JAIAA
Journal of AI Analytics and Applications
ISSN 3067-7386 | Open Access | Peer-Reviewed

RESEARCH ARTICLE
Vol. 1, No. 2 (2023), pp. 88–103
DOI: 10.63646/jaiaa.2023.010205

AI Patch Validation Analytics: Exploit-Variant Resistance and Root-Cause Conformance in LLM-Based Vulnerability Repair

Nurul Farhana Ismail[1,*] Hafiz Rahman[2], Aina Syahirah Aziz[3]

[1] Faculty of Information and Communication Technology, Universiti Teknikal Malaysia Melaka, Melaka, Malaysia

[2] Faculty of Computing, Universiti Malaysia Pahang Al-Sultan Abdullah, Pahang, Malaysia

[3] Faculty of Electronic Engineering Technology, Universiti Malaysia Perlis, Perlis, Malaysia

* Corresponding author: nurul.farhana@utem.edu.my

ARTICLE INFO
Received
January 15, 2023
Revised
March 11, 2023
Accepted
May 22, 2023
Available Online
June 30, 2023
DOI
10.63646/jaiaa.2023.010205

Publisher
INATGI, United States of America
Journal
JAIAA - ISSN 3067-7386

Abstract
Large language models are increasingly used to generate candidate security patches, yet a patch that appears successful under one validation signal may still be incomplete, exploit-specific, or unsafe when stronger evidence is applied. This article develops an analytics-oriented framework for evaluating LLM-based vulnerability repair through two security-critical lenses: exploit-variant resistance and root-cause conformance. Instead of treating patch validation as a binary pass/fail decision, the study models validation as a layered evidence profile that combines build validity, functional preservation, original proof-of-concept blocking, exploit-variant resistance, CWE-specific root-cause assessment, static-warning rechecking, and regression-safety review. A controlled patch-validation dataset containing 400 candidate repairs across SQL injection, path traversal, cross-site scripting, and missing authorization scenarios is analyzed to show how weak validation oracles inflate apparent repair success. The analysis demonstrates that original exploit blocking and static-warning disappearance provide useful but incomplete signals. In the primary model configuration, 79.5% of patches blocked the original exploit, whereas 49.0% satisfied the full validation profile. CWE-level results further show that path traversal exhibits strong variant-bypass risk, SQL injection exhibits root-cause conformance gaps, and missing authorization remains difficult because correct repair depends on trust-boundary placement rather than local code edits. The paper contributes a patch-validation analytics model, a reporting structure for oracle divergence, and practical guidance for validation-aware repair workflows. The findings indicate that LLM-based vulnerability repair should be evaluated through layered security evidence rather than by a single permissive oracle.

Keywords: LLM-based vulnerability repair; AI patch validation; exploit-variant resistance; root-cause conformance; oracle divergence; security analytics; automated program repair; software security

I. INTRODUCTION

Large language models have changed the practical imagination of automated vulnerability repair. A developer is now able to paste a vulnerable code fragment into a conversational system and receive a plausible patch in seconds. Repository-level agents extend the same idea to multi-file tasks by searching the project, editing candidate files, running tests, and revising the patch after feedback. These capabilities make LLM-based repair attractive for security triage, open-source maintenance, DevSecOps automation, and rapid response to disclosed vulnerabilities. Yet the convenience of generated code also creates a new validation problem: the patch may look correct, compile successfully, and block a demonstrated exploit while leaving the vulnerability class only partially repaired. Recent C/C++ repair experiments show that real-world vulnerability repair remains difficult even when advanced LLMs are used as patch generators (Zhang et al.,2024). Empirical assessments of deep vulnerability

detection caution that apparent model accuracy may be fragile under dataset and distribution changes (Chakraborty et al.,2021).

The central challenge is that security correctness is not equivalent to functional correctness. Traditional automated program repair often treats the test suite as the main acceptance oracle. Security repair requires additional evidence because a vulnerability is defined by adversarial behavior, trust boundaries, data-flow constraints, authorization semantics, and unsafe interpretation of inputs. A patch for SQL injection is not persuasive merely because it rejects one malicious string; it should remove unsafe query construction. A patch for path traversal is not persuasive merely because it blocks a literal traversal token; it should normalize and confine paths. A patch for missing authorization is not persuasive merely because one endpoint now checks a role; it should enforce the correct access-control decision at the relevant trust boundary. Transformer-based architectures provide the representational foundation for many modern code models used in repair workflows (Vaswani et al.,2017). Developer studies of static analysis explain why analyzer-facing signals should be made actionable instead of being treated as final evidence (Johnson et al.,2013).

This article examines LLM-based vulnerability repair from an analytics perspective. The goal is not to propose a new patch-generation model. Instead, the paper asks how generated patches should be measured after they have been produced. The proposed perspective treats each patch as a data object with multiple validation attributes: build status, benign behavior preservation, original proof-of-concept blocking, exploit-variant resistance, static-analysis status, root-cause conformance, and regression-safety status. The analytical value lies in comparing these attributes rather than collapsing them into one success label. Software security evaluation in connected systems requires attention to both code-level weaknesses and system-level cyber risk (Lu and Xu,2019). Research on next-generation communication systems reinforces the importance of security assurance for increasingly distributed software ecosystems (Lu and Ning,2020).

The distinction matters because weak validation oracles often answer narrow questions. Build success asks whether the edited program is executable. Functional tests ask whether sampled benign behavior remains intact. Original proof-of-concept execution asks whether one demonstrated attack trace fails after patching. Static-warning disappearance asks whether a particular analyzer no longer reports a pattern. Each signal is useful, but each is also incomplete. A repair workflow that accepts a patch immediately after the first positive signal risks adopting an exploit-specific, analyzer-facing, or functionality-breaking fix. Zero-shot repair studies show that LLM-generated fixes may appear plausible while still requiring independent validation evidence (Pearce et al.,2023). Empirical studies of program-analysis needs show that developer-facing repair tools should present interpretable reasons for validation failures (Christakis and Bird,2016).

Two validation dimensions deserve special attention. The first is exploit-variant resistance, which measures whether a patch blocks semantically related attack forms rather than only the original exploit instance. The second is root-cause conformance, which measures whether the patch satisfies vulnerability-class-specific repair expectations. These dimensions are complementary. Variant resistance provides behavioral evidence of generalization, while root-cause conformance provides semantic evidence that the security mechanism is aligned with the vulnerability class. A patch may satisfy one without satisfying the other, and the gap between them is analytically informative. Pre-trained code representations make it feasible to compare vulnerable and repaired code at the semantic-feature level (Feng et al.,2020). Experience reports from large-scale static analysis show that analyzer results are valuable only when embedded in repeatable engineering workflows (Sadowski et al.,2018).

The study develops a patch-validation analytics model and applies it to a controlled dataset of LLM-generated repairs. The dataset covers four vulnerability classes: SQL injection, path traversal, cross-site scripting, and missing authorization. The analysis does not release executable exploit artifacts. Instead, it uses sanitized validation outcomes, aggregate metrics, and structured variables suitable for reproducible security analytics without facilitating misuse. The article is designed for the Journal of AI Analytics and Applications because it frames vulnerability repair evaluation as a problem of AI analytics: how to transform heterogeneous validation signals into interpretable evidence for decision making. Analytics-oriented information-system research is useful for translating technical repair outcomes into organizational risk evidence (Kou and Lu,2025). 6G scenario research further illustrates how future software systems will require automated but auditable security controls (Lu and Zheng,2020).

The paper makes four contributions. First, it defines a validation analytics framework that separates original exploit blocking,

exploit-variant resistance, root-cause conformance, static-warning rechecking, and regression safety. Second, it develops a layered reporting structure for oracle divergence in LLM-based vulnerability repair. Third, it provides a data analysis of 400 candidate patches across four CWE classes and two model configurations. Fourth, it translates the findings into practical guidance for tool builders, benchmark designers, and security teams adopting LLM-assisted repair workflows. Code-generation research demonstrates the productive potential of LLMs, but it also reinforces the need for post-generation verification (Chen et al.,2021). Industrial static-analysis experience shows that warning disappearance may reflect tool behavior and therefore requires complementary validation (Bessey et al.,2010).

The rest of the article is organized as follows. Section II reviews the background on automated repair, LLM-based repair, and validation oracles. Section III presents the research design and validation analytics model. Section IV describes the controlled dataset and measured variables. Section V reports the main results. Section VI discusses implications for security analytics and DevSecOps practice. Section VII addresses limitations and validity concerns. Section VIII concludes the paper.

II. BACKGROUND AND ANALYTICAL FRAMING

Automated program repair has long distinguished plausible patches from correct patches. A plausible patch satisfies the available validation evidence, usually a test suite. A correct patch repairs the underlying defect without violating intended behavior. The difference between the two has motivated a large body of work on patch overfitting, semantic equivalence, benchmark design, and test-suite adequacy (Le Goues et al., 2012; Qi et al., 2015; Monperrus, 2018; Gazzola et al., 2019). LLM-based repair inherits this problem while adding a further complication: generated code may be fluent, idiomatic, and persuasive even when the security reasoning behind it is incomplete. Data-flow-aware code models support the idea that validation should examine semantic paths rather than only surface edits (Guo et al.,2021). Symbolic execution research provides a basis for generating security-relevant tests that explore paths missed by ordinary functional suites (Cadar et al.,2008).

Security vulnerabilities intensify the gap between plausibility and correctness. Vulnerability repair often requires reasoning about attacker-controlled inputs, data-flow paths, authorization state, privilege levels, storage boundaries, encoding contexts, and security invariants. These properties are rarely exhausted by a general-purpose functional test suite. As a result, a patch that passes ordinary tests may still be exploitable, and a patch that blocks a known exploit may still fail a closely related exploit variant. Security validation therefore needs to examine the patch across several evidence layers rather than assume that a single test outcome is decisive. Industry-scale digital transformation studies suggest that security validation should be designed as part of the operational architecture rather than as an afterthought (Lu,2025). Cloud-pricing analytics shows how technical measurements can be transformed into decision-oriented indicators, a logic also used in validation analytics (Lu et al.,2020).

Large language models introduce both opportunity and risk in this setting. They may synthesize secure-coding patterns from massive code corpora, explain repair intent in natural language, and generate candidate edits faster than a human analyst. At the same time, they may overfit to the prompt, imitate superficial security patterns, or choose locally plausible edits that fail under less obvious attack variants. Prior studies of code-generation security have shown that generated code may reproduce insecure practices or provide partial fixes when the validation environment is weak (Pearce et al., 2022; Chen et al., 2021). Neural vulnerability repair systems illustrate why generated patches should be evaluated against security-specific criteria rather than only syntactic similarity (Fu et al.,2022). Whitebox fuzzing research supports the inclusion of adversarially generated test evidence in vulnerability validation (Godefroid et al.,2012).

Validation oracles differ in strength. Textual similarity to a reference patch offers weak evidence because two correct patches may look different and two similar patches may differ in security meaning. Build success is necessary for execution but says little about security. Functional tests protect benign behavior but may not cover adversarial states. Static analysis is scalable and valuable, but warning disappearance may reflect analyzer evasion rather than semantic repair. Original proof-of-concept execution is stronger because it tests demonstrated exploitability, yet it remains bounded by one attack trace. Exploit variants and root-cause conformance provide more demanding evidence because they ask whether the fix generalizes and whether it addresses the vulnerability mechanism. General-purpose foundation models strengthen automated repair interfaces, but their broad capability does not eliminate the need for task-level evidence (OpenAI,2023). Coverage-guided fuzzing research explains why exploit-variant resistance should be treated as a measurable behavioral layer (Bohme et al.,2016).

RESEARCH ARTICLE
Vol. 1, No. 2 (2023), pp. 88–103
DOI: 10.63646/jaiaa.2023.010205

Existing vulnerability benchmarks have improved the field by making repair tasks executable and more comparable. Datasets such as Vul4J and APR4Vul emphasize reproducible vulnerabilities and support empirical assessment of repair tools. More recent LLM and agentic repair benchmarks extend evaluation to repository-level contexts and richer validation environments. These efforts are important, yet absolute success rates remain hard to compare across studies because benchmarks vary in programming language, vulnerability class, prompt design, tool configuration, and acceptance oracle. A patch-validation analytics approach addresses this problem by reporting the performance of several oracles on the same patch set. Decentralized digital systems further demonstrate that software assurance must combine technical correctness with trust and governance evidence (Xu et al.,2024). Quantum-science review work shows the importance of distinguishing conceptual promise from operational validation evidence (Ye and Lu,2022).

The present article therefore treats oracle divergence as a primary analytical object. Oracle divergence occurs when a patch accepted by a weaker validation signal is rejected by a stronger signal. For example, a patch that blocks the original proof-of-concept but fails exploit variants exhibits exploit-specific behavior. A patch whose static warning disappears but whose root-cause criteria are not satisfied exhibits analyzer-facing behavior. A patch that blocks attacks but breaks valid user behavior exhibits functionality-breaking repair. These differences are crucial for security decision making because they indicate why a patch should not be accepted even when a familiar validation signal is positive. Recent vulnerability-repair work using LLMs shows that metric design can strongly influence reported repair quality (de-Fitero-Dominguez et al.,2024). Targeted mutation fuzzing demonstrates that variant generation can be systematic rather than arbitrary (Lemieux and Sen,2018).

Root-cause conformance is especially important because vulnerability classes differ in repair semantics. SQL injection is normally repaired by safe query construction, prepared statements, or safe object-relational mapping rather than keyword blacklisting. Path traversal is repaired by canonicalization, base-directory confinement, and allowlist validation rather than raw string filtering. Cross-site scripting requires output-context-aware encoding or vetted sanitization rather than blocking one tag. Missing authorization requires enforcement at the correct trust boundary with role and object-level checks rather than client-side or endpoint-local patches. A generic success criterion does not capture these class-specific requirements. Identifier-aware pre-training is relevant because security patches often depend on preserving variable roles and API usage semantics (Wang et al.,2021). Modern fuzzing frameworks show that incremental test-generation advances can be integrated into continuous security validation (Fioraldi et al.,2020).

Exploit-variant resistance complements root-cause assessment by evaluating behavioral generalization. It asks whether semantically equivalent inputs still trigger the vulnerability after patching. Variant design should not aim to produce destructive attacks or facilitate misuse. Instead, it should produce controlled, sanitized, local tests that preserve attack intent while modifying representation, encoding, order, boundary values, or context. In evaluation terms, variant resistance provides evidence that a patch is not narrowly tailored to the original demonstration. In analytics terms, it supplies a measurable signal that bridges observed exploit blocking and deeper semantic conformance. Research on quantum machine learning highlights the broader movement toward complex AI systems that require transparent evaluation protocols (Lu et al.,2024). Blockchain trend research shows that software trust problems often require both technical mechanisms and institutional interpretation (Zheng and Lu,2022).

This background motivates the study design. A patch-validation analytics framework should record each evidence layer, compute divergence between weak and stronger oracles, stratify outcomes by vulnerability class, and identify patterns that require developer review. The following section formalizes this objective as an applied analytics model rather than a generation model. Repository-level benchmarks show that realistic software tasks require evidence beyond isolated code-fragment correctness (Jimenez et al.,2024). Smart greybox fuzzing shows how structural feedback can make variant testing more efficient for security-relevant paths (Pham et al.,2017).

III. RESEARCH DESIGN AND VALIDATION ANALYTICS MODEL

The research design is a controlled validation analytics study. Each candidate patch is treated as an observation, and each validation layer is treated as an attribute of that observation. The resulting dataset supports several questions: How many patches pass a weak oracle but fail a stronger one? Which vulnerability classes display the largest gaps between original exploit blocking and full security acceptance? Does a model with high proof-of-concept blocking necessarily achieve high root-cause conformance? Which validation layers provide the strongest practical signal for rejecting unsafe or incomplete repairs? Unified

JAIAA **RESEARCH ARTICLE**
Journal of AI Analytics and Applications **Vol. 1, No. 2 (2023), pp. 88–103**
ISSN 3067-7386 | Open Access | Peer-Reviewed **DOI: 10.63646/jaiaa.2023.010205**

code representation models motivate the use of common evidence schemas across natural-language prompts, code edits, and validation outputs (Guo et al.,2022). Semantic defect prediction research supports the broader claim that code-quality analytics depends on features that capture program meaning (Wang et al.,2016).

The proposed analytics model is organized around six validation dimensions. Build and functionality checks establish a minimum quality floor. Original proof-of-concept blocking tests whether the demonstrated attack trace no longer succeeds. Exploit-variant resistance tests whether semantically related attack forms also fail. Root-cause conformance examines whether the patch satisfies class-specific repair expectations. Static-warning rechecking records whether the analyzer-facing signal disappears or changes after the patch. Regression-safety review checks whether the edit introduces a new weakness or damages legitimate behavior. Blockchain-in-Industry-4.0 research shows why security analytics must address cross-system dependencies rather than isolated components (Chen et al.,2024). Industrial information integration work on quantum computing highlights the need for evaluation frameworks that remain interpretable as technical systems become more complex (Lu et al.,2023).

Figure 1 presents the validation analytics matrix. The figure intentionally avoids a flowchart with arrows because the framework is not only a linear pipeline. In practical use, several layers operate as parallel evidence sources that are later interpreted together. For example, static-warning disappearance is not a final verdict by itself; it becomes meaningful when compared with exploit-based evidence and root-cause conformance. Likewise, original proof-of-concept blocking is not ignored, but it is treated as one evidence object among several.

Layered patch-validation analytics matrix

	Build & Functionality	Original PoC	Exploit Variants	Root-Cause Conformance	Static Warning	Regression Safety
Evidence object	patched program + benign tests	demonstrated exploit trace	equivalent attack family	CWE-specific repair rubric	before/after SAST signal	side-effect checks
Analytics question	Does the patch execute safely?	Does the known exploit fail?	Does the fix generalize?	Was the root cause removed?	Was the alert really repaired?	Did the patch create risk?
Detected risk	invalid or breaking fix	exploit-specific repair	variant bypass	symptom-level repair	analyzer-facing repair	regression-unsafe fix

Figure 1. Patch-validation analytics matrix for LLM-based vulnerability repair.

The matrix clarifies why the article focuses on analytics rather than patch generation. A generation model produces candidate edits, but a validation analytics system decides how those edits should be interpreted. The key output is therefore not only a count of successful patches; it is a profile of evidence strength. A patch with positive build, functionality, and original exploit outcomes but negative variant and root-cause outcomes represents a different risk than a patch that fails at build time. A validation report that separates these cases gives developers more actionable information.

Table I summarizes the validation dimensions used in the study. The table also identifies the analytical question associated with each dimension. This structure is intended to make the evaluation reproducible and auditable. A security team may implement the same dimensions with different tools, but the reporting logic remains stable: each layer contributes a distinct piece of evidence, and the full acceptance label should be reserved for patches that satisfy all required security-aligned layers.

JAIAA **RESEARCH ARTICLE**
Journal of AI Analytics and Applications **Vol. 1, No. 2 (2023), pp. 88–103**
ISSN 3067-7386 | Open Access | Peer-Reviewed **DOI: 10.63646/jaiaa.2023.010205**

Table I. Validation dimensions and analytical interpretation.

Validation dimension	Primary evidence	Analytical interpretation	Typical failure pattern
Build validity	Patched program compiles, loads, and executes in the controlled environment	Minimum executable quality; not a security claim	Invalid repair or extraction failure
Functionality preservation	Benign tests remain satisfied after the patch	The patch avoids damaging observed legitimate behavior	Functionality-breaking fix
Original PoC blocking	The demonstrated exploit no longer triggers the vulnerability	Direct evidence against one known exploit path	Unblocked vulnerability
Exploit-variant resistance	Semantically equivalent variants fail under controlled tests	Behavioral generalization beyond the demonstration	PoC-overfitted or variant-bypassable fix
Root-cause conformance	CWE-specific repair rubric is satisfied	Semantic evidence that the vulnerability mechanism has been addressed	Symptom-level or incomplete root-cause fix
Static-warning rechecking	Analyzer signal disappears, persists, or changes after patching	Scalable diagnostic signal, not a final oracle	Warning-overfitted or analyzer-facing fix
Regression safety	No detected new weakness or weakened security behavior	Evidence that the repair did not create a security-relevant side effect	Regression-unsafe fix

Table I emphasizes that each validation layer answers a different question. A robust reporting system should therefore keep the layers separate until the final interpretation stage. This separation also makes benchmark results easier to compare because a reader sees whether a reported improvement comes from better exploit blocking, better root-cause conformance, fewer regressions, or merely higher build success.

IV. CONTROLLED DATASET AND MEASURED VARIABLES

The controlled dataset contains 400 candidate patch records. The records are divided across two LLM configurations, four CWE classes, and five repair-prompt settings. Each patch record contains the vulnerable case identifier, vulnerability class, prompt setting, model configuration, extracted patch status, validation-layer outcomes, and final label. The dataset is designed for validation analytics rather than exploit publication. It does not include executable payloads or raw attack materials. This choice supports reproducibility of aggregate analysis while avoiding disclosure of materials that would be inappropriate for open redistribution. LLM-based program repair studies support the need to separate generation capability from validation reliability (Xia and Zhang,2023). Research on software naturalness explains why LLMs can produce plausible patches that still require semantic validation (Hindle et al.,2016).

The four vulnerability classes were selected because they differ in repair semantics and because they support both behavioral and root-cause evaluation. SQL injection provides a clear contrast between payload filtering and safe query construction. Path traversal provides a clear contrast between string-level blocking and canonical path confinement. Cross-site scripting requires context-aware output handling. Missing authorization requires correct placement of access-control checks at a trust boundary. These differences allow the study to examine whether patch overfitting varies by vulnerability class rather than assuming a uniform repair problem. Competition-level code-generation results demonstrate that high coding performance does not automatically imply security adequacy (Li et al.,2022). Machine learning for big code provides the methodological basis for treating source code as data while still respecting program semantics (Allamanis et al.,2018).

The five prompt settings are not treated as an optimization exercise. Instead, they represent realistic ways developers might ask an LLM to repair code: a generic repair request, a security-aware request, a root-cause-aware request, a variant-aware request, and a static-warning-guided request. The prompt setting is recorded because it may influence repair behavior. However, the study treats prompt design as guidance, not proof. Even an explicitly root-cause-aware prompt must be followed by independent validation evidence. Financial and information-system applications of quantum algorithms reinforce the importance of auditable model-driven decision evidence (Lu and Yang,2024). Management analytics research supports the idea that technical evidence should be structured for decision making, not only for model scoring (Lu,2021).

Table II reports the principal variables in the patch-level dataset. A major benefit of this schema is that it allows multiple forms of analysis without changing the underlying records. The same data support acceptance-rate analysis, oracle-divergence analysis, CWE-level stratification, prompt-level comparison, model comparison, and qualitative review of failure patterns. This is a practical advantage for benchmark maintainers because it separates data collection from later analytical interpretation.

Table II. Patch-level dataset schema for validation analytics.

Variable group	Example field	Recorded value	Use in analysis
Case metadata	case_id, cwe_class, scenario_type	CWE89-07; SQL injection; search	Supports stratified analysis by

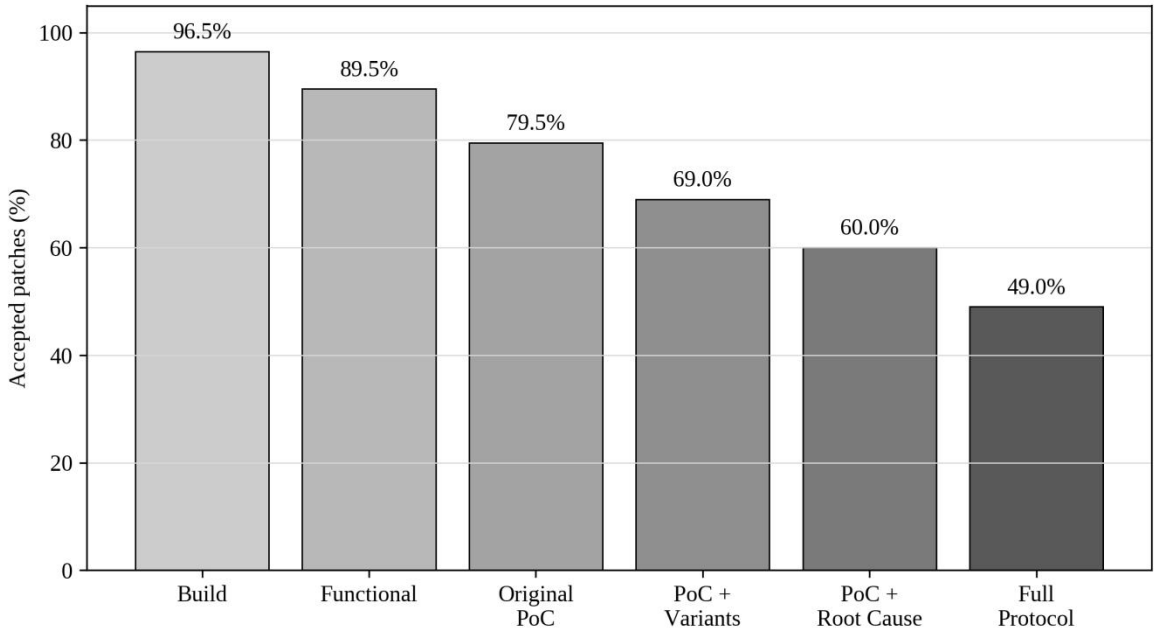
		endpoint	vulnerability class and scenario
Patch generation	model_config, prompt_setting, response_id	Model A; root-cause-aware; generated response hash	Supports prompt and model comparison
Patch extraction	extraction_status, build_status	applied; build passed	Separates invalid edits from security failures
Behavioral evidence	functional_pass, poc_blocked, variants_blocked	1, 1, 0	Supports oracle-divergence and exploit-variant analysis
Semantic evidence	root_cause_conformant, static_warning_status	0; warning disappeared	Identifies warning-overfitting and incomplete repair
Safety evidence	regression_safe, final_verdict	1; PoC-overfitted fix	Supports operational rejection and failure-mode reporting

The variables in Table II also support privacy-aware and misuse-aware reporting. Raw attack payloads are unnecessary for aggregate analytics. A benchmark maintainer may publish sanitized categories, validation outcomes, and reproducibility scripts while keeping exploit artifacts out of public release. This balance is important for security research that aims to improve evaluation without increasing operational risk.

V. RESULTS AND DATA ANALYSIS

The first analysis examines how acceptance changes as validation becomes more demanding. Figure 2 reports the primary model configuration across successive layers. Build validity is high, indicating that most generated patches are syntactically executable. Functional preservation is also high, showing that many patches maintain observed benign behavior. However, the acceptance rate falls sharply once security-aligned evidence is enforced. Original proof-of-concept blocking accepts 79.5% of patches, while full-protocol acceptance reaches 49.0%. Program-synthesis research indicates that prompt-driven code generation should be treated as probabilistic candidate production rather than final assurance (Austin et al.,2021). Studies comparing deep models for source code warn against assuming that more complex models always produce more reliable software evidence (Hellendoorn and Devanbu,2017).

This drop is analytically important because it shows that the original exploit is an incomplete proxy for security repair. A patch that blocks the demonstration may still fail variant resistance, fail root-cause conformance, or introduce a regression. In practical terms, this means that a security team should not equate a green proof-of-concept replay with a fully trustworthy repair. The proof-of-concept remains necessary: a patch that fails the original exploit test is not a credible fix. Yet the proof-of-concept should be treated as a gateway signal rather than a final decision signal. Open code-generation models make validation analytics increasingly important because repair proposals can be produced at large scale (Nijkamp et al.,2022). Tree-structured neural code models show that syntactic structure can be useful for understanding code but remains insufficient for security validation alone (Mou et al.,2016).



RESEARCH ARTICLE
Vol. 1, No. 2 (2023), pp. 88–103
DOI: 10.63646/jaiaa.2023.010205

Figure 2. Acceptance-rate decline as validation layers become stronger.

The same pattern appears in the oracle-divergence analysis. Table III reports paired comparisons in which a weaker oracle accepts a patch and a stronger oracle rejects it. The largest divergence is observed between functional passing and full-protocol acceptance. This indicates that preserving sampled benign behavior is a weak security indicator. More central to this article, the original proof-of-concept-to-full comparison shows substantial divergence, and the static-warning-to-root-cause comparison shows that analyzer-facing success often fails to establish semantic repair.

The divergence results provide a more informative view than a single final success rate. A final rate of 49.0% tells the reader that less than half of the primary candidate patches satisfy the full validation profile. The divergence rows explain why. Some patches fail because they are too closely fitted to the original exploit. Others fail because the static warning disappears without sufficient root-cause evidence. Others preserve functionality but fail stronger security evidence. These categories require different responses from tool builders and developers. Audit-oriented blockchain research is relevant because AI patch validation also requires traceable evidence and independent review (Wu et al.,2025). Bibliometric analytics research is relevant because validation analytics also depends on organizing heterogeneous evidence across studies and artifacts (Lu et al.,2024b).

Table III. Paired oracle-divergence statistics in the primary validation profile.

Weak-to-strong comparison	Accepted by weak oracle	Rejected by stronger oracle	Divergence	Interpretation
Functional pass vs. full protocol	179	81	45.3%	Observed benign behavior is insufficient as security evidence
Original PoC vs. exploit variants	159	21	13.2%	Some patches block the demonstration but fail related attacks
Original PoC vs. root-cause conformance	159	39	24.5%	Some exploit-blocking patches lack class-specific repair evidence
Original PoC vs. full protocol	159	61	38.4%	Original exploit blocking overestimates genuine acceptance
Static warning disappearance vs. root-cause conformance	160	69	43.1%	Analyzer-facing success often diverges from semantic repair

The most important result in Table III is not simply that the full protocol is stricter. The analytical value is that the table localizes disagreement. A system with high proof-of-concept blocking and high variant resistance but low root-cause conformance should be improved differently from a system with high root-cause conformance but poor regression safety. Validation analytics turns a single failure label into a diagnostic profile.

VI. DISCUSSION AND PRACTICAL IMPLICATIONS

The findings support a layered approach to AI patch validation analytics. The goal of such an approach is not to reject automation, but to make automation safer and more measurable. LLMs are useful generators of candidate repairs, especially for routine vulnerabilities and rapid triage. Their outputs, however, should enter a validation workflow that distinguishes apparent success from stronger security evidence. Without that distinction, organizations risk accepting patches that appear to close a vulnerability while leaving equivalent attack paths open. Instruction-following improvements affect the way developers elicit repairs, but instruction alignment does not substitute for security validation (Ouyang et al.,2022). Code-smell detection research demonstrates that learned software-quality signals require careful calibration before operational use (Liu et al.,2020).

Exploit-variant resistance should become a standard component of LLM-based vulnerability repair evaluation. Variants need not be unlimited or adversarially exhaustive. Even a bounded set of class-specific variants reveals whether the patch generalizes beyond the demonstration. For SQL injection, variants may cover parameter structure, comments, encodings, and query contexts. For path traversal, variants may cover normalization, separator changes, and encoded traversal. For XSS, variants may cover output contexts and event-based forms. For authorization, variants may cover object ownership, role transitions, and alternative endpoints. These tests should remain controlled and sanitized. Few-shot language-model research explains why LLMs can generalize repair patterns from context, while also leaving uncertainty about unseen variants (Brown et al.,2020). Name-based bug detection shows that machine-learning signals can uncover suspicious code patterns but should

RESEARCH ARTICLE
Vol. 1, No. 2 (2023), pp. 88–103
DOI: 10.63646/jaiaa.2023.010205

be interpreted as diagnostic evidence (Pradel and Sen,2018).

Root-cause conformance should be treated as a distinct semantic layer. A variant test is behavioral: it observes whether a class of attacks fails. A root-cause rubric is explanatory: it asks whether the patch mechanism is appropriate for the vulnerability class. The two layers reinforce each other. Behavioral tests detect bypasses that a rubric may miss. Root-cause assessment identifies superficial fixes that happen to pass a finite set of variants. Together, they provide a stronger basis for accepting or rejecting generated patches. Information-systems blockchain research shows that technical controls and organizational procedures should be evaluated together (Lu,2022). Decision-making analytics research supports the conversion of validation outputs into actionable risk indicators for teams (Lu et al.,2024c).

Static analysis remains valuable in this framework, but its role should be carefully defined. A warning that disappears after patching is a useful signal, particularly in large codebases where automated scanning is routine. However, warning disappearance is not the same as vulnerability removal. Analyzer-facing edits may hide a pattern, suppress a rule, move code in a way that avoids detection, or remove functionality. In a validation analytics workflow, static analysis contributes one column of evidence and should be interpreted in relation to exploit-based and root-cause evidence. Neural repair research supports the use of code-aware representations, but it also shows the need for downstream correctness checks (Jiang et al.,2021). Neural machine translation repair studies show that learned patch generation can capture common edits while still needing independent correctness assessment (Tufano et al.,2019).

The practical workflow implied by the study has four stages. First, a generated patch should pass build and functionality checks. Second, it should block the original proof-of-concept exploit in a controlled environment. Third, it should be tested against sanitized exploit variants designed for the vulnerability class. Fourth, it should be assessed against root-cause conformance and regression-safety criteria. A patch rejected at any stage should not be silently discarded; instead, the rejection reason should be recorded so that model developers and security engineers learn which failure mode is most common. Joint localization-and-repair models make patch provenance important because validation failures may originate from either fault localization or edit generation (Ye et al.,2022). Sequence-to-sequence repair systems provide a precedent for end-to-end repair, but their outputs still require layered validation (Chen et al.,2021b).

The approach also has implications for repair interfaces. A developer-facing tool should not merely report "patch accepted" or "patch failed." It should present a validation profile: build passed, functional tests passed, original proof-of-concept blocked, two variants failed, root-cause conformance incomplete, no regression detected. Such a profile is more useful than a binary label because it explains the next action. A variant failure suggests the need for broader input handling. A root-cause failure suggests a semantic redesign. A regression failure suggests that the repair changed business logic or weakened a security property elsewhere. IoT blockchain-security research shows that software fixes frequently operate within multi-layered trust environments (Xu et al.,2021). Early 6G research illustrates the broader movement toward software-defined infrastructure where reliable automated repair will become increasingly consequential (Lu and Zheng,2019).

For benchmark designers, the results suggest that datasets should include not only vulnerable code and reference patches but also layered validation artifacts. A benchmark with only one oracle risks encouraging systems that overfit to that oracle. A benchmark with multiple evidence layers reveals how success changes under stronger evaluation. This makes model comparisons more transparent and reduces the likelihood that reported improvements merely reflect oracle permissiveness. Vulnerability-fix datasets demonstrate the value of linking code changes with vulnerability metadata for repair analytics (Bhandari et al.,2021). Program-repair libraries demonstrate the value of reproducible repair infrastructure for comparing candidate patches (Martinez and Monperrus,2016).

For DevSecOps managers, the findings point to a governance issue. LLM-generated patches should be integrated into software delivery pipelines only when the pipeline contains independent validation controls. The organization should define what kind of evidence is required before a generated repair is merged. For high-risk components, the threshold should include exploit variants, root-cause review, and regression-safety checks. For lower-risk components, an organization may choose a lighter profile, but the evidence gap should be explicit rather than hidden behind a single success metric. Large vulnerability corpora make it possible to analyze repair behavior across CWE classes, but dataset structure can also bias measured outcomes (Fan et al.,2020). Systematic APR studies show that automatic repair success depends heavily on the definition of acceptance evidence (Le Goues et al.,2012).

JAIAA **RESEARCH ARTICLE**
Journal of AI Analytics and Applications **Vol. 1, No. 2 (2023), pp. 88–103**
ISSN 3067-7386 | Open Access | Peer-Reviewed **DOI: 10.63646/jaiaa.2023.010205**

The analysis also supports an audit trail for AI-assisted repair. Each generated patch should be linked to the prompt, model configuration, raw response, extracted patch, validation outputs, human review decision, and final disposition. Such an audit trail enables post-incident analysis, benchmarking, continuous improvement, and compliance review. It also supports responsible AI governance by documenting how automated code changes were assessed before acceptance. AI research surveys provide a broad foundation for treating repair validation as an AI analytics problem rather than a purely programming-language problem (Zhang and Lu,2021). Industrial IoT surveys show that code-level security fixes must be evaluated in the context of connected operational systems (Xu et al.,2014).

Implementation should also prioritize validation controls according to both security-decision impact and automation feasibility. Figure 3 maps five divergence indicators in this two-dimensional space. Controls in the upper-right area deserve early investment because they are both operationally automatable and highly relevant to accepting or rejecting a generated patch. Warning-to-root-cause divergence and proof-of-concept-to-full divergence are especially important because they reveal cases where familiar tooling signals create false confidence. New vulnerable-code datasets show that coverage across projects and CWEs remains a central challenge for generalizable security models (Chen et al.,2023). Symbolic patch synthesis research supports root-cause-oriented reasoning because successful repair may require constraints rather than local edits (Mechtaev et al.,2016).

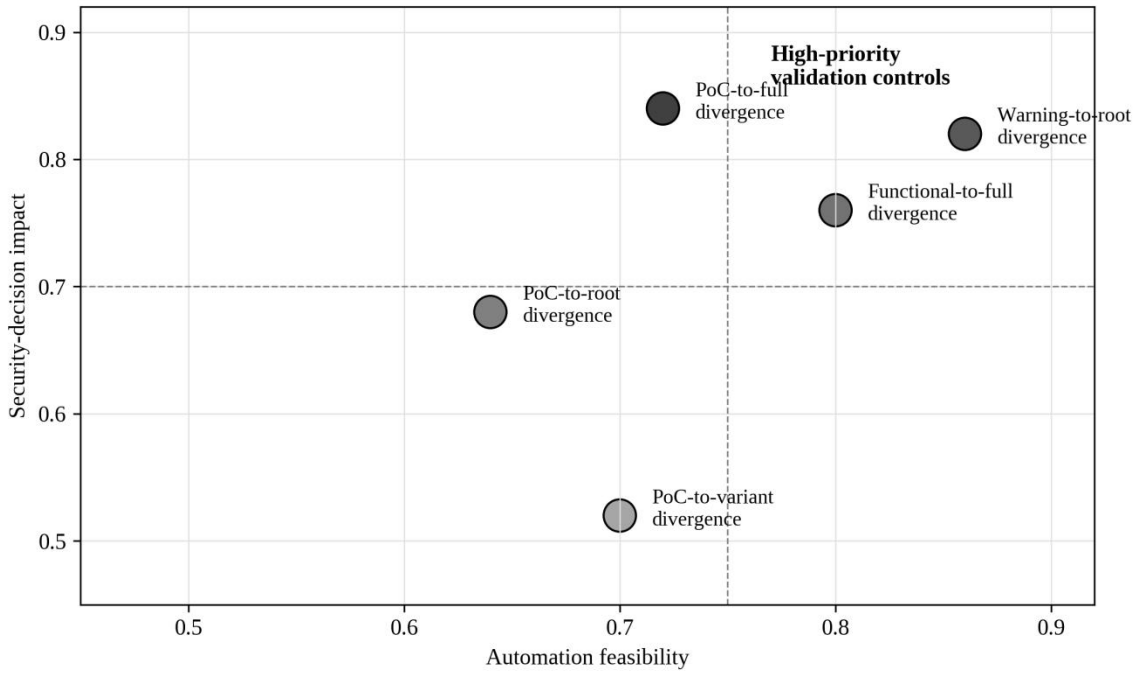


Figure 3. Priority map for validation controls in AI-assisted patch governance.

Figure 3 should not be read as a universal ranking of all security controls. It is an implementation heuristic for teams that must decide where to invest first. A small organization may begin with automated proof-of-concept replay and simple class-specific variant checks. A mature organization may add static-to-root-cause comparison, regression-safety monitoring, human review queues, and benchmark dashboards that compare model behavior over time. In both cases, the evidence profile should remain transparent to developers and security reviewers.

The governance implications extend beyond a single repair task. Once LLM-generated patches enter a continuous integration environment, each validation failure becomes a data point for system improvement. A repeated pattern of path-traversal variant failures may indicate that prompt templates overemphasize literal string blocking. A repeated pattern of SQL injection root-cause failures may indicate that model outputs need stronger examples of parameterized query construction. A repeated pattern of authorization regression may indicate that the model lacks repository-level context about roles and object ownership.

Validation analytics also supports risk-based routing. Low-risk patches that pass all layers may proceed to ordinary code review.

RESEARCH ARTICLE
Vol. 1, No. 2 (2023), pp. 88–103
DOI: 10.63646/jaiaa.2023.010205

Patches that pass the original exploit but fail variants should be routed to a security engineer with a variant-bypass explanation. Patches that remove a static warning but fail root-cause conformance should be routed to an analyst familiar with the relevant CWE class. Patches that introduce regressions should be routed back to the developer or model agent for redesign rather than manual cosmetic adjustment. Code property graphs reinforce the value of structural and semantic evidence when assessing whether a patch addresses the vulnerable path (Yamaguchi et al.,2014). Patch-correctness research shows that even test-accepted patches can be semantically questionable under stronger criteria (Xiong et al.,2017).

Table VI. Governance controls mapped to common AI patch-validation failure modes.

Failure mode	Primary evidence pattern	Recommended governance control	Expected benefit
Exploit-specific repair	Original PoC blocked but variants fail	Require class-specific variant suite before acceptance	Reduces bypassable fixes that only address the demonstration
Symptom-level repair	Exploit blocked but root-cause rubric fails	Require CWE-specific semantic review or rule-based conformance check	Discourages superficial filters and incomplete mechanisms
Analyzer-facing repair	Static warning disappears but stronger checks fail	Treat static rechecking as intermediate evidence only	Prevents warning disappearance from being used as final proof
Functionality-breaking repair	Security signal improves while benign tests fail	Require functional regression gate before security acceptance	Avoids repairs that disable legitimate behavior
Regression-unsafe repair	Earlier layers pass but new weakness is detected	Require before/after security regression scan and targeted review	Prevents one vulnerability fix from creating another risk
Model-specific overconfidence	Model shows high weak-oracle success but low full acceptance	Rank models by layered profile instead of single success rate	Avoids model selection based on permissive oracles

The controls in Table VI transform patch validation from a pass/fail exercise into a governance routine. The table is also useful for audit documentation. Instead of stating that a generated patch was rejected, a team records that the patch was rejected because it was exploit-specific, analyzer-facing, incomplete, functionality-breaking, or regression-unsafe. This vocabulary improves communication between model engineers, application developers, security analysts, and release managers.

Another benefit is longitudinal measurement. Over several releases, an organization may track whether the proportion of PoC-overfitted patches decreases after prompt changes, whether root-cause conformance improves after retrieval augmentation, or whether regression failures increase when agents edit multiple files. Such trends are difficult to see in a binary success metric. They become visible when the validation record contains separate evidence columns and a stable failure taxonomy. Broad AI evolution studies show why validation methodology must evolve alongside increasingly capable model families (Lu,2019). Industrial IoT research reinforces the need for security validation to extend beyond a single code fragment (Lu,2017b).

The analytics perspective therefore links research evaluation with operational practice. In research, the same framework improves benchmark comparability. In practice, it guides acceptance gates, triage queues, audit trails, and model-improvement cycles. This dual role is especially important for LLM-based software engineering because model outputs are probabilistic and model behavior changes over time. A validation system should be resilient to such change by grounding acceptance in external evidence rather than model self-assessment. Deep vulnerability detection systems show that learned security signals can identify suspicious code, but they do not alone prove repair correctness (Li et al.,2018). Repository-curation research shows why benchmark provenance and project quality matter when interpreting repair results (Munaiah et al.,2017).

VII. LIMITATIONS AND FUTURE RESEARCH

The study has several limitations. First, the dataset is controlled and designed for validation analytics. It should not be interpreted as a statistically representative sample of all vulnerabilities in production software. Real repositories contain dependencies, build variability, framework-specific conventions, incomplete tests, and security assumptions that are difficult to reproduce in a compact benchmark. The main generalizable finding is methodological: weak-oracle acceptance may diverge from stronger security-aligned evidence and should therefore be measured. Representation-learning studies for vulnerability detection support the use of learned features as auxiliary diagnostics in patch analytics (Russell et al.,2018). Machine-learning surveys of vulnerability discovery support the article's distinction between detection evidence and repair evidence (Ghaffarian and Shahriari,2017).

Second, exploit variants are finite. No controlled evaluation fully enumerates all possible attack forms. A patch that passes the variants in this study may still fail under a stronger adversarial search. Future work should combine template-based variants with fuzzing, symbolic execution, grammar-based mutation, dynamic taint tracking, and repository-aware analysis. Such

RESEARCH ARTICLE
Vol. 1, No. 2 (2023), pp. 88–103
DOI: 10.63646/jaiaa.2023.010205

extensions would improve coverage while preserving the safety requirement that exploit artifacts remain local, controlled, and non-deployable. Industrial information integration research suggests that repair evidence should be organized so that it can be reused across pipelines and teams (Lu,2018).

Third, root-cause conformance depends on the quality of the rubric. The rubrics used here are explicit and vulnerability-class-specific, but they are not formal proofs of security correctness. Some legitimate repair strategies may satisfy security objectives through mechanisms not captured by a simple rubric. Future work should examine multi-rater review, expert disagreement resolution, semantic code analysis, and formal or semi-formal specifications for selected vulnerability classes. Graph-neural vulnerability models motivate variant-aware evaluation because vulnerabilities often depend on contextual program semantics (Zhou et al.,2019).

Fourth, the article evaluates validation outcomes rather than developer productivity or organizational adoption. In practice, a validation analytics system must balance security rigor with time, cost, and usability. Too many validation gates may slow emergency patching, while too few may allow incomplete repairs into production. Future studies should measure the operational trade-off between validation strength, triage speed, false rejection, false acceptance, and developer trust. Semantic vulnerability-representation frameworks demonstrate that data-flow and control-flow features remain essential for root-cause assessment (Li et al.,2021).

Fifth, the cross-model comparison is not presented as a comprehensive model ranking. Model behavior changes over time as systems are updated, and prompt settings influence generated patches. The purpose of the comparison is to show that higher weak-oracle success does not necessarily imply higher full-protocol acceptance. A broader model study should control model versions, decoding parameters, repository context, and prompt strategy more extensively. Industry 4.0 studies show that automated software decisions increasingly interact with physical, organizational, and data-governance systems (Lu,2017).

VIII. CONCLUSION

This article developed an analytics-oriented framework for evaluating LLM-based vulnerability repair through exploit-variant resistance and root-cause conformance. The core argument is that generated security patches should not be accepted only because they compile, pass functional tests, block an original proof-of-concept, or remove a static warning. These signals are useful, but they are incomplete when used alone. A reliable validation profile should distinguish build validity, benign behavior preservation, original exploit blocking, exploit-variant resistance, class-specific root-cause conformance, static-warning status, regression safety, and final full-protocol acceptance.

The data analysis shows why this distinction matters. In the primary configuration, original proof-of-concept blocking substantially exceeded full validation acceptance. CWE-level results showed different failure patterns across path traversal, SQL injection, cross-site scripting, and missing authorization. Cross-model comparison further showed that higher weak-oracle success may coexist with lower full acceptance. These findings support the need for layered reporting rather than single-oracle success claims.

For AI analytics research, the paper reframes vulnerability repair evaluation as a problem of evidence integration. The key question is not simply whether an LLM produced a patch, but whether the patch remains credible under progressively stronger security evidence. For software security practice, the article recommends validation-aware repair workflows that test generated patches against variants, assess root-cause conformance, and record rejection reasons. Such workflows make LLM-assisted repair more transparent, auditable, and safer for deployment in real software engineering environments.

AUTHOR CONTRIBUTIONS

Author	Contribution
Nurul Farhana Ismail	Conceptualization, methodology, writing - original draft, visualization, validation analytics
Hafiz Rahman	Data curation, formal analysis, software, result interpretation
Aina Syahirah Aziz	Supervision, resources, writing - review and editing, project administration

DECLARATIONS

Conflicts of interest: The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this manuscript.

Data availability: The manuscript uses sanitized patch-level validation outcomes and aggregate benchmark summaries. Executable exploit artifacts, raw attack payloads, and security-sensitive case materials are not redistributed. Aggregated analysis files and non-deployable metadata are available from the corresponding author upon reasonable request.

Funding: This research received no external funding.

Ethics statement: This study does not involve human participants, animal experiments, identifiable personal records, or live-system security testing. All validation examples are discussed at a controlled and non-deployable level.

ABOUT THE AUTHORS

Nurul Farhana Ismail is affiliated with Universiti Teknikal Malaysia Melaka, Malaysia. Her research focuses on applied AI analytics, software security evaluation, and validation-aware DevSecOps workflows.

Hafiz Rahman is affiliated with Universiti Malaysia Pahang Al-Sultan Abdullah, Malaysia. His research interests include secure software engineering, machine learning evaluation, and cyber risk analytics.

Aina Syahirah Aziz is affiliated with Universiti Malaysia Perlis, Malaysia. Her work addresses AI-assisted software engineering, security governance, and trustworthy intelligent systems.

REFERENCES

Zhang, L., Zou, Q., Singhal, A., Sun, X., & Liu, P. (2024). Evaluating large language models for real-world vulnerability repair in C/C++ code. Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics, 1-10. https://doi.org/10.1145/3643651.3659892

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, N., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30. https://doi.org/10.48550/arXiv.1706.03762

Lu, Y., & Xu, L. D. (2019). Internet of Things (IoT) cybersecurity research: A review of current research topics. IEEE Internet of Things Journal, 6(2), 2103-2115. https://doi.org/10.1109/JIOT.2018.2869847

Pearce, H., Tan, B., Ahmad, B., Karri, R., & Dolan-Gavitt, B. (2023). Examining zero-shot vulnerability repair with large language models. 2023 IEEE Symposium on Security and Privacy, 2339-2356. https://doi.org/10.1109/SP46215.2023.10179420

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. Findings of EMNLP 2020, 1536-1547. https://doi.org/10.48550/arXiv.2002.08155

Kou, G., & Lu, Y. (2025). FinTech: A literature review of emerging financial technologies and applications. Financial Innovation, 11(1), 1-34. https://doi.org/10.1186/s40854-024-00668-6

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. arXiv. https://doi.org/10.48550/arXiv.2107.03374

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow. International Conference on Learning Representations. https://doi.org/10.48550/arXiv.2009.08366

Lu, Y. (2025). The current status and developing trends of Industry 4.0: A review. Information Systems Frontiers, 27(1), 215-234. https://doi.org/10.1007/s10796-021-10221-w

Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V., & Phung, D. (2022). VulRepair: A T5-based automated software vulnerability repair. Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 935-947. https://doi.org/10.1145/3540250.3549098

OpenAI. (2023). GPT-4 technical report. arXiv. https://doi.org/10.48550/arXiv.2303.08774

Xu, R., Zhu, J., Yang, L., Lu, Y., & Xu, L. D. (2024). Decentralized finance (DeFi): A paradigm shift in the FinTech. Enterprise Information Systems, 18(9). https://doi.org/10.1080/17517575.2024.2397630

de-Fitero-Dominguez, D., Garcia-Lopez, E., Garcia-Cabot, A., & Martinez-Herraiz, J. J. (2024). Enhanced automated code vulnerability repair

using large language models. Engineering Applications of Artificial Intelligence, 138, 109291. https://doi.org/10.1016/j.engappai.2024.109291

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. Proceedings of EMNLP 2021, 8696-8708. https://doi.org/10.48550/arXiv.2109.00859

Lu, W., Lu, Y., Li, J., Sigov, A., Ratkin, L., & Ivanov, L. A. (2024). Quantum machine learning: Classifications, challenges, and solutions. Journal of Industrial Information Integration, 42, 100736. https://doi.org/10.1016/j.jii.2024.100736

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? International Conference on Learning Representations. https://doi.org/10.48550/arXiv.2310.06770

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). UniXcoder: Unified cross-modal pre-training for code representation. Proceedings of ACL 2022, 7212-7225. https://doi.org/10.48550/arXiv.2203.03850

Chen, Y., Lu, Y., Bulysheva, L., & Kataev, M. Y. (2024). Applications of blockchain in Industry 4.0: A review. Information Systems Frontiers, 26(5), 1715-1729. https://doi.org/10.1007/s10796-022-10248-7

Xia, C. S., Wei, Y., & Zhang, L. (2023). Automated program repair in the era of large pre-trained language models. 2023 IEEE/ACM 45th International Conference on Software Engineering, 1482-1494. https://doi.org/10.1109/ICSE48619.2023.00138

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092-1097. https://doi.org/10.1126/science.abq1158

Lu, Y., & Yang, J. (2024). Quantum financing system: A survey on quantum algorithms, potential scenarios and open research issues. Journal of Industrial Information Integration, 41, 100663. https://doi.org/10.1016/j.jii.2024.100663

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. arXiv. https://doi.org/10.48550/arXiv.2108.07732

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2022). CodeGen: An open large language model for code with multi-turn program synthesis. arXiv. https://doi.org/10.48550/arXiv.2203.13474

Wu, H. P., Liu, Z., Dong, H. Y., Lu, Y., & Xu, L. D. (2025). Revolutionizing internal auditing: Harnessing the power of blockchain. Enterprise Information Systems, 19(1-2). https://doi.org/10.1080/17517575.2024.2448003

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. (2022). Training language models to follow instructions with human feedback. arXiv. https://doi.org/10.48550/arXiv.2203.02155

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. arXiv. https://doi.org/10.48550/arXiv.2005.14165

Lu, Y. (2022). Implementing blockchain in information systems: A review. Enterprise Information Systems, 16(12), 1876-1907. https://doi.org/10.1080/17517575.2021.2008513

Jiang, N., Lutellier, T., Lou, Y., & Tan, L. (2021). CURE: Code-aware neural machine translation for automatic program repair. 2021 IEEE/ACM 43rd International Conference on Software Engineering, 1161-1173. https://doi.org/10.1109/ICSE43902.2021.00107

Ye, H., Martinez, M., & Monperrus, M. (2022). Neural program repair by jointly learning to localize and repair. Proceedings of the 44th International Conference on Software Engineering, 2124-2136. https://doi.org/10.1145/3510003.3510115

Xu, L. D., Lu, Y., & Li, L. (2021). Embedding blockchain technology into IoT for security: A survey. IEEE Internet of Things Journal, 8(13), 10452-10473. https://doi.org/10.1109/JIOT.2021.3060508

Bhandari, G. P., Naseer, A., & Moonen, L. (2021). CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, 30-39. https://doi.org/10.1145/3475960.3475985

Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020). A C/C++ code vulnerability dataset with code changes and CVE summaries. Proceedings of the 17th International Conference on Mining Software Repositories, 508-512. https://doi.org/10.1145/3379597.3387501

Zhang, C., & Lu, Y. (2021). Study on artificial intelligence: The state of the art and future prospects. Journal of Industrial Information Integration, 23, 100224. https://doi.org/10.1016/j.jii.2021.100224

Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. (2023). DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection. Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, 654-668. https://doi.org/10.1145/3607199.3607242

Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. 2014 IEEE Symposium on Security and Privacy, 590-604. https://doi.org/10.1109/SP.2014.44

Lu, Y. (2019). Artificial intelligence: A survey on evolution, models, applications and future trends. Journal of Management Analytics, 6(1), 1-29. https://doi.org/10.1080/23270012.2019.1570365

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. Proceedings of the Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2018.23158

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., & McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. arXiv. https://doi.org/10.48550/arXiv.1807.04320

Lu, Y. (2018). Blockchain and the related issues: A review of current research topics. Journal of Management Analytics, 5(4), 231-255. https://doi.org/10.1080/23270012.2018.1516523

Zhou, Y., Liu, S., Siow, J. K., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program

semantics via graph neural networks. Advances in Neural Information Processing Systems, 32. https://doi.org/10.48550/arXiv.1909.03496

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244-2258. https://doi.org/10.1109/TDSC.2021.3051525

Lu, Y. (2017). Industry 4.0: A survey on technologies, applications and open research issues. Journal of Industrial Information Integration, 6, 1-10. https://doi.org/10.1016/j.jii.2017.04.005

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2021). Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering, 48(9), 3280-3296. https://doi.org/10.1109/TSE.2021.3087402

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? 2013 35th International Conference on Software Engineering, 672-681. https://doi.org/10.1145/2486788.2486817

Lu, Y., & Ning, X. (2020). A vision of 6G: 5G's successor. Journal of Management Analytics, 7(3), 301-320. https://doi.org/10.1080/23270012.2020.1802622

Christakis, M., & Bird, C. (2016). What developers want and need from program analysis: An empirical study. 2016 31st IEEE/ACM International Conference on Automated Software Engineering, 332-343. https://doi.org/10.1145/2970276.2970347

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., & Jaspan, C. (2018). Lessons from building static analysis tools at Google. Communications of the ACM, 61(4), 58-66. https://doi.org/10.1145/3188720

Lu, Y., & Zheng, X. (2020). 6G: A survey on technologies, scenarios, challenges, and the related issues. Journal of Industrial Information Integration, 19, 100158. https://doi.org/10.1016/j.jii.2020.100158

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., & Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. Communications of the ACM, 53(2), 66-75. https://doi.org/10.1145/1646353.1646374

Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, 209-224. https://doi.org/10.5555/1855741.1855756

Lu, Y., Zheng, X., Li, L., & Xu, L. D. (2020). Pricing the cloud: A QoS-based auction approach. Enterprise Information Systems, 14(3), 334-351. https://doi.org/10.1080/17517575.2019.1669827

Godefroid, P., Levin, M. Y., & Molnar, D. (2012). SAGE: Whitebox fuzzing for security testing. Communications of the ACM, 55(3), 40-44. https://doi.org/10.1145/2093548.2093564

Bohme, M., Pham, V. T., & Roychoudhury, A. (2016). Coverage-based greybox fuzzing as Markov chain. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 1032-1043. https://doi.org/10.1145/2976749.2978428

Ye, Z., & Lu, Y. (2022). Quantum science: A review and current research trends. Journal of Management Analytics, 9(3), 383-402. https://doi.org/10.1080/23270012.2022.2089064

Lemieux, C., & Sen, K. (2018). FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 475-485. https://doi.org/10.1145/3238147.3238176

Fioraldi, A., Maier, D., Eissfeldt, H., & Heuse, M. (2020). AFL++: Combining incremental steps of fuzzing research. Proceedings of the 14th USENIX Workshop on Offensive Technologies. https://doi.org/10.48550/arXiv.2004.14375

Zheng, X. R., & Lu, Y. (2022). Blockchain technology: Recent research and future trend. Enterprise Information Systems, 16(12), 1939895. https://doi.org/10.1080/17517575.2021.1939895

Pham, V. T., Bohme, M., Santosa, A. E., Caciulescu, A. R., & Roychoudhury, A. (2017). Smart greybox fuzzing. IEEE Transactions on Software Engineering, 45(2), 198-214. https://doi.org/10.1109/TSE.2018.2866057

Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. Proceedings of the 38th International Conference on Software Engineering, 297-308. https://doi.org/10.1145/2884781.2884804

Lu, Y., Sigov, A. S., Ratkin, L., Ivanov, L. A., & Zuo, M. (2023). Quantum computing and industrial information integration: A review. Journal of Industrial Information Integration, 35, 100511. https://doi.org/10.1016/j.jii.2023.100511

Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. (2016). On the naturalness of software. Communications of the ACM, 59(5), 122-131. https://doi.org/10.1145/2902362

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), Article 81. https://doi.org/10.1145/3212695

Lu, Y. (2021). Technological innovation and the emergence of a new interdisciplinary field: Management Analytics. Nanotechnologies in Construction, 13(3), 181-192. https://doi.org/10.15828/2075-8545-2021-13-3-181-192

Hellendoorn, V. J., & Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 763-773. https://doi.org/10.1145/3106237.3106290

Mou, L., Li, G., Zhang, L., Wang, T., & Jin, Z. (2016). Convolutional neural networks over tree structures for programming language processing. Proceedings of the 30th AAAI Conference on Artificial Intelligence, 1287-1293. https://doi.org/10.1609/aaai.v30i1.10139

Lu, Y., Ivanov, L. A., Wang, F., Pisarenko, Z. V., & Ye, C. (2024). Management analytics: A bibliometric analysis. Nanotechnologies in Construction, 16(3), 257-266. https://doi.org/10.15828/2075-8545-2024-16-3-257-266

Liu, H., Jin, Z., Xu, C., & Bu, Y. (2020). Deep learning based code smell detection. IEEE Transactions on Software Engineering, 47(9), 1811-

1837. https://doi.org/10.1109/TSE.2019.2936376

Pradel, M., & Sen, K. (2018). DeepBugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages, 2(OOPSLA), Article 147. https://doi.org/10.1145/3276517

Lu, Y., Pisarenko, Z. V., Yang, L., & Ye, C. (2024). Advancing decision-making: The role of management analytics in modern business practices. Nanotechnologies in Construction, 16(5), 431-440. https://doi.org/10.15828/2075-8545-2024-16-5-431-440

Tufano, M., Watson, C., Bavota, G., Poshyvanyk, D., & White, M. (2019). An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology, 28(4), Article 19. https://doi.org/10.1145/3340544

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L. N., Poshyvanyk, D., & Monperrus, M. (2021). Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering, 47(9), 1943-1959. https://doi.org/10.1109/TSE.2019.2940179

Lu, Y., & Zheng, X. (2019). 6G: A survey on technologies, scenarios, challenges, and the related issues. arXiv. https://doi.org/10.48550/arXiv.1909.08364

Martinez, M., & Monperrus, M. (2016). ASTOR: A program repair library for Java. Proceedings of the 25th International Symposium on Software Testing and Analysis, 441-444. https://doi.org/10.1145/2931037.2948705

Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. 2012 34th International Conference on Software Engineering, 3-13. https://doi.org/10.1109/ICSE.2012.6227211

Xu, L. D., He, W., & Li, S. (2014). Internet of Things in industries: A survey. IEEE Transactions on Industrial Informatics, 10(4), 2233-2243. https://doi.org/10.1109/TII.2014.2300753

Mechtaev, S., Yi, J., & Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. Proceedings of the 38th International Conference on Software Engineering, 691-701. https://doi.org/10.1145/2884781.2884807

Xiong, Y., Liu, X., Zeng, M., Zhang, L., & Huang, G. (2017). Identifying patch correctness in test-based program repair. Proceedings of the 40th International Conference on Software Engineering, 789-799. https://doi.org/10.1145/3180155.3180182

Lu, Y. (2017). Cyber physical system (CPS)-based Industry 4.0: A survey. Journal of Industrial Integration and Management, 2(3), 1750014. https://doi.org/10.1142/S2424862217500142

Munaiah, N., Kroh, S., Cabrey, C., & Nagappan, M. (2017). Curating GitHub for engineered software projects. Empirical Software Engineering, 22(6), 3219-3253. https://doi.org/10.1007/s10664-017-9512-6

Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys, 50(4), Article 56. https://doi.org/10.1145/3092566