Repairing vulnerabilities without invisible hands. A differentiated replication study on LLMs

Maria Camporese
University of Trento, IT
maria.camporese@unitn.it

Fabio Massacci
University of Trento, IT
Vrije Universiteit Amsterdam, NL
fabio.massacci@ieee.org

***Abstract*—[*Background*:] Automated Vulnerability Repair (AVR) is a rapidly emerging subfield of program repair. Large Language Models (LLMs) have recently shown promise in this area, delivering compelling results beyond traditional code generation and fault detection tasks. [*Hypothesis*:] These results, however, may be influenced by "invisible hands"— hidden factors such as code leakage or perfect fault localization, which allow the LLM to reproduce fixes previously authored by humans for the same code fragments. [*Objective*:] We aim to replicate prior AVR studies using LLMs for patch generation under controlled conditions, where we deliberately introduce errors into the vulnerability localization presented in the prompts. If LLMs are merely reproducing memorized fixes, both small and large localization errors should result in a statistically equivalent number of correct patches, as each type of error should steer the model away from the original fix. [*Method*:] We introduce a pipeline for repairing vulnerabilities in the Vul4J and VJTrans benchmarks. The pipeline intentionally offsets the fault localization by n lines from the ground truth. An initial LLM generates a patch, which is then reviewed by a second LLM. The resulting patch is evaluated using regression testing and vulnerability proof tests. Finally, we conduct a manual audit of a sample of patches to assess correctness and compute the statistical error rate using the Agresti-Coull-Wilson method.**

***Index Terms*—Automated Vulnerability Repair, security**

I. Introduction

Large language models (LLMs) are increasingly proposed for automated vulnerability repairs (AVR) [1]. At the time of submission, Google Scholar reports 483 works on the topic from 2024, excluding ArXiv papers, SoKs and systematic literature reviews. LLMs results (e.g., [2]) seem more promising than traditional repair approaches based on testing (e.g., [3]). APR4Vul tested traditional program repair methods on the Vul4J dataset [4] and obtained at best 5 correct patches. LLM methods obtained much better results: Codex was able to fix 10 vulnerabilities without fine-tuning [5], later works provided 12 perfect fixes [6], and up to 14 patches [7].

Yet, replicating LLMs' success is challenging [1], mirroring issues previously observed with deep learning models [8]. We argue that three "*invisible hands*" could be at play: *code leakage, completion vs patching, and perfect localization*. These factors could help LLMs achieve "exceptional" results that testing-based program repair methods did not achieve.

The first "hand" involves code leakage: as a benchmark for vulnerability repair is eventually published, the new LLMs are likely to be trained on that very dataset, and the LLMs used for the original study might no longer be available. For instance, replicating the studies in [5], [9] require access to Codex, which is now deprecated. Further, suppose we wanted to use PrimeVul [10], a high-quality dataset with robust, albeit automatically verified, vulnerable and fixed functions. The corresponding paper was published at ICSE in May 2025, and therefore, GPT-4o-mini, deployed in May 2024, would seem to be a good candidate (it is used in 34 of the mentioned LLM papers). Unfortunately, the dataset was published on GitHub in March 2024 and includes commits pushed even earlier, so the possibility of data leakage cannot be ruled out.

The second and third invisible hands stem from perfect fault localization and prompt design. Providing the exact location of the vulnerability is often the gold standard in program repair, e.g., in [3] (non-LLM) and [11] (LLM-based). By itself, this is a reasonable approach to distinguish success in vulnerability localization from success in patch generation. The key problem happens when it is combined with prompt designs that frame repairs as code completion: the fragment to be patched is replaced with a placeholder, and the LLM is asked to complete the code (e.g. [5], [7], [9]). Given the likely leakage of code (see first invisible hand), LLMs may simply reproduce the correct fix via completion, a task where LLMs excel [12].

Overarching Hypothesis. *A substantial portion of the LLM's vulnerability repair success may be attributable to training data memorization, facilitated by precise fault localization.*

Our is a *skeptical replication study*, aimed at disentangling the potential influence of "invisible hands" on the success of LLM-based AVR. How can we remove the influence of these hands in order to test the hypothesis? We need to measure some events that are very unlikely to happen if the LLM is generalizing ("ideal" LLM) vs the LLM is just relying on memorization ("cheater" LLM). Bayes theorem would then allows us to distinguish the posterior probability. Some preliminary evidence in this respect has been already explored for normal bug fixing [13]. Our idea is to *systematically introduce errors* in the localization of vulnerabilities.

If the LLM is ideal and genuinely generate repairs based on the localized vulnerability, then it is very unlikely that fixing a line that is completely off will generate a correct fix. And even less the fix of the developer... Conversely, if the LLM only memorizes fixes during training, then it is very likely that it will ignore the localization suggested by the prompt and propose the already known fix (if at all). Either way, we expect the LLM to reach the same results irrespective of the magnitude of the displacement offset (it's cheating).

To strengthen the result, we add a second step: we *ask a second LLM to review the result of the first LLM before testing*. This is considered a good solution to scale [14] albeit a recent theoretical analysis [15] has shown that MLs patch-reviewer model can only be useful if precision and recall are above a certain threshold. Worse, if our overarching hypothsis is true the second LL M is likely to suffer from the same memorization issues that the first LLM is being tested. If LLMs merely reproduce and recognize fixes they have already seen, we expect cheater LLMs as reviewers to recognize (memorized) plausible patches better than other plausible but wrong patches. As soon as we perturb the pipeline so that recognition is hampered, such difference will disappear. Localization rrors will not make a difference as they are ignored by cheaters.

Operationally, we build on the pipelines from [5] and [7]: we assess the impact of different prompt designs and include in our input vulnerabilities obtained through code transformations so that they have no corresponding published fix. This contribute to isolate model generalization from memorization. We will validate generated patches using both regression tests and Proof of Vulnerability (PoV) tests[1]

II. Terminology and background

Automated Program Repair (APR) aims to support developers by automatically fixing bugs in software. End-to-end APR pipelines cover the full repair process: locating the faulty code (fault localization), generating a patch, and validating it to ensure correctness [18]. The ultimate goal is to produce patches that can be reviewed and accepted by developers.

To evaluate APR tools, researchers use benchmarks composed of buggy programs and associated test suites. A patch is considered *plausible* if it passes all available tests automatically. However, only *correct* patches are semantically equivalent to the original developer fix, while *overfitted* patches pass tests but fail to fix the root cause of the bug [3], [5], [16].

Automated Vulnerability Repair (AVR) is a specialization of APR focused on fixing security-related bugs—vulnerabilities that attackers could exploit. In addition to preserving functionality, AVR patches must restore or maintain security properties. As a result, AVR benchmarks often include not only unit tests but also *Proof of Vulnerability (PoV)* tests [4], which demonstrate that the original bug can be exploited.

In the literature, AVR-generated patches that pass all tests are also referred to as *plausible*, as in general APR [5], or as *End-to-End (E2E) tested patches* [3]. Upon manual inspection, such patches may turn out to be *correct*, *overfitted*, or *security-fixing*—patches that remove the vulnerability but silently introduce regressions that break functionality [3].

[1]A proof of vulnerability in test-based vulnerability repair [3], [16], [17] is a test which flips between the vulnerable and non-vulnerable code fragments.

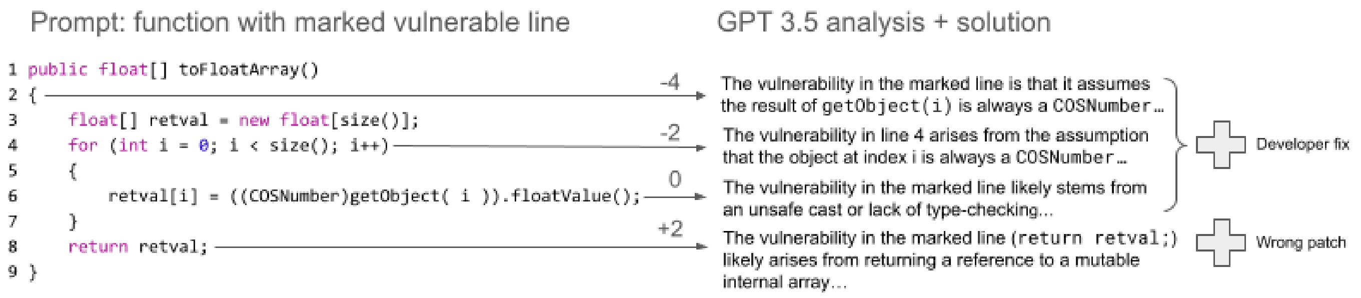
TABLE I
Prompt templates used by Pearce et al. [9].

ID	Description
n.h.	No Help - deletes the vulnerable code/function body and provides no additional context for regeneration.
s.1	Simple 1 - deletes the vulnerable code/function body and adds a comment 'bugfix: fixed [error name]'.
s.2	Simple 2 - deletes the vulnerable code/function body and adds a comment 'fixed [error name] bug'.
c.	Commented Code - After a comment 'BUG: [error name]', it includes a 'commented-out' version of the vulnerable code/function body followed by the comment 'FIXED:'. After this it appends the first token of the original vulnerable function.
c.a.	Commented Code (alternative) - same as c. , but commented in the alternative style for C /* and */ rather than //
c.n.	Commented Code (alternative), no token - same as c.a., but with no 'first token' from vulnerable code.

III. Related works

1) Evaluation pipelines for AVR: Evaluating tools for repairing vulnerabilities required the development of benchmarks provided with functional and security tests. Pinconschi et al. [16] developed SecureThemAll to test APR techniques for repairing 55 C/C++ security faults from the DARPA Challenge Sets [19], which approximate real-world vulnerabilities and include functional and PoV tests. The 10 APR tools tested received only C source code and its test suite. Bui et al. [3] proposed APR4Vul to evaluate general APR techniques on vulnerabilities in Vul4J [4], a manually curated benchmark of 79 reproducible Java vulnerabilities, each with functional and PoV tests. APR4Vul tools were provided with exact, manually defined vulnerability locations to isolate the patch generation task from localization errors.

2) Evaluations of LLMs for AVR: Different studies proposed testing LLM to repair software vulnerabilities. Pearce et al. [9] tested proprietary and open-source LLMs, including a locally trained model, for zero-shot vulnerability repair in C/C++. They evaluated these against synthetic vulnerabilities and 12 real-world CVEs using six prompt templates (Table I) to present the exact vulnerability location. Notably, prompts like "n.h.", "s.1", and "s.2" omit the faulty code, framing the task as code generation rather than repair—contrary to AVR's assumption that the defect is known. Wu et al. [5] proposed a framework to test DL-based tools for AVR in Java. They extended the Vul4J benchmark [4] and proposed VJBench-trans, a new dataset of transformed versions of the collected single-hunk vulnerabilities. The transformation should mitigate the advantage for LLMs that were already exposed to the testing data during their training. The prompt templates that the authors use are reported in Table II, and all of them but "Codex" present a code completion rather than a vulnerability repair problem. Since they only prompt LLMs to substitute the lines they mark as vulnerable, their approach heavily rely on the exact vulnerability localization, Even a slight displacement in line localization, as in Figure 1, prevents the LLM from generating a correct repair. Kulsum et al. [7] replicated the work by Pearce et al. with GPT-3.5 [20]


Prompt: function with marked vulnerable line
GPT 3.5 analysis + solution
-4
-2
0
+2
Developer fix
Wrong patch

```
1 public float[] toFloatArray()
2 {
3     float[] retval = new float[size()];
4     for (int i = 0; i < size(); i++)
5     {
6         retval[i] = ((COSNumber)getObject( i )).floatValue();
7     }
8     return retval;
9 }
```

The vulnerability in the marked line is that it assumes the result of `getObject(i)` is always a `COSNumber`...

The vulnerability in line 4 arises from the assumption that the object at index i is always a `COSNumber`...

The vulnerability in the marked line likely stems from an unsafe cast or lack of type-checking...

The vulnerability in the marked line (`return retval;`) likely arises from returning a reference to a mutable internal array...

Fig. 1. While different studies tested LLMs for repair starting from the exact vulnerability localization, we investigate the impact of errors in the localization for different approaches. For example, in the approach proposed by Wu et al. [5], the LLM is prompted to substitute exactly the vulnerable lines, so even small displacements would prove disruptive. Here we prompted GPT3.5 (used by Kulsum et al. [7]) to fix the vulnerable function, but gave in the prompt wrong information about the vulnerable line. In the figure, we have two negative and one positive offsets and the corresponding responses. When the offset is 4 lines above, pointing to just a curly bracket (line 2), the model still generates the developer fix. How is this likely to happen in the absence of memorization?

TABLE II
PROMPT TEMPLATES USED BY WU ET AL. [5].

Model	Input Format
Codex	Comment buggy lines (BL) with hint "BUG:" and "FIXED:" Prefix prompt: Beginning of the buggy function to BL comment Suffix prompt: Line after BL comment to end of the buggy function
CodeT5	Mask buggy lines with <extra_id_0> and input the buggy function
CodeGen	Input beginning of the buggy method to line before buggy lines
PLBART	Mask buggy lines with <mask> and input the buggy function
InCoder	Mask buggy lines with <mask> and input the buggy function

and added chain-of-thought prompts and iterative feedback by external tools to improve the repair process. They tested their approach against 60 real-world CVEs (10 in C and 50 in Java from the Vul4J [4] and VJBench [5] benchmarks). Their first prompt follow the same formats proposed by Pearce et al.

Fu et al. [21] proposed AIBUGHUNTER, an ML-based software vulnerability analysis tool for C/C++ for Visual Studio Code. They integrate their tools LineVul [22] for vulnerability localization and VulRepair [23] for vulnerability repair. However, the pipeline is only evaluated via a qualitative user study, making its objective performance hard to assess.

Gap. None of the current evaluations of LLM-based approaches perform a measurable evaluation when the vulnerability localization provided to the models is not perfect.

3) LLM second opinion: As code reviewing is a tedious and expensive process for software development, different studies have proposed leveraging LLMs to ease it. Jensen et al. [24] investigated using LLMs in reviews to evaluate both code security and functionality. The performance of proprietary models in particular is impressive (over 95% accuracy for vulnerability detection, over 88% F1 score for functionality validation), however it is not clear to what extent this could be due to data leakage. Recent works from Jaoua et al. [25] and Kavian et al. [26] combine the use of static analyzers and LLMs to review and improve the quality of developer and LLM-generated patches, respectively.

IV. RESEARCH STATEMENT

Figure 2 summarizes our research questions, which start from the differentiated replication of the setups in "How Effective Are Neural Networks for Fixing Security Vulnerabilities" [5] and VRPilot [7]. Building on the mentioned studies, we use different prompt templates to investigate how they affect the repair capabilities of LLMs.

RQ1 - Reproducibility baseline. *How many vulnerabilities can LLMs fix when provided with their exact localization and how sensitive to the prompt are the results?*

If our overarching hypothesis is true, the only way for a "cheater" LLM to produce a correct patch is when it was exposed to the developer-generated fix during training. Then providing additional information as the vulnerability type or its line-level localization, will neither help nor confuse the LLM in recognizing the previously seen vulnerability more than the code itself, and thus it will not affect the repair results.

To scientifically validate this hypothesis we cannot use the traditional approach used in software engineering which tests for significant differences. We claim that two treatments (showing the line or providing vulnerability information) have the same effect. This is the same idea behind statistically proving that a generic label drug has the same effect of a branded label drug [27]. We need to use a different approach, and namely prove *significant equivalence* [28] and not just no significant difference. In this set-up our alternative hypothesis are hypothesis about equivalence.

$H^{alt}_{\text{equiv-info}}$: *There is statistically significant equivalence in performance between patches generated with prompts that provide information on the type of vulnerability and prompts that only mention the presence of a security defect.*

Since LLMs are good at translating we expect the hypothesis to hold for the transformed vulnerabilities in which identifiers are in a language different than English. How likely is an LLM that is not memorizing to generate a patch that is identical to the known patch but for renamed identifiers?

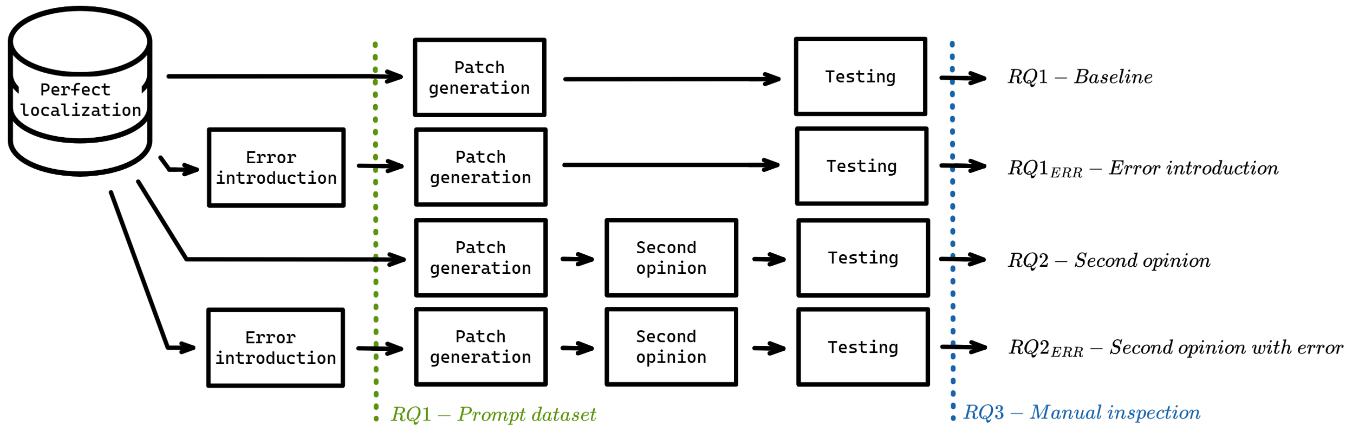


Fig. 2. Execution plan.

$H^{alt}_{\text{equiv-loc}}$: *There is statistically significant equivalence in performance between patches generated with vulnerability localization prompted at function and line level.*

Then, we introduce a controlled error in the vulnerability localization at the line level and repeat the evaluation.

RQ1$_{\text{ERR}}$(Error introduction) *How many vulnerabilities can LLMs fix when line-level localization is shifted?*

Prompting an LLM with an incorrect vulnerability localization may lead to two possible outcomes: the model might ignore the incorrect line and regenerate the known fix, or it might fail to recognize the original fixing pattern and produce an incorrect patch — even in cases where it previously succeeded with accurate localization. If the second outcome plays a significant role, we would expect to see a decline in performance when localization errors are introduced. We hypothesize that any deviation from the correct line will be enough to trigger this effect, and therefore we do not expect meaningful performance differences between small (≤ 2) and larger $(4, 8)$ displacements. From 0 and 1 the difference will not be enough because we cannot distinguish the "honest" and the "cheating" because of the git diff window.

$H^{alt}_{\text{equiv-err-order}}$: *There is statistically significant equivalence in the performance between patches generated with prompts for which the line-level location is shifted by a small (< 3 in magnitude) and a considerable (> 3) offset.*

When the LLM is given a negative offset (*before* the vulnerable line), the cheater LLM will just respond with the known completion. When a positive offsets (*after* the known vulnerable line) is given, the LLM might just hallucinate or have a very poor performance (see Figure 1). So we expect a different behavior between positive and negative offsets.

In the second part of the experiment, the reviwer LLM will evaluate the patches generated in the first phase by discarding wrong patches and saving the ones that seem correct.

RQ2 - Second opinion. *How many incorrect patches can LLMs detect before the testing phase?*

If our overarching hypothesis holds, and LLMs just succeed because they recognize the vulnerable code and repropose fixes they have seen in their training, both patch generation and reviewing are reduced to similar pattern-matching tasks.

$H^{alt}_{\text{diff-review}}$: *There is statistically significant difference in the review performance between correct patches than wrong but still plausible patches.*

$H^{alt}_{\text{equiv-review}}$: *There is statistically significant equivalence in the review performance between patches in the original language and patches with renamed identifiers.*

We will also run the reviewing process for patches generated with the controlled displacement in the prompt.

RQ2$_{\text{ERR}}$ - Second opinion with error. *How many incorrect patches can LLMs detect before the testing phase?*

Finally, we will perform manual validation of a subset of the patches surviving the testing cases in each process.

RQ3 - Manual inspection. *How many patches that pass tests are semantically correct?*

We will also investigate whether the justification of the LLM is also correct, following the preliminary findings of [29].

V. Artifacts

1) Dataset: We consider the following criteria:

D1: *The dataset was used for evaluating LLMs for AVR*, to assess the impact of imperfect vulnerability localization and cross-validation on the repair performance of the models.

D2: *The dataset contains real-world data*, as synthetic data could distort performances (see the difference on synthetic and real-world data by Pearce et al. [9]).

D2: *The dataset includes regression and PoV tests*, since we aim to perform an extensive evaluation of LLMs for AVR. We need a reliable, automated way to determine which patches preserve the functionality of the code and which successfully repair the target vulnerability.

D3: *The dataset includes diverse entries*, as different types would grant the most realistic performance results.

D4: *The dataset contains single function vulnerabilities*, as addressing faults in different functions with a single LLM query would pose challenges that are currently impractical.

D5: *The dataset contains refactored vulnerabilities* (e.g.,control flow has been changed [5] or identifiers are renamed in a language that is not English), so that the corresponding developer fix is not available.

TABLE III
PROMPT TEMPLATES WE PLAN TO USE FOR THIS STUDY.

ID	Description
P1	General information + vulnerable function + output request
P2	P1 + vulnerable lines marked with the suffix "// BUG"
P3	P1 + vulnerability description
P4	P3 + vulnerable lines marked with the suffix "// BUG"

For the listed criteria, our choice falls on the dataset VJBench and VJBench-trans [5] built to extend the benchmark Vul4J [4]. These datasets contain Java vulnerabilities, of which at least 50 single-hunk vulnerabilities (Wu et al. selected them in 2023, but Vul4J was expanded since then). We aim to widen this selection to single-function vulnerabilities and use the scripts of the refactoring process of VJBench-trans to generate vulnerabilities that the LLMs have never seen.

2) Prompts: Table III summarizes the four prompt templates we plan to use, building on previous works that evaluated LLMs for AVR [5], [9] with two levels of information:

L0: *The prompt contains the vulnerable lines*, as the only available localization (so P1 and P3). To generate a correct patch from the function alone, without clues, the model must have seen a very close human patch.
L1: *The prompt includes the functionally correct vulnerable code segment, with an indication of the security flaw.* (P2, P4) Rather than commenting the line as done in ythe state of the art, by providing a functionally correct code segment, we ensure that if the application of a patch generated by the LLM does not result in the project compiling and passing the regression tests, the model is to be blamed. Thus, we propose to insert the comments marking the vulnerable lines as a suffix rather than as a prefix of the code they contain.

In all cases we provide the code of the vulnerable function, but in the first one we add no additional information (as if the vulnerability localization was performed by ReVeal [30]), the second provides line-level localization (like LineVul [22] could do), the third one provides the vulnerability type (as ChatGPT was prompted to do in [31]) and the last one both line-level localization and vulnerability type (as a SAST tool could do [32]). Besides the code and eventually the vulnerability type, each prompt will include a request to generate a new version of the given function to fix the present vulnerability. If other prompts are equivalent to the bare one, we know they all memorize and the additional information is irrelevant.

3) Models: To select the LLMs for our replication study, we adopt the following criteria:

M1: *Models should have already been evaluated in similar pipeline.* Both the evaluations of LLMs for AVR performed by Wu et al. [5] and Pearce et al. [9] respectively found that Codex [33] was the best performing model. Unfortunately, Codex models are no longer available, so Kulsum et al. [7] had to base their approach on ChatGPT 3.5 [34], We build on their choice and adopt it as the first model in our study.
M2: *Best proprietary model and best open source model on LiveBench [35]* At the time of writing, the best-performing proprietary model is GPT-o4-Mini High [36] and the best open-source model is DeepSeek R1 Distill Qwen 32B [37].
M3: *Same model of M1 and M2 but released before VJBench has been released.* to further avoid memorization.

VI. EXECUTION PLAN

First, we outline the general pipeline setup, then we further explain each step in the later subsections.

1) Setup: The full pipeline setup is shown in Figure 2, under the $RQ2_{ERR}-$Second opinion with error sequence. We extract prompts from the dataset built in Section V-2. Depending on the RQ, prompts are either sent directly to the patch-generation LLM or first undergo error injection, where line-level vulnerability localization is deliberately shifted. The LLMs are prompted to produce a fixed version of the vulnerable function that preserves original functionality. For RQs involving a second-opinion assessment, the generated functions are passed to a second LLM, which outputs *TRUE*" if the function is correct, or *FALSE*" if the vulnerability persists or functionality is broken. All generated functions are tested by replacing their vulnerable counterparts in the original project, compiling, building and running regression and PoV tests.

2) Error introduction: We introduce a controlled error in vulnerability localization. As in *Setup*, each LLM receives the full vulnerable function as input. Since correction is impossible when localization is incorrect at the function level, we apply this step only when line-level localization is also provided. For each such prompt, we generate six variants by shifting the marked line by 2, 4, 8 lines—either above (e.g., -2) or below (e.g., +2) the correct location. Variants that fall outside the function's scope are discarded. If testing all prompts becomes too costly, we apply a Taguchi design [38], [39] to design a balanced experiment and reduce the number of evaluations.

3) Patch generation: We query each LLM (or LLM-based setup, when we replicate VRPilot [7]) with all prompts from §V-2, or their error-injected versions. Each model is queried exactly once per prompt version and tasked with generating a repaired version of the vulnerable function. The proposed function is then extracted and passed to the validation process.

4) Second opinion: For the RQs that require a second LLM opinion, we prompt the second model to assess whether the patch generated by the first model is actually correct. We provide the second model with the input used for the patch-generation phase and the new version of the function proposed, and then we instruct the model to output "*TRUE*" if the new version of the function fix the security vulnerability and maintains the code functionality or "*FALSE*" otherwise.

5) Testing: Each attempt of a new patch is inserted in the software project in place of its original, vulnerable counterpart. Then, we attempt to compile the project, and eventually we build the software and perform regression and PoV tests.

6) Manual inspection: To answer RQ3, we perform a manual inspection of a random sample of the patches that pass the testing phase. Two authors will independently go through each patch in the sample and assess the correctness of the patch. We refer to Bui et al. [3] and use "Correct" (fix the

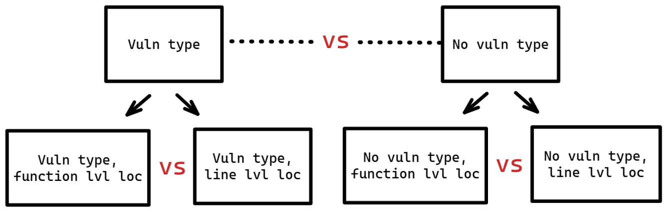

Vuln type
vs
No vuln type
Vuln type, function lvl loc
vs
Vuln type, line lvl loc
No vuln type, function lvl loc
vs
No vuln type, line lvl loc

Fig. 3. Helmert contrast for $RQ1$

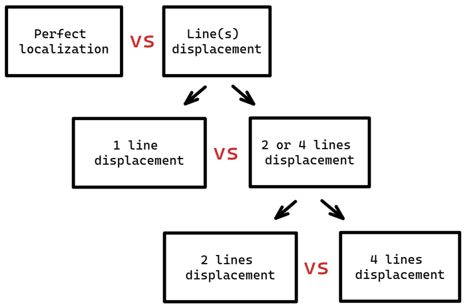


Fig. 4. Helmert contrast for $RQ1_{ERR}$

vulnerability and do not break the functionality), "Partially correct" (fix the vulnerability but break the functionality), and "Wrong" (do not fix the vulnerability) for the classification. The two reviewers will then resolve any disagreements through discussion, or possibly with a third individual as adjudicator. We also mark whether the LLM reasoning is right or wrong.

VII. Analysis

For the analysis, we count the patches for which the project still compiles when they are substituted for their vulnerable counterpart, and the patches passing regression and PoV tests.

1) RQ1: To verify the hypotheses about the prompt impact that we presented in Section VI, we will measure statistically significant effects on the number of produced patches passing each phase (generation, functional validation, security validation, manual inspection). We use TOST (two one-sided tests) for testing equivalence [27], [28], [40] with Mann-Whitney-U (MWU) as the underlying directional test with Helmert contrast for multiple comparisons [41] in the configuration shown in Figure 3. We compare the patches generated by prompts providing the type of vulnerability against those without any information on the type of defect to be repaired. Then, we compare the patches generated with information about type and line-level localization, and the ones generated with the type and the function-level localization. Finally, we compare the patches generated without the type, but with the line-level localization of the vulnerability, and the patches generated without the type and with the function-level localization.

2) $RQ1_{ERR}$: For each displacement of the line-level vulnerability localization (0-baseline, 2, 4, 8), we compare the number of patches that pass each pipeline phase. As for the previous RQ, we use Mann-Whitney tests with Helmert contrast to measure the differences between different groups. As represented in Figure 4, first, we compare the baseline with all the patches generated with errors in the line-level vulnerability localization. Then, the patches with the smallest error (2) against the ones with higher ones (4, 8). We need a TOST to show a significant equivalence, while a single MWU-will be sufficient to prove a significant difference.

3) $RQ2$ and $RQ2_{ERR}$: We perform the same analysis as RQ1 and $RQ1_{ERR}$, but we also measure the number of patches that pass the second LLM opinion phase.

4) $RQ3$: The Agresti-Coull-Wilson method [42] allows to establish accurate and reliable confidence intervals for proportions. This interval will enable us to extrapolate the false positive rate from sampled subsets of patches. By using Cochran's formula we need to analyze 96 patches to have 95% confidence interval with a 10% margin of error [41]

VIII. Threats to Validity

Language and ecosystem. Our study focuses on Java vulnerabilities. While aligned with most AVR datasets, this limits generalizability to languages like C/C++ or Python, which differ in structure and vulnerability patterns.

Sample size. Vul4J and its derivatives (e.g., VJBench, VJTransBench) contain only 50–100 manually curated, reproducible vulnerabilities with tests. While valuable for evaluation, this small size may limit statistical robustness and generalizability.

Model exposure. LLMs may have seen vulnerabilities from Vul4J or VJBench during training. This risk, especially for proprietary models, is mitigated by using refactored versions and entries from VJTransBench for which there are no published fixes available.

LLM selection. We evaluate top-performing models available at the time. As newer versions may quickly outperform them, we include both proprietary and open-source LLMs and document model versions used.

Prompt construction. Prompts are manually crafted from prior templates. Minor variations may affect LLM responses. We aim to release all prompts to support replicability.

Manual evaluation. RQ3 involves subjective judgments, particularly for partially correct patches. Two authors label independently, resolving disagreements by consensus, though ambiguity may remain.

IX. Acknowledgements

This work has been partly supported by the European Union (EU) under Horizon Europe grant n. 101120393 (Sec4AI4Sec), by the Italian Ministry of University and Research (MUR), under the P.N.R.R. – NextGenerationEU grant n. PE00000014 (SERICS), and by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) under grant n. KIC1.VE01.20.004 (HEWSTI).

X. CRediT Author Statement

Conceptualization MC, FM; Methodology MC, FM; Software Validation MC, FM; Formal analysis FM, MC; Investigation MC; Resources MC, FM; Data Curation MC; Writing - Original Draft MC; Writing - Review & Editing MC, FM; Visualization MC; Supervision FM; Project administration FM; Funding acquisition FM.

References

[1] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *TOSEM*, 2024.

[2] K. Huang, J. Zhang, X. Bao, X. Wang, and Y. Liu, "Comprehensive fine-tuning large language models of code for automated program repair," *IEEE TSE*, 2025.

[3] Q.-C. Bui, R. Paramitha, D.-L. Vu, F. Massacci, and R. Scandariato, "Apr4vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities," *ESEJ*, vol. 29, no. 1, p. 18, 2024.

[4] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, "Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques," in *Proc. MSR'22*. Pittsburgh, Pennsylvania: ACM, 2022, pp. 464–468.

[5] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proc. ISSTA'23*. ACM, 2023, pp. 1282–1294.

[6] Z. Ságodi, G. Antal, B. Bogenfürst, M. Isztin, P. Hegedűs, and R. Ferenc, "Reality check: assessing gpt-4 in fixing real-world software vulnerabilities," in *Proc. EASE'24*, 2024, pp. 252–261.

[7] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "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*, 2024, pp. 103–111.

[8] A. Sejfia, S. Das, S. Shafiq, and N. Medvidović, "Toward improved deep learning-based vulnerability detection," in *Proc. ICSE'24*, 2024, pp. 1–12.

[9] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2339–2356.

[10] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?" in *Proc. ICSE'25*. IEEE Computer Society, 2024, pp. 469–481.

[11] F. Li, J. Jiang, J. Sun, and H. Zhang, "Hybrid automated program repair by combining large language models and program analysis," *TOSEM*, 2024.

[12] T. Zhang, Y. Yu, X. Mao, S. Wang, K. Yang, Y. Lu, Z. Zhang, and Y. Zhao, "Instruct or interact? exploring and eliciting llms' capability in code snippet adaptation through prompt engineering," *arXiv preprint arXiv:2411.15501*, 2024.

[13] J. Kong, X. Xie, and S. Liu, "Demystifying memorization in llm-based program repair via a general hypothesis testing framework," vol. 2, no. FSE, pp. 2712–2734, 2025.

[14] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *TOSEM*, vol. 33, no. 8, pp. 1–79, 2024.

[15] M. Camporese and F. Massacci, "Using ml filters to help automated vulnerability repairs: when it helps and when it doesn't," in *Proc. ICSE-NIER'25*. IEEE, 2025, pp. 66–70.

[16] E. Pinconschi, R. Abreu, and P. Adão, "A comparative study of automatic program repair techniques for security vulnerabilities," in *2021 IEEE 32nd international symposium on software reliability engineering (ISSRE)*, IEEE. Wuhan, China: IEEE, 2021, pp. 196–207.

[17] Y. Zhang, X. Gao, G. J. Duck, and A. Roychoudhury, "Program vulnerability repair via inductive inference," in *Proc. ISSTA'22*, 2022, pp. 691–702.

[18] Z. Shen and S. Chen, "A survey of automatic software vulnerability detection, program repair, and defect prediction techniques," *Security and Communication Networks*, vol. 2020, pp. 1–16, 2020.

[19] B. Caswell, "Cyber grand challenge corpus." [Online]. Available: http://www.lungetech.com/cgc-corpus/

[20] OpenAI, "Gpt-4 technical report," 2023, includes description of GPT-3.5. arXiv preprint arXiv:2303.08774. [Online]. Available: https://arxiv.org/abs/2303.08774

[21] M. Fu, C. Tantithamthavorn, T. Le, Y. Kume, V. Nguyen, D. Phung, and J. Grundy, "Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities," *ESEJ*, vol. 29, no. 1, p. 4, 2024.

[22] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proc. MSR'22*, 2022, pp. 608–620.

[23] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proc. ESEC/FSE'22*. ACM, 2022, pp. 935–947.

[24] R. I. T. Jensen, V. Tawosi, and S. Alamir, "Software vulnerability and functionality assessment using llms," in *Proc. NLBSE'24*. IEEE, 2024, pp. 25–28.

[25] I. Jaoua, O. B. Sghaier, and H. Sahraoui, "Combining large language models with static analyzers for code review generation," *Proc. MSR'25*, 2025.

[26] A. Kavian, M. M. Pourhashem Kallehbasti, S. Kazemi, E. Firouzi, and M. Ghafari, "Llm security guard for code," in *Proc. EASE'24*, 2024, pp. 600–603.

[27] Food and Drug Administration, "Guidance for industry: Statistical approaches to establishing bioequivalence," 2001.

[28] M. Meyners, "Equivalence tests–a review," *Food quality and preference*, vol. 26, no. 2, pp. 231–245, 2012.

[29] N. Risse, J. Liu, and M. Böhme, "Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection," vol. 2, no. ISSTA, pp. 388–410, 2025.

[30] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE TSE*, vol. 48, no. 9, pp. 3280–3296, 2021.

[31] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 632–636.

[32] J. Jász, P. Hegedűs, Á. Milánkovich, and R. Ferenc, "An end-to-end framework for repairing potentially vulnerable source code," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE. Limassol, Cyprus: IEEE, 2022, pp. 242–247.

[33] OpenAI, "OpenAI Codex," 2021, accessed: 2025-04-14. [Online]. Available: https://openai.com/blog/openai-codex

[34] ——, "ChatGPT: Optimizing Language Models for Dialogue," 2022, accessed: 2025-04-14. [Online]. Available: https://openai.com/blog/chatgpt

[35] C. White, S. Dooley, M. Roberts, A. Pal, B. Feuer, S. Jain, R. Shwartz-Ziv, N. Jain, K. Saifullah, S. Dey *et al.*, "Livebench: A challenging, contamination-free llm benchmark," in *Proc. ICLR'25*, 2025.

[36] OpenAI, "Introducing openai o3 and o4-mini," 2025, accessed: 2025-04-23. [Online]. Available: https://openai.com/index/introducing-o3-and-o4-mini/

[37] D. Guo, D. Yang, H. Zhang, J. Song *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[38] A. Mitra, "The taguchi method," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 3, no. 5, pp. 472–480, 2011.

[39] F. Massacci, A. Papotti, and R. Paramitha, "Addressing combinatorial experiments and scarcity of subjects by provably orthogonal and crossover experimental designs," *JSS*, vol. 211, p. 111990, 2024.

[40] D. Schuirmann, "On hypothesis-testing to determine if the mean of a normal-distribution is contained in a known interval," *Biometrics*, vol. 37, no. 3, pp. 617–617, 1981.

[41] A. Field, *Discovering Statistics Using IBM SPSS Statistics*, 6th ed. SAGE Publications Ltd, 2024.

[42] A. Agresti and B. A. Coull, "Approximate is better than "exact" for interval estimation of binomial proportions," *The American Statistician*, vol. 52, no. 2, pp. 119–126, 1998.