to increase the practicality of LLM-based vulnerability patching.

REFERENCES

[1] Bui, Q., Paramitha, R., Vu, D., Massacci, F., Scandariato, R.: Apr4vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities. Empir. Softw. Eng. **29**(1), 18 (2024). https://doi.org/10.1007/S10664-023-10415-7, https://doi.org/10.1007/s10664-023-10415-7

[2] Bui, Q., Scandariato, R., Ferreyra, N.E.D.: Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In: 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022. pp. 464–468. ACM (2022). https://doi.org/10.1145/3524842.3528482, https://doi.org/10.1145/3524842.3528482

[3] Cve-2018-1000850. https://nvd.nist.gov/vuln/detail/CVE-2018-1000850 ((accessed January 10, 2023))

[4] Cve-2018-17201. https://nvd.nist.gov/vuln/detail/CVE-2018-17201 ((accessed January 10, 2023))

[5] Dibia, V., Fourney, A., Bansal, G., Poursabzi-Sangdeh, F., Liu, H., Amershi, S.: Aligning offline metrics and human judgments of value for code generation models. In: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (eds.) Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023. pp. 8516–8528. Association for Computational Linguistics (2023). https://doi.org/10.18653/V1/2023.FINDINGS-ACL.540, https://doi.org/10.18653/v1/2023.findings-acl.540

[6] Dikici, S., Bilgin, T.T.: Advancements in automated program repair: a comprehensive review. Knowl. Inf. Syst. **67**(6), 4737–4783 (Mar 2025). https://doi.org/10.1007/s10115-025-02383-9, https://doi.org/10.1007/s10115-025-02383-9

[7] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R.: Lava: Large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 110–121 (2016). https://doi.org/10.1109/SP.2016.15

[8] Eghbali, A., Pradel, M.: Crystalbleu: Precisely and efficiently measuring the similarity of code. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022. pp. 28:1–28:12. ACM (2022). https://doi.org/10.1145/3551349.3556903, https://doi.org/10.1145/3551349.3556903

[9] Eladawy, H., Le Goues, C., Brun, Y.: Automated program repair, what is it good for? not absolutely nothing! In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639095, https://doi.org/10.1145/3597503.3639095

[10] Fakih, M., Dharmaji, R., Bouzidi, H., Araya, G.Q., Ogundare, O., Faruque, M.A.A.: LLM4CVE: enabling iterative automated vulnerability repair with large language models. CoRR **abs/2501.03446** (2025). https://doi.org/10.48550/ARXIV.2501.03446, https://doi.org/10.48550/arXiv.2501.03446

[11] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020. Findings of ACL, vol. EMNLP 2020, pp. 1536–1547. Association for Computational Linguistics (2020). https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.139, https://doi.org/10.18653/v1/2020.findings-emnlp.139

[12] Garg, A., Degiovanni, R., Jimenez, M., Cordy, M., Papadakis, M., Traon, Y.L.: Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach. CoRR **abs/2012.11701** (2020), https://arxiv.org/abs/2012.11701

[13] Garg, A., Degiovanni, R., Jimenez, M., Cordy, M., Papadakis, M., Traon, Y.L.: Learning from what we know: How to perform vulnerability prediction using noisy historical data. Empir. Softw. Eng. **27**(7), 169 (2022). https://doi.org/10.1007/S10664-022-10197-4, https://doi.org/10.1007/s10664-022-10197-4

[14] Garg, A., Degiovanni, R., Papadakis, M., Traon, Y.L.: Vulnerability mimicking mutants. CoRR **abs/2303.04247** (2023). https://doi.org/10.48550/ARXIV.2303.04247, https://doi.org/10.48550/arXiv.2303.04247

[15] Garg, A., Degiovanni, R., Papadakis, M., Traon, Y.L.: On the coupling between vulnerabilities and llm-generated mutants: A study on vul4j dataset. In: IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024. pp. 305–316. IEEE (2024). https://doi.org/10.1109/ICST60714.2024.00035, https://doi.org/10.1109/ICST60714.2024.00035

[16] Garg, A., Ojdanic, M., Degiovanni, R., Chekam, T.T., Papadakis, M., Traon, Y.L.: Cerebro: Static subsuming mutant selection. IEEE Trans. Software Eng. **49**(1), 24–43 (2023). https://doi.org/10.1109/TSE.2022.3140510, https://doi.org/10.1109/TSE.2022.3140510

[17] Ghanbari, A.: Toward practical automatic program repair. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1262–1264 (2019). https://doi.org/10.1109/ASE.2019.00156

[18] Halfond, W.G.J., Viegas, J., Orso, A.: A classification of SQL injection attacks and countermeasures. In: Jr., S.T.R. (ed.) 2006 IEEE International Symposium on Secure Software Engineering, ISSSE 2006, Arlington, VA, USA, March 16 -17, 2006 (2006)

[19] Hsueh, M., Tsai, T.K., Iyer, R.K.: Fault injection techniques and tools. Computer **30**(4), 75–82 (1997). https://doi.org/10.1109/2.585157, https://doi.org/10.1109/2.585157

[20] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62, https://doi.org/10.1109/TSE.2010.62

[21] Just, R., Jalali, D., Ernst, M.D.: Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. p. 437–440. ISSTA 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2610384.2628055, https://doi.org/10.1145/2610384.2628055

[22] Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 654–665. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635929, https://doi.org/10.1145/2635868.2635929

[23] Khan, Z.A., Garg, A., Guo, Y., Tang, Q.: Evaluating pre-trained models for multi-language vulnerability patching. CoRR **abs/2501.07339** (2025). https://doi.org/10.48550/ARXIV.2501.07339, https://doi.org/10.48550/arXiv.2501.07339

[24] Khan, Z.A., Garg, A., Tang, Q.: A Multi-Dataset Evaluation of Models for Automated Vulnerability Repair. International Workshop on Artificial Intelligence, Cyber and Cyber-Physical Security (AI&CCPS), ARES 2025 (May 2025). https://doi.org/10.48550/arXiv.2506.04987

[25] Kulsum, U., Zhu, H., Xu, B., d'Amorim, M.: A case study of LLM for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In: Adams, B., Zimmermann, T., Ozkaya, I., Lin, D., Zhang, J.M. (eds.) Proceedings of the 1st ACM International Conference on AI-Powered Software, AIware 2024, Porto de Galinhas, Brazil, July 15-16, 2024. ACM (2024). https://doi.org/10.1145/3664646.3664770, https://doi.org/10.1145/3664646.3664770

[26] Lin, D., Koppel, J., Chen, A., Solar-Lezama, A.: Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In: Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. p. 55–56. SPLASH Companion 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3135932.3135941, https://doi.org/10.1145/3135932.3135941

[27] Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In: Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023 (2023)

[28] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B.P., Hermann, K., Welleck, S., Yazdanbakhsh, A., Clark, P.: Self-refine: iterative refinement with self-feedback. In: Proceedings of the 37th International Conference on Neural Information Processing Systems. NIPS '23, Curran Associates Inc., Red Hook, NY, USA (2023)

[29] Meem, F.N., Smith, J., Johnson, B.: Exploring experiences with automated program repair in practice. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639182, https://doi.org/10.1145/3597503.3639182

[30] Naik, A.: On the limitations of embedding based methods for measuring functional correctness for code generation. CoRR **abs/2405.01580** (2024).

https://doi.org/10.48550/ARXIV.2405.01580, https://doi.org/10.48550/arXiv.2405.01580

[31] Nasrabadi, M.Z., Parsa, S., Ramezani, M., Roy, C., Ekhtiarzadeh, M.: A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. J. Syst. Softw. **204**, 111796 (2023). https://doi.org/10.1016/J.JSS.2023.111796, https://doi.org/10.1016/j.jss.2023.111796

[32] Nong, Y., Yang, H., Cheng, L., Hu, H., Cai, H.: Automated software vulnerability patching using large language models. CoRR **abs/2408.13597** (2024). https://doi.org/10.48550/ARXIV.2408.13597, https://doi.org/10.48550/arXiv.2408.13597

[33] Nong, Y., Yang, H., Cheng, L., Hu, H., Cai, H.: Appatch: automated adaptive prompting large language models for real-world software vulnerability patching. In: Proceedings of the 34th USENIX Conference on Security Symposium. SEC '25, USENIX Association, USA (2025)

[34] Ojdanic, M., Garg, A., Khanfir, A., Degiovanni, R., Papadakis, M., Traon, Y.L.: Syntactic versus semantic similarity of artificial and real faults in mutation testing studies. IEEE Trans. Software Eng. **49**(7), 3922–3938 (2023). https://doi.org/10.1109/TSE.2023.3277564, https://doi.org/10.1109/TSE.2023.3277564

[35] Ojdanic, M., Khanfir, A., Garg, A., Degiovanni, R., Papadakis, M., Le Traon, Y.: On comparing mutation testing tools through learning-based mutant selection. In: 2023 IEEE/ACM International Conference on Automation of Software Test (AST). pp. 35–46 (2023). https://doi.org/10.1109/AST58925.2023.00008

[36] Pearce, H., Tan, B., Ahmad, B., Karri, R., Dolan-Gavitt, B.: Examining zero-shot vulnerability repair with large language models. In: 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. pp. 2339–2356. IEEE (2023). https://doi.org/10.1109/SP46215.2023.10179420, https://doi.org/10.1109/SP46215.2023.10179420

[37] Prenner, J.A., Babii, H., Robbes, R.: Can openai's codex fix bugs? an evaluation on quixbugs. In: Proceedings of the Third International Workshop on Automated Program Repair. p. 69–75. APR '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524459.3527351, https://doi.org/10.1145/3524459.3527351

[38] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis. CoRR **abs/2009.10297** (2020), https://arxiv.org/abs/2009.10297

[39] Renzullo, J., Reiter, P., Weimer, W., Forrest, S.: Automated program repair: Emerging trends pose and expose problems for benchmarks. ACM Comput. Surv. **57**(8) (Mar 2025). https://doi.org/10.1145/3704997, https://doi.org/10.1145/3704997

[40] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: language agents with verbal reinforcement learning. In: Proceedings of the 37th International Conference on Neural Information Processing Systems. NIPS '23, Curran Associates Inc., Red Hook, NY, USA (2023)

[41] Wang, W., Ma, W., Hu, Q., Zhang, Y., Sun, J., Wu, B., Liu, Y., Xu, G., Jiang, L.: Vulnrepaireval: An exploit-based evaluation framework for assessing large language model vulnerability repair capabilities (2025), https://arxiv.org/abs/2509.03331

[42] Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models (2023), https://arxiv.org/abs/2203.11171

[43] Wu, Y., Jiang, N., Pham, H.V., Lutellier, T., Davis, J., Tan, L., Babkin, P., Shah, S.: How effective are neural networks for fixing security vulnerabilities. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023. pp. 1282–1294. ACM (2023). https://doi.org/10.1145/3597926.3598135, https://doi.org/10.1145/3597926.3598135

[44] Ye, H., Martinez, M., Durieux, T., Monperrus, M.: A comprehensive study of automatic program repair on the quixbugs benchmark. Journal of Systems and Software **171**, 110825 (2021). https://doi.org/https://doi.org/10.1016/j.jss.2020.110825, https://www.sciencedirect.com/science/article/pii/S0164121220302193

[45] Zhang, M., Yin, H.: Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014), https://www.ndss-symposium.org/ndss2014/appsealer-automatic-generation-vulnerability-specific-patches-preventing-component-hijacking

[46] Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., Chen, Z.: A survey on large language models for software engineering. CoRR **abs/2312.15223** (2023). https://doi.org/10.48550/ARXIV.2312.15223, https://doi.org/10.48550/arXiv.2312.15223

[47] Zibaeirad, A., Vieira, M.: Vulnllmeval: A framework for evaluating large language models in software vulnerability detection and patching. CoRR **abs/2409.10756** (2024). https://doi.org/10.48550/ARXIV.2409.10756, https://doi.org/10.48550/arXiv.2409.10756