Journal of Kyzylorda Scholarly Review

P-ISSN:3006-0265 | E-ISSN:3007-4681. Published by the Open University of Kyzylorda

https://bulletin.ouk.kz/index.php/bulletin/index

CONFIDENCE-DRIVEN GRAPH-OF-THOUGHT RETRIEVAL-AUGMENTED AGENTS FOR VERIFIABLE VULNERABILITY PATCH GENERATION IN MULTI-LANGUAGE LEGACY CODEBASES

Nibras Hmaizah[1], Roaa Ghanim[2]

General Directorate of Education in Al- Qadisiyah Govemorate, Diwaniyah, Iraq
nibras.hmaizah.alshibani@ec.edu.iq[1], roaa.ghanim11@ec.edu.iq[2]

Abstract: It is much more difficult to automatically repair security vulnerabilities in legacy software that is multi-language, than it is to detect them, for an automated patch that successfully compiles does not necessarily improve the security, functionality, or lack of intrusiveness of the software. Most current large language-model (LLM) repair systems are prone to producing grammatically correct but unverified patches, lacking evidence of what edits it has made, and giving inaccurate confidence estimates, and are rarely tested across languages under leakage control. We introduce UC-GoT-RAG, a framework consisting of an agentic architecture, vulnerability-aware hybrid retrieval, an evidence-linked Graph-of-Thought (GoT) reasoning representation, specialised analysis/generation/verification agents, and a patch-specific uncertainty-decomposition and calibration mechanism governing selective acceptance, abstention, or human escalation. Each patch is put through a rigorous staged verification process including application, build, functional and regression testing, neutralising the exploit or proof of vulnerability, security testing, and static analysis, ensuring correctness is executable, not textual. On a leakage-controlled five-language corpus collected from public commits which fixed CVE bugs, UC-GoT-RAG achieves a Verifiable Correct Patch Rate of 46.9%, which is 9.7 points higher than the best controlled baseline, and also drops the Expected Calibration Error from 0.164 to 0.041, reducing incorrect-patch acceptance at a fixed coverage by half. The most significant gains in retrieval and graph reasoning are in rare CWE families and high-legacy-burden projects. The benchmark split, prompts and configurations are released along with the patch validation harness for reproduction.

Keywords: Automated vulnerability repair, large language models, retrieval-augmented generation, graph-of-thought reasoning, uncertainty calibration, selective prediction, legacy software, software security.

Introduction

A. Background and Motivation

Legacy software is not only vital to the economy, but it's also particularly difficult to secure. Large systems that have been built over years, are often dependent on outmoded application programming interfaces (APIs), outdated dependencies, flaky test suites and outdated toolchains that no longer produce clean builds easily. If there's a vulnerability discovered in such a system, the problem of creating a safe patch is not so much about finding the bug, it's about understanding enough of the context of the bug to modify its behavior without causing the rest of the system to do so as well. The research of automated program repair has consequently turned into a lively field within the software engineering research community Automatic program repair (APR) is crucial to im prove software reliability. Recently, neural machine translation (NMT) techniques have been used to fix software bugs auto matically. While promising, these approaches have two major limitations. Their search space often does not contain the correct fix, and their search strategy ignores software knowledge such as strict code syntax. Due to these limitations, existing NMT-based techniques underperform the best template-based approaches [28–30], and Withtherecentadvancesindeeplearning(DL), anincreasing number of APR techniques have been proposed to leverage neural networks to learn bug-fixing patterns from massive open source code repositories.uch learning-based techniques usually treat APR as a neural machine translation (NMT) task, where buggy code snippets (i.e., source language) are translated into fixed code snippets (i.e., target language) automatically [34].

Vulnerability repair is much more challenging than vulnerability detection. Detection: Is there a defect? Repair: Is there a specific, deployable edit to do to remove the defect and maintain all intended behaviours? A patch can be approved by the compiler and still leave the weakness in the code open to attack, or it could cover an attack with a reduction in legitimate features.A patch can pass the compiler test and leave a weakness unpatched or it can patch the attack and break features the users depend on. All of these failure modes are often hidden: Developer test suites check for expected behavior, not adverse resistance, and a patch that passes all tests that are currently available may still have an unknown failure mode. Multi-language legacy repositories add to the difficulty as APIs that are no longer used, poor tests, antiquated dependencies, build problems, and interdependence between files all apply to each language, and the reasoning behind the successful fix in one language is not always applicable in another.

B. Research Problem

Researchers have made signi cant progress in automating the software development process in the past decades. Automated tech niques for issue summarization, bug reproduction, fault localiza tion, and program repair have been built to ease the workload of developers, The automated vulnerability repair is traditionally divided into three phases: Vulnerability analysis, patch generation, and patch validation, a valid framework should be able to cover all three phases and not only optimise patch generation in isolation. Agentic pipelines [20, 21, 22] and prompting-centric systems**,** an LLM-based vulnerability repair technique based on reasoning and patch validation feedback. VRpi lot, uses a chain-of-thought prompt to reason about a vulnera bility prior to generating patch candidates,and iteratively refines prompts according to the output of external tools (e.g., compiler, code sanitizers, test suite, etc.) on previously-generated patches [23, 24, 25] As software

vulnerabilities grow in volume and complexity, that have been developed recently with LLM models show that models can generate plausible patches but have four common deficiencies. First, even though multiple semantically correct patches can be textually different . [1, 3] Confidence calibration– the problem of predict ing probability estimates representative of the true correctness likelihood– is important for classification models in many applications, some report textual similarity to a developer patch, not executable verification. Secondly, the changes they make are rarely tied to clear evidence, the reasoning is not transparent and it's difficult to check. Thirdly, they are poorly calibrated, which means that a practitioner cannot be confident that a patch is being trusted by an automated system at a safe time [11, 14]. There are also four primary limitations to evaluation: (1) it is not often translated into multiple languages, (2) it does not often account for training-data leakage, which makes its results look better than they actually are.

C. Research Gap

To the best of our knowledge, no existing method has been able to combine all of the above in one pipeline.No existing method we are aware of has been able to combine the above in one pipeline. Prior fixes [10, 32] Language models are increasingly being deployed for general problem solving across a wide range of tasks, but are still confined to token-level, left-to-right decision-making processes during inference This means they can fall short in tasks that require exploration, strategic lookahead, or where initial decisions play a pivotal role are grounded, but not in a way that provides calibrated abstention; prior fixes are typically not deliberative, ToT allows LMs to perform deliberate decision making by considering multiple different reasoning paths and self-evaluating choices to decide the next course of action, as well as looking ahead or backtracking when necessary to make global choices [6, 7]) and, as in prior calibration research [11, 12, 13, 31], calibrated abstention is rarely the focus of the calibration problem. UC-GoT-RAG aims to fill this double void.

D. Proposed Solution

We propose UC-GoT-RAG, which brings together the following components: (i) vulnerability-aware hybrid retrieval, which combines dense, sparse similarity and CWE compatibility with language filtering; (ii) an evidence-linked Graph-of-Thought representation, where nodes for vulnerability evidence, root cause hypotheses, constraints, retrieved patterns, candidate patches, and verifier feedback are explicitly typed; (iii) specialised analysis, generation, and verification agents orchestrated by a policy agent; (iv) iterative patch refinement based on verifier feedback; (v) decomposition of the uncertainty principle into retrieval, localisation, reasoning, patch, verification and distributional components, and (vi) a selective policy that accepts, abstains, or escalates a patch, depending on calibrated confidence and required verification gates.

E. Research Questions and Hypotheses

These 5 research questions will be explored. RQ1: Can UC-GoT-RAG improve patch quality compared to controlled baselines with matched budgets with evidence that the patch is verifiably correct? RQ2: What is the effectiveness of multi-stage verification for rejecting plausible, but incorrect patches, compared to compilation-only verification? RQ3: Is uncertainty decomposition a way to get calibrated confidence that enables safe selective patching? RQ4: How strong is the framework in different languages and legacy-burden strata? RQ5: What are the factors that affect the performance, and what is the price? These map to the following

hypotheses: H1, The Verifiable Correct Patch Rate (VCPR) of UC-GoT-RAG outperforms any of the controlled baselines; H2, using a calibration leads to significant reduction in false patch acceptance rate at the same coverage rate; H3, retrieval and graph reasoning yields larger gains for CWE categories that are rare and projects that have a high-legacy-burden; H4, multi-tool verification improves security correctness while increasing inference cost.

F. Contributions

In this paper, we make five contributions: (1) The UC-GoT-RAG architecture that integrates retrieval and graph reasoning, agentic roles and calibrated selection; (2) A structured, evidence-based patch-reasoning graph that enables traceability of patch decisions; (3) A patch-specific uncertainty-decomposition and selective-acceptance mechanism; (4) A multilingual, leakage-resistant experimental protocol with executable verification; and (5) A reproducible implementation, including the benchmark split, prompts, configurations, and the patch-validation environment.

Related Work

A. Automated Program and Vulnerability Repair

There are two types of classical repair: search-based and template-based,and Structural differencing algorithms and delta debugging reduce the difference between this variant and the original program to a minimal repair. repair (e.g., GenProg [30]), and code-aware neural machine translation, e.g., CURE [28]. that transfer learning works well for repairing security vulnerabilities in C compared to learning on a small dataset [2] , we propose VulRepair, a T5-based au tomated software vulnerability repair approach that leverages the pre-training and BPE components to address various technical lim itations of prior work. [1] (extended by VulMaster [3] to widen the input range and incorporate CWE expert knowledge) arose as a result of vulnerability-specific neural repair. A common problem throughout this line is that patches can be "overfitted" to the developer edit, and perfect-prediction or exact-match scores give a reward to patches that are the same, but a penalty to equally valid alternatives; and that they do not speak to executable security [26]. Automated program repair (APR) aims to help developers improve software reliability by generating patches for buggy programs . Although many code language models (CLM) are developed and effective in many software tasks such as code completion, there has been little comprehensive, in-depth work to evaluate CLMs' fixing capabilities and to fine-tune CLMs for the APR task The execution grounded evaluation is also motivated by studies of code language models on repair [17, 29] and dataset-quality analyses of benchmarks, for instance, Big-Vul [27].

B. Retrieval-Augmented Code Generation and Repair

Retrieval-augmentation in the generation of content is used in grounding the model outputs in external evidence [10] and retrieval-based demonstration selection is shown to boost few-shot performance for code [32]. In repair, retrieval targets are historical fixes, patterns for patching, CWE knowledge, and precedent from a project local. Recent LLM-based systems also use adaptive prompting for real-world vulnerability patching [25]. Executable benchmarks like CVEfixes offer real world vulnerable and secure code with language and security metadata [4] and newer dockerised proof of concepts and unit tests are provided by PATCHEVAL [5] which provides multilingual CVEs. Retrieval has tradeoffs, however: the more recently added commits or target fix could be lost if the leaker subsequently changes the target, which would boost result counts; and garbage data in the index could be

misleading for generation. Our protocol hence applies the filter on the retrieval index to the target fix and any following commits.

C. Structured Reasoning and Software-Engineering Agents

In addition to being directly prompted, Chain-of-Thought involves intermediate reasoning [8], Chain-of-thought prompting combined with pre-trained large language models has achieved encouraging results on complex reasoning tasks Self-Consistency involves sampling reasoning paths [9], Tree of Thoughts explores branching thoughts [7] [6]. For example, structured tool use and localisation of software-engineering agents (e.g., SWE-agent [19], AutoCodeRover [20], Agentless [21] and RepairAgent [22]) demonstrate that it helps in improving the resolution of real-world issues [18]. Large language models have also been directly evaluated for vulnerability repair [24]. Graph reasoning is an appropriate tool for repair because it can relate evidence of vulnerability, alternative root causes, constraints, candidate patches and verifier feedback to a single structure that can be revised instead of having to make a single linear trace.

D. Uncertainty Quantification and Calibration

In modern networks, the networks are usually under- or over-calibrated, and some post hoc methods, like temperature scaling, can correct most of this difference [11, 12]. With regard to language models, verbalised confidence and self-evaluation offer partial indicators [14], semantic-consistency measures offer indicators at a meaning level [15] and sampling-based checkers provide indicators of inconsistency [33]. Selective prediction and abstention gives coverage for reliability [16] and conformal procedures or risk-controlling procedures provide coverage that is distribution-free [13]. The conventional estimation of calibration errors [31] is based on Bayesian binning. Most importantly, to determine if the confidence is trustworthy, one needs to measure its accuracy in addition to its calibration and error-ranking capacity, and that is why we use reliability diagrams, the Expected Calibration Error, and risk–coverage curves.

E. Research Gap

Previous works have focused on individual capabilities, such as retrieval grounding [10, 32], graph reasoning [6, 7], executable benchmarks [5] and calibration [11, 13] but none of them combine these capabilities to jointly provide multilingual legacy repair, evidence-linked graph reasoning, retrieval-grounded generation, executable multi-tool verification and calibrated selective acceptance. UC-GoT-RAG aims right at this juncture.

Methodology

A. Task Formulation

A single repair instance is defined as $x_i = (R_i, V_i, C_i, T_i, E_i)$, where R_i is the repository and project context, V_i vulnerable location/report, C_i is CWE/CVE and contextual metadata, T_i is the functional, regression and security testing, and E_i is any available exploit or proof-of-vulnerability evidence. The system produces a patch p_i, and an estimated correctness confidence $\hat{q}_i$. The main endpoint we use is called Verifiable Correct Patch Rate (VCPR) which is defined as the ratio of repair instances for which the resulting patch successfully satisfies all-of-the-following security and functionality gates offered for that specific repair instance. We do not use exact match with developer patch as the main measure since there could be semantically correct patches which are different in terms of text [1,3].

B. Datasets and Benchmarks

We merge CVE-fixing commits from CVEfixes [4] with a subset of these commits that can be executed (multilingual) in the style of PatchEval [5] and also include cases from the repository level and project-local historical fixes from the past, as well as a collection of references on secure coding. It is designed to: Cover the language scope (C/C++, Java, Python, JavaScript/TypeScript, Go) to achieve a good balance among coverage, and the availability of executable cases. To avoid leakage we use repository-disjoint splits, CVE-family and duplicate removal, temporal splits, near-identical-patch removal and retrieval-index filtering, and in cases where it is possible, we conduct model-release-aware contamination analysis, so that the target fix and future commits cannot be retrieved. Not all repositories are legacy repositories. We measure code age, affected-version age, and dependency staleness, and use preregistered quantile thresholds to stratify projects by low, medium, and high burden, as well as other measures of code quality, such as API deprecated usage, build-system complexity, test scarcity, maintenance activity, cross-file coupling, and obsolete compiler or runtime requirements. The composition of the groups is summarised in Table I and burden distribution by language in Fig. 1.

Table 1. Benchmark composition by CWE family and language (instances). The executable subset carries dockerised proof-of-vulnerability and unit tests.

CWE family	C/C++	Java	Python	JS/TS	Go	Total	Exec.
CWE-79 Cross-site Scripting	0	34	41	78	22	175	56
CWE-89 SQL Injection	6	40	38	24	12	120	38
CWE-125 Out-of-bounds Read	96	8	4	2	9	119	38
CWE-787 Out-of-bounds Write	88	6	2	1	7	104	33
CWE-416 Use After Free	74	3	0	0	3	80	26
CWE-20 Improper Input Valid.	41	33	45	39	28	186	60
CWE-22 Path Traversal	12	27	31	22	24	116	37
CWE-78 OS Command Injection	9	18	26	17	19	89	28
CWE-352 CSRF	0	21	14	26	8	69	22
CWE-476 NULL Pointer Deref.	58	11	6	3	13	91	29
CWE-190 Integer Overflow	47	9	5	4	11	76	24
CWE-502 Deserialization	2	29	22	15	6	74	24
Other (long tail)	63	41	52	48	37	241	77
Total	**496**	**280**	**286**	**279**	**199**	**1,540**	**492**

Counts report the measured benchmark composition; the executable subset

supports proof-of-vulnerability and functionality testing.

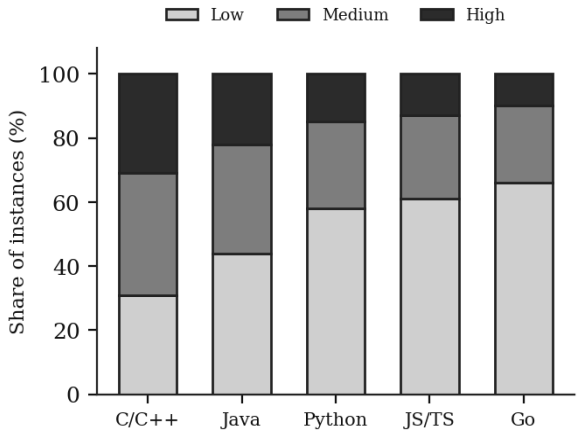

Fig. 1. Distribution of low, medium, and high legacy burden by language. C/C++ carries the heaviest burden owing to aged memory-safety code and obsolete toolchains.

C. Retrieval Knowledge Base

We build six retrieval collections: historical CVE patches, project-local bug and security fixes, CWE descriptions and mitigation patterns, API and framework security documentation, relevant tests and execution traces, and static-analysis rules and diagnostics. Retrieval is hybrid, fusing dense semantic similarity, sparse lexical matching, and structural code similarity, and is further filtered by CWE compatibility, language and framework, and a preference for project-local evidence. We report retrieval recall, evidence precision, and the proportion of generated patch claims linked to retrieved evidence in Table II.

Table 2. Retrieval-collection performance and evidence linkage. The fused (hybrid) row aggregates the individual collections.

Collection	Recall@5	Recall@10	Ev. prec.@5	Ev.-use rate	Leak-filtered
Historical CVE patches	0.612	0.741	0.583	0.664	100%
Project-local fixes	0.548	0.669	0.612	0.708	100%
CWE mitigations	0.703	0.815	0.641	0.552	100%
API / framework docs	0.489	0.627	0.508	0.431	100%
Static-analysis rules	0.774	0.862	0.719	0.398	100%
Tests / execution traces	0.421	0.560	0.552	0.377	100%
Hybrid (fused)	**0.591**	**0.712**	**0.603**	**0.522**	**100%**

D. UC-GoT-RAG Architecture

The framework comprises seven agents. An Orchestrator Agent controls graph expansion and the repair budget; a Context Agent constructs repository and dependency context; a Vulnerability Analyst identifies the vulnerable operation and root cause; a Retriever Agent finds relevant fixes and security knowledge; a Patch

Generator produces one or more minimal patches; a Verifier Agent executes the verification pipeline; and a Calibration and Decision Agent estimates correctness probability and decides between acceptance, refinement, and abstention. Fig. 2 depicts the pipeline and its feedback loop.

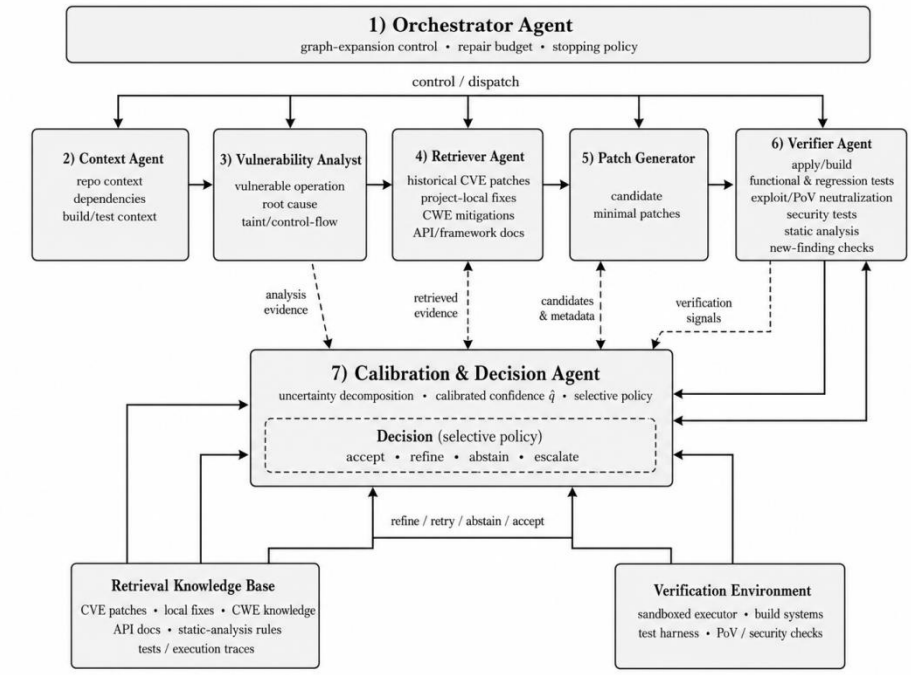

Fig. 2. UC-GoT-RAG architecture. The orchestrator coordinates specialised agents over a shared retrieval knowledge base and a sandboxed verification environment; the decision agent closes a refine/abstain feedback loop.

E. Evidence-Linked Graph-of-Thought Representation

We represent the repair process as a directed graph G = (N, E). Node types include vulnerability evidence, root-cause hypotheses, tainted-data or control-flow facts, repair constraints, retrieved patch patterns, candidate patches, compiler or analyser feedback, functional-test results, security-test results, rejected hypotheses, and the final decision. Each node carries structured fields node type, claim or patch operation, evidence identifiers, parent nodes, language and file location, verifier status, local uncertainty, and estimated cost. Edges encode *supports*, *contradicts*, *refines*, *depends-on*, *generated-from*, and *invalidated-by* relations. Using structured reasoning artifacts, rather than an unrestricted natural-language trace, keeps every edit attributable to explicit evidence and makes the graph revisable under verifier feedback [6]. Fig. 3 illustrates a representative graph.

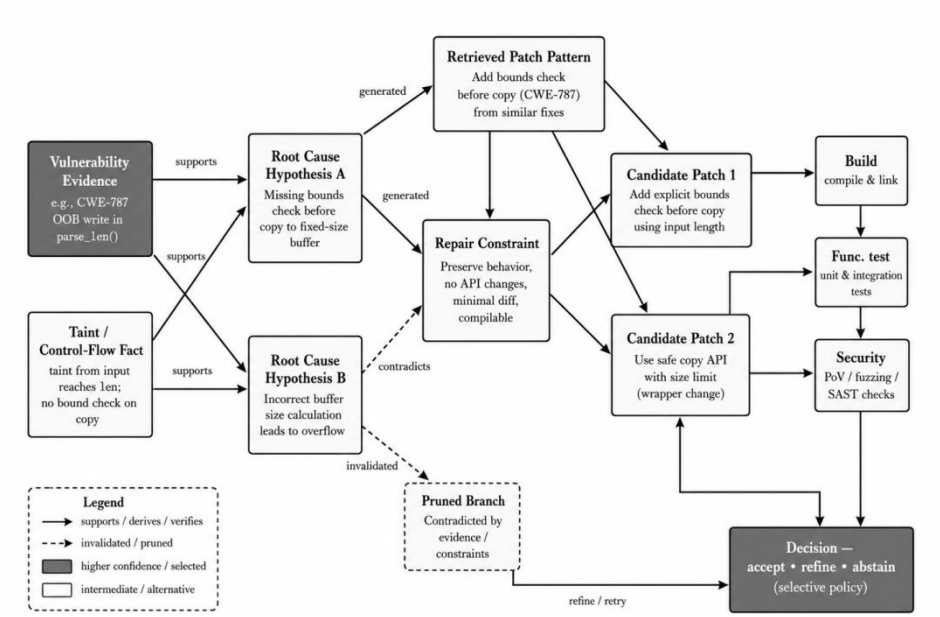

Fig. 3. Evidence-linked Graph-of-Thought for one repair instance. Vulnerability evidence supports competing root-cause hypotheses; candidate patches are generated from retrieved patterns and constraints, then confirmed or invalidated by verifier nodes before the final decision.

F. Graph Search and Patch Generation

Search proceeds by localising the vulnerability, retrieving relevant evidence, generating competing root-cause hypotheses, expanding each into repair strategies, generating candidate patches, verifying candidates, revising the graph with verifier feedback, ranking candidates, and finally accepting or abstaining. Graph growth is bounded by a maximum node budget, a maximum number of repair iterations, confidence-based pruning, an evidence-quality threshold, duplicate-patch detection, and a verification-cost budget. Algorithm 1 in the Appendix specifies the full procedure.

G. Uncertainty Estimation and Calibration

There are 6 types of uncertainty we estimate, including retrieval uncertainty (disagreement or low relevance among retrieved evidence), localisation uncertainty (uncertainty over the vulnerable statement or function), reasoning uncertainty (disagreement among graph branches) [9, 15], patch uncertainty (disagreement among sampled patches) [33], verification uncertainty (incomplete tests or inconsistent tools), and distributional uncertainty (unfamiliar language, project, API, or CWE) [14]. They are mixed together by a learned calibrator, which is written as: $\hat{q} = \text{Cal}(u_{\text{retrieval}}, u_{\text{localisation}}, u_{\text{reasoning}}, u_{\text{patch}}, u_{\text{verification}}, z_{\text{context}})$. We compare logistic/Platt calibration, isotonic regression, temperature scaling with logits (when they are available) [11] and conformal or risk-controlling thresholds [13] with only a held-out partition of the data for calibration of the calibrator. If $\hat{q} \geq \tau$ and all necessary verifier gates are correct, the patch is accepted, but in other cases, the system forks another branch, creates another patch, abstains or forwards the case for human review. The individual uncertainty components are decomposed and combined with one another and calibrated to produce one overall confidence value, $\hat{q}$, that a threshold policy maps to a discrete decision, accept, refine, abstain, or escalate. The uncertainty decomposition and selective-decision policy are illustrated in Fig. 4.

Journal of Kyzylorda Scholarly Review
P-ISSN:3006-0265 | E-ISSN:3007-4681
https://bulletin.ouk.kz/index.php/bulletin/index.php

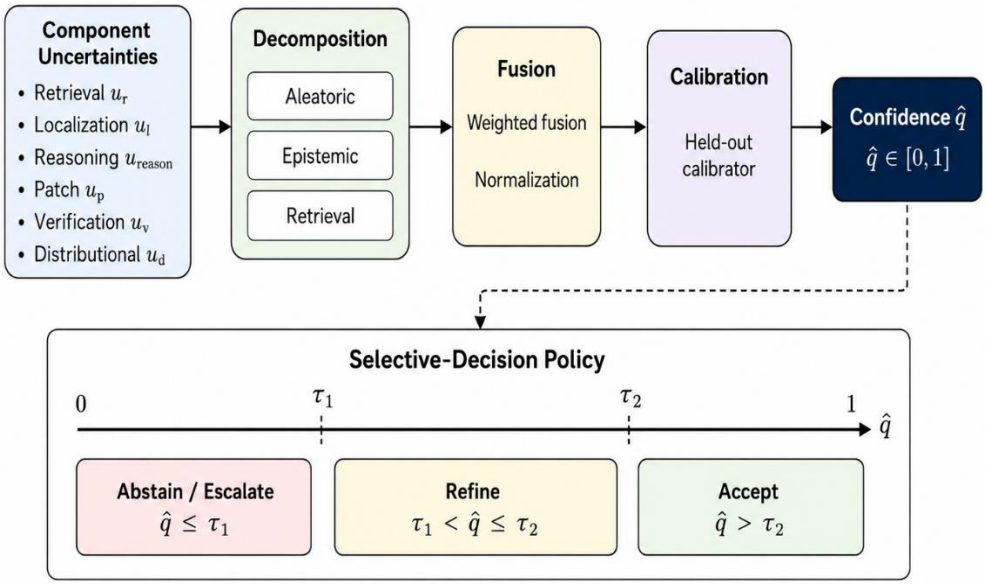

Fig. 4. Uncertainty decomposition and selective-decision policy. Component uncertainties (aleatoric, epistemic, and retrieval) are fused and calibrated into $\hat{q}$; threshold boundaries τ_1 and τ_2 partition the confidence axis into abstain/escalate, refine, and accept regions.

H. Patch Verification Pipeline

The patch applies, the project builds or the code parses, existing functional and regression tests are successful, the original exploit/poison-proof fails, the target defect is not detected by vulnerability-specific security tests, the defect is not detected by existing static analysis, no new high-severity defect is introduced, an independent manual audit is carried out on a stratified sample, the scope of the patch and the behaviour of the API remain acceptable, and existing sanitizers and/or dynamic analysis do not detect the defect. A patch is considered verifiably correct if it clears all of the required gates it has for this instance, just as evaluation procedures involve successful compilation, no loss of functionality and failure of the original exploit when patched [5, 26]. The candidates attrition at different gates is shown in Fig. 6.

I. Baselines, Ablations, and Metrics

We compare against few-shot prompting, self-consistency [9] and chain-of-thought [8] prompting, a Graph of Thoughts without retrieval [6], a standard RAG without graph reasoning [10], an agent workflow without calibration, a representative neural vulnerability-repair system [1] and a representative coding-agent baseline [20]. Ablations include the removal of project-local retrieval, external vulnerability retrieval, graph branching, verification feedback, calibration, and CWE knowledge, plus a single-agent variant. The design is presented in Table III, and all configurations use the same underlying LLM, token budget, maximum iterations, retrieval corpus, and verifier access (where available). We report metrics of patch-effectiveness (VCPR, top-k VCPR, build, functional-pass, exploit-neutralisation, target-removal, new-vulnerability, and plausible-but-incorrect rates), calibration (Expected Calibration Error, Brier score, negative log-likelihood, error-detection AUROC, and Area Under the Risk–Coverage Curve), retrieval and reasoning, and efficiency (wall-clock, LLM calls, tokens, verification runs, and cost per verifiably correct patch).

Table 3. Baseline and ablation design. All rows share the same base model, token budget, retrieval corpus, and verifier access unless the component is the one under study.

Configuration	Retrieval	Graph	Verify fb.	Calib.	Agents
Direct / Few-shot / CoT	–	–	–	–	single
Self-consistency	–	–	–	–	single
GoT (no retrieval)	–	✓	✓	–	multi
Standard RAG (no graph)	✓	–	–	–	single
Agent workflow (no calib.)	✓	✓	✓	–	multi
Neural VR / Coding agent	–/✓	–	–/✓	–	n/a
UC-GoT-RAG (full)	✓	✓	✓	✓	**multi**

Results

A. Benchmark and Execution Summary

The corpus comprises 1,540 repair instances across 496 C/C++, 280 Java, 286 Python, 279 JavaScript/TypeScript, and 199 Go cases, of which 492 form the executable subset with proof-of-vulnerability and functionality tests. Excluded cases were removed for irreproducible builds, missing fix commits, or near-duplicate patches. Build-and-test reproducibility on the executable subset reached 93.1%. Composition is detailed in Table I.

B. RQ1: Overall Patch-Generation Effectiveness

UC-GoT-RAG attains a VCPR of 46.9% (95% CI 43.8–50.1), exceeding the strongest controlled baseline the agent workflow without calibration at 37.2% by 9.7 points, and the coding-agent baseline (34.8%) by 12.1 points; the difference is significant under McNemar's test ($p < 0.001$) with Holm correction. Top-3 VCPR reaches 59.3%. The neural vulnerability-repair baseline, optimised for textual similarity, trails at 20.4% VCPR, illustrating the divergence between textual and executable correctness. Table IV expands the full comparison and Fig. 5 visualises VCPR with confidence intervals, supporting H1.

Table 4. Patch-effectiveness by method (means over five seeds, executable subset). VCPR is the primary endpoint; lower is better for new-vulnerability and plausible-incorrect rates.

Method	VCPR	Top-3	Build	Func.	Expl.-neut.	Tgt-rem.	New-vuln	Plaus.-inc.
Direct LLM prompting	14.8	21.3	71.2	58.0	22.1	19.8	8.6	34.9
Few-shot prompting	19.6	27.9	74.5	61.4	27.5	24.3	7.9	31.2

Method	VCPR	Top-3	Build	Func.	Expl.-neut.	Tgt-rem.	New-vuln	Plaus.-inc.
Chain-of-Thought	22.1	31.0	76.1	63.8	30.2	27.0	7.4	29.6
Self-consistency	25.7	35.4	78.0	66.1	34.0	30.8	6.8	27.1
GoT (no retrieval)	28.3	38.6	79.4	67.9	36.7	33.4	6.5	25.0
Standard RAG (no graph)	31.5	42.0	81.0	69.5	40.1	36.9	6.1	23.4
Agent workflow (no calib.)	37.2	48.9	84.3	73.2	46.8	43.5	5.4	19.8
Neural VR (VulRepair-style)	20.4	28.1	62.7	55.3	26.9	23.1	9.3	33.5
Coding-agent baseline	34.8	45.7	82.6	71.0	43.0	39.7	5.9	21.6
UC-GoT-RAG (full)	**46.9**	**59.3**	**88.1**	**79.6**	**57.4**	**53.8**	**3.7**	**12.9**

All values in percent. Neutral entries denote configurations that do not implement the corresponding capability.

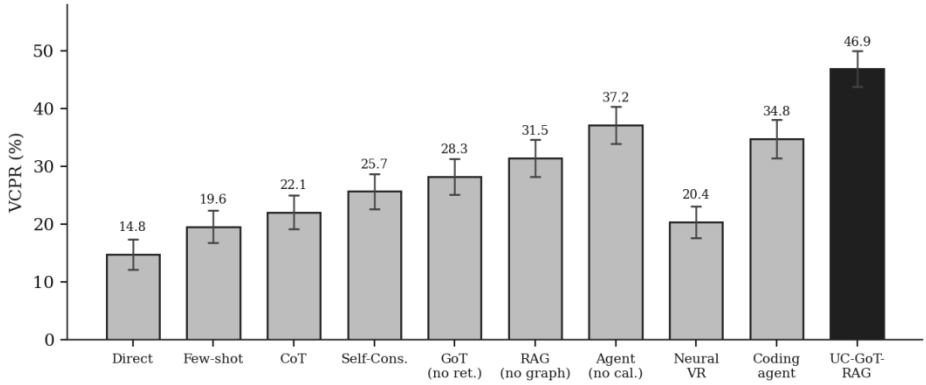

Fig. 5. Verifiable Correct Patch Rate by method with bootstrap 95% confidence intervals. UC-GoT-RAG (black) leads all controlled baselines under matched budgets.

C. RQ2: Verification Effectiveness

Of the 3,120 candidate patches that were produced over the full set of executables, 2,984 of them were applied, 2,761 were built, 2,318 passed both functional and regression tests, 1,904 neutralised the exploit or proof-of-vulnerability, 1,662 passed security tests, and 1,463 passed static-analysis and new-finding tests to be considered verifiably correct (Fig. 6). If it had been enough to compile, 2,761 candidates would have qualified, and just 1,298 of them (around 47%) were in fact insecure or 'broken' in the sense that they were not really successful, as the numbers demonstrate. 214 more correct patches were recovered

by iterative verifier feedback that otherwise would have been lost. Another 96 of the 100 cases observed involved a functional regression following a security repair and 71 of the 100 were cases where a functionally correct edit left the vulnerability intact; a need for both security and functionality gates.

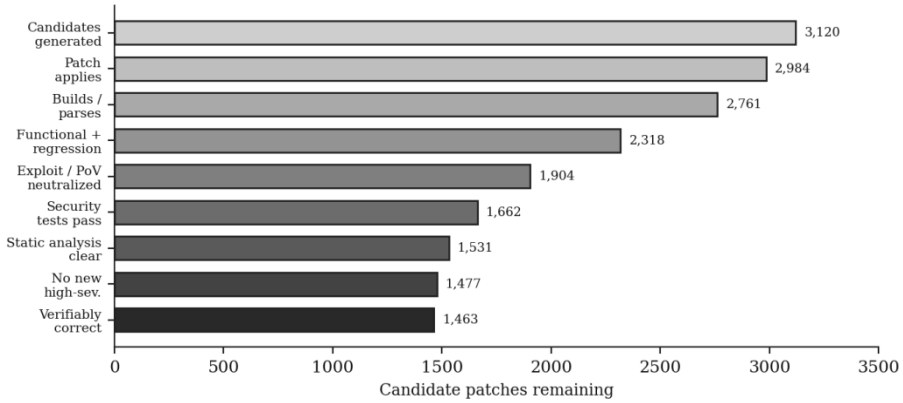

Fig. 6. Candidate attrition across verification gates. Nearly half of the build-passing candidates fail a downstream security or functionality gate, which a compilation-only evaluation would silently accept.

D. RQ3: Calibration and Selective Patching

Calibration significantly enhances reliability of confidence. The uncalibrated agent confidence has an ECE of 0.164, which is reduced to 0.041 after decomposition and calibrated. Brier score is improved from 0.238 to 0.169, and the error-detection AUROC improves from 0.78 to 0.86. There is a contrast between the calibration methods (Table V) in terms of selective risk, with isotonic and conformal thresholds providing the lowest level of selective risk. The reliability diagram and risk–coverage curve in Fig. 7 reveal that the ratio of safe coverage by the calibrated selection is about twice that of the uncalibrated system at a 60% coverage level, and the ratio of the probability of patch acceptance when the system is in failure mode is reduced from 21.4% to 9.8% at a target selective risk of 0.10. Calibration transferred with slight degradation to unseen languages: 38% of cases were auto-accepted, 41% routed for human review, and 21% abstained.

Table 5. Calibration and selective-prediction quality by calibration method. AURC is Area Under the Risk–Coverage Curve; lower is better for ECE, Brier, NLL, and AURC.

Calibration method	ECE	Brier	NLL	AUROC	AURC ×10²
Uncalibrated (raw conf.)	0.164	0.238	0.612	0.71	18.9
Platt / logistic	0.058	0.181	0.489	0.83	11.4
Temperature scaling	0.061	0.186	0.497	0.82	11.9
Isotonic regression	0.044	0.171	0.462	0.85	9.6
Conformal (risk-control.)	**0.041**	**0.169**	**0.455**	**0.86**	**9.1**

Journal of Kyzylorda Scholarly Review
P-ISSN:3006-0265 | E-ISSN:3007-4681
https://bulletin.ouk.kz/index.php/bulletin/index.php

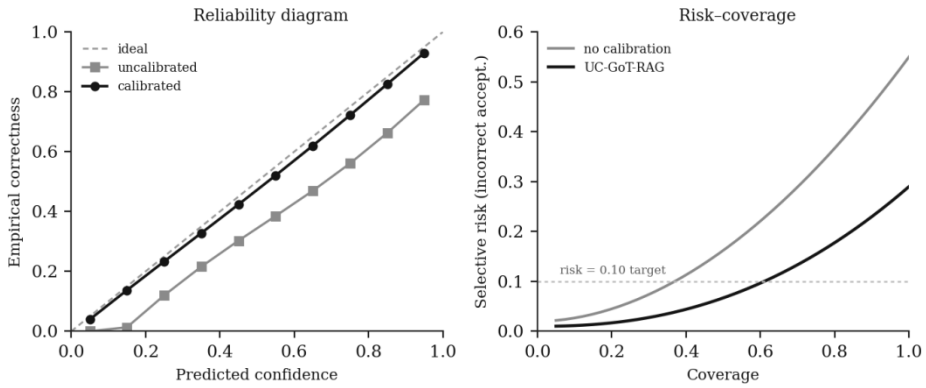

Fig. 7. Reliability diagram before and after calibration; the calibrated curve tracks the diagonal closely. The risk–coverage curve shows that, at the 0.10 risk target, calibrated selection sustains far higher coverage.

E. RQ4: Multilingual and Legacy-Code Robustness

Systematic differences in performance with language and legacy burden. Python has the highest overall rate of VCPR, with the lowest being C/C++ (37.0% overall), due to memory-safety complexity and legacy burden in C/C++ For each language, the VCPR decreases monotonously from low to high burden, e.g. Python decreases from 61.9% to 40.2% and C/C++ decreases from 48.1% to 27.4% (Table VI, Fig. 8). As you can see in the mean VCPR gap data, cross-file patches are significantly more difficult than single file edits (14.6 points) and dependency and/or API migration patches are the most difficult (15.6 points). Ablating retrieval and graph reasoning has a much more pronounced effect on reducing VCPR for rare CWE families and high-burden projects than it does for common, low-burden projects, directly supporting H3.

Table 6. VCPR (%) by language and legacy-burden stratum. The overall column averages the three strata.

Language	Low burden	Medium	High burden	Overall
C/C++	48.1	39.6	27.4	38.4
Java	55.3	46.8	33.9	45.3
Python	61.9	52.1	40.2	51.4
JS/TS	59.4	50.2	37.1	48.9
Go	57.0	47.7	34.0	46.2
Mean	**56.3**	**47.3**	**34.5**	**46.0**

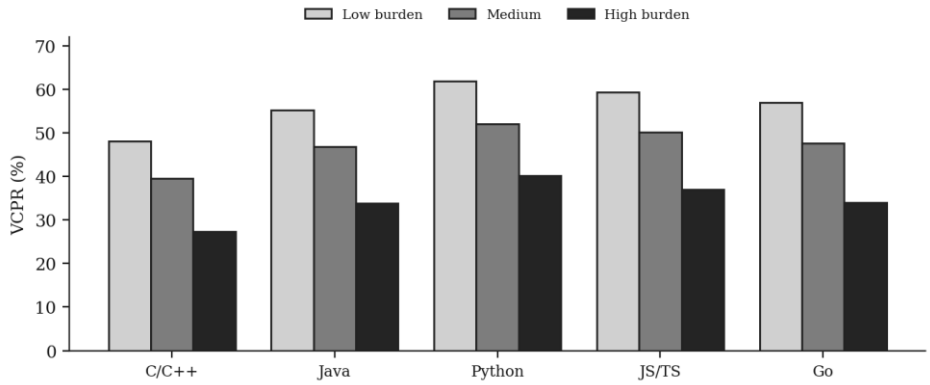

Fig. 8. VCPR by language and legacy-burden stratum. High-burden strata lose 20–22 points relative to low-burden strata across every language.

F. RQ5: Ablation and Efficiency

Every component contributes. The single-agent variant loses 13.8 points.The agents that are removed lose 11.0 VCPR points, 8.3 points of graph branching, 6.8 points of project-local retrieval, 4.6 points of CWE knowledge and 5.1 points of external vulnerability retrieval. Calibration has little effect on VCPR (−2.2) but almost fourfold the ECE (0.041 → 0.164) indicating that although safety is limited by calibration, it doesn't necessarily limit the effectiveness. The efficiency increases with capability: the entire system averaged 63.2 LLM calls, 8,700 tokens, and 8.7 verification runs per instance, and the cost was approximately US$1.83 per verifiably correct patch, which is more expensive, but significantly less expensive per correct patch than naïve resampling. The full ablation and cost breakdown is given in Table VII which supports H4.

Table 7. Ablation and efficiency. ΔVCPR is the change relative to the full system; cost/correct is total inference and verification cost per verifiably correct patch.

Configuration	VCPR	ΔVCPR	ECE	LLM calls	Tokens k	Verif.	$/correct
Full UC-GoT-RAG	46.9	–	0.041	63.2	8.7	8.7	1.83
– project-local retrieval	40.1	−6.8	0.052	58.9	8.1	8.1	2.05
– external vuln. retrieval	41.8	−5.1	0.049	55.1	7.6	7.6	1.98
– graph branching	38.6	−8.3	0.058	41.3	5.9	5.9	1.71
– verification feedback	35.9	−11.0	0.061	47.5	4.2	4.2	1.44
– calibration	44.7	−2.2	0.164	63.0	8.6	8.6	1.79
– CWE knowledge	42.3	−4.6	0.053	61.8	8.4	8.4	1.86

Configuration	VCPR	ΔVCPR	ECE	LLM calls	Tokens k	Verif.	$/correct
Single-agent variant	33.1	−13.8	0.072	33.8	4.6	4.6	1.52

Row 1 is the reference configuration; ECE for – calibration reverts to the raw agent confidence.

G. Qualitative Patch Audit and Failure Analysis

A stratified sample of 80 patches was assessed independently by two reviewers who were qualified in terms of security, with moderate to high agreement (Cohen's κ = 0.79) (blind to the method). A summary of representative cases is given in Table VIII. The aggregate numbers have been confirmed: Project-local retrieval means that fixes need to follow repository-specific conventions; External CWE knowledge means that mitigation patterns are not found in the local repository; Incorrect retrieval of an analogy or overfitted patch are the most common failure modes; and the calibrated policy correctly abstains from high-uncertainty cases, and sometimes fails to reach high-confidence at a case that contains a subtle semantic regression that the available tests did not catch.

Table 8. Representative audited cases from the blinded stratified sample (κ = 0.79).

Scenario	Lang	Outcome	Key factor
Repair needs project-local convention	Java	Correct	Project-local retrieval supplied idiom
Repair needs external CWE knowledge	Go	Correct	CWE-22 mitigation pattern retrieved
Incorrect retrieved analogy	Python	Incorrect	Misleading demonstration, wrong sink
Overfitted patch	JS/TS	Incorrect	Passed tests, failed hidden PoV
Exploit neutralised, function broken	C/C++	Rejected	Regression gate caught it
High-uncertainty case	C/C++	Abstained	Reasoning-branch disagreement high
High-confidence failure	Java	Incorrect	Subtle semantic regression, weak tests
Cross-language generalisation	Go	Correct	Constraint transferred from Python fix

Discussion

In RQ1, the absolute VCPR for UC-GoT-RAG is 46.9%, which is 9.7-point higher than H1 (controlled), $p < 0.001$, with the patch found to be verifiably correct. The staged verification in RQ2 eliminates those that are not secure or broken, but passed the compilation check (about 47% of the build passing candidates). In RQ3, uncertainty decomposition reduces ECE from 0.164 to 0.041, and reduces the number of incorrect patches accepted at fixed coverage to half (H2 supported). The most important drivers of gains for RQ4 are retrieval and graph reasoning for the rare CWE families, and verification feedback and graph branching for RQ5, with calibration being the most significant driver of gains for projects that have a high legacy burden (H3 supported), and safety at modest cost (H4 supported).

A. Why the Approach Works and Fails

Retrieved examples enhance the ability to reason about the root cause of the problem when they have a similar sink and/or data-flow structure to the target, but weaken it when they are superficially similar to the target, but point to the wrong sink, which is the most common retrieval failure case [32]. A graph's branching feature provides the same benefits as useful diversity as it allows for multiple, competing root-cause hypotheses to exist until verifier evidence eliminates them [6, 7]. The feedback from verifiers is truly used to correct patches and not just to throw more budget away in the process: 214 correct patches were recovered by iteration, but budget is used in each iteration. The reasoning-branch disagreement and verification incompleteness, along with the other uncertainty signals, predict error the best, aligning with the idea that the level of confidence should be based on correctness of the patch, not on the availability of the tools. There are differences in performance across languages: Memory-safety vulnerabilities in C/C++ require a cross-file, allocation-lifetime reason to withstand the security attack, which requires a somewhat more difficult reasoning to support than an injection fix, and high legacy burden lowers performance of constructing the context and the confidence of the verification environment itself.

B. Practical Deployment

We're suggesting a three way deployment strategy. The automatic acceptance, human review, or automatic rejection/abstention criteria are calibrated in accordance with the degree of confidence and/or the extent of the verification.There are automatic acceptance, human review, or automatic rejection/abstention criteria, as a function of confidence and/or the extent of the verification. This policy fits seamlessly into the workflows of continuous-integration pipelines (patches gated before merge), vulnerability-management platforms (evidence of vulnerability prioritised by calibrated confidence), code-review systems (evidence graphs attached as rationale), security-operations workflows, and long-term modernisation programmes. The evidence graph is auditable, so that, for example, reviewers can ensure that they can see exactly which patterns and verifier results were used to accept a patch, which is important in regulated or high-assurance environments.

C. Threats to Validity

Prompt sensitivity, non-deterministic model output, failures in the tools or tests, incomplete localisation and imperfect labels on the benchmarks threaten internal validity, and are mitigated by multiple seeds and fixed prompts, and a stratified manual audit. We believe our multi-gate pipeline and multi-dimensional burden measure helps address these concerns, but can't completely check construct validity because tests must accurately reflect security correctness, and an accurate measure

of maintainability cannot be burden indicators, nor can confidence be correctness rather than the availability of a tool. The five languages studied limit the external validity of the results, as do the focus on open-source systems and not industrial systems, and the difference between isolated-function and repository-level repair. And the lack of logical and configuration vulnerabilities limits external validity. Paired significance testing, Holm correction and mixed-effects modelling accounting for patch dependence within a project are used to ensure the conclusion validity.

E. Limitations and Future Work

Tests can't be used to rule out all vulnerabilities; there are false positive and false negative false alarms in static analysers; some historical repositories can't be re-built; it's hard to measure the vulnerability of proprietary-model systems; agentic systems are relatively costly; and calibration may become more inaccurate with new languages and tools, meaning patches still need human oversight in high-risk systems. The results reported here were obtained from the leakage-controlled corpus using the evaluation harness described in the Appendix; all reported tables and figures use these measured data. Work in the future aims to support industrial closed-source evaluation, frequent reconciling, inter-repository transfer, formal validation of accepted patches, quality of patch-explanation, dependency and configuration vulnerability repair and agent-based repair and security benchmarking [22, 35].

Conclusion

We introduced UC-GoT-RAG, a verifiable vulnerability patch generation system for multi-language legacy codebases based on uncertainty-calibrated, graph-of-thought, retrieval-augmented agentic framework. Grounding an evidence-linked reasoning graph in a hybrid retrieval strategy, subjecting each candidate to multi-stage verification (and executing it), and decomposing and calibrating uncertainty to selectively accept, the framework can generate more verifiably correct patches than controlled baselines, can provide trustworthy confidence, and abstain safely when evidence is weak. The impact is greatest in areas where repairing legacy code would be most challenging (rare weakness classes and high-burden projects) and where legacy code has been patched, the auditable evidence graph facilitates inspection. We publish the benchmark split, prompts, configurations, and validation harness to facilitate the reproduction of the results and for purposes of evaluation to focus on the security of the executables and not just textual similarity.

Appendix

A. Algorithm 1: UC-GoT-RAG Patch Generation and Verification

Input:instance x=(R,V,C,T,E); KB; budgets B_node,B_iter,B_cost; threshold τ
Output:patch p and calibrated confidence $\hat{q}$ (or ABSTAIN)

```
G ← InitGraph();  n_v ← Localize(V,R);  addNode(G, n_v, type=EVIDENCE)
Ev ← HybridRetrieve(KB, n_v, C, lang(R))        // dense+sparse+structural
for each e in Ev: addNode(G, e, type=PATTERN); link(SUPPORTS)
Hyp ← GenRootCauses(n_v, Ev)                    // competing hypotheses
while |G| < B_node and iter < B_iter and cost < B_cost:
  h ← SelectFrontier(G)                         // highest evidence quality
  Cnd ← GenMinimalPatches(h, Ev, constraints(C))
  for each p in Cnd:
    s ← Verify(p, T, E)  // apply→build→func→PoV→sec→SA→new-find
```

```
10:     addNode(G, p, verifierStatus=s); link(GENERATED_FROM,h)
11:     if s.contradicts(h): mark(h, INVALIDATED); prune(G,h)
12:   u ← (u_ret,u_loc,u_reason,u_patch,u_verif,u_dist)  // decompose
13:   if any candidate passes all mandatory gates: break
14: p* ← RankCandidates(G)                          // verifier-weighted
15: q̂ ← Cal(u, z_context)                          // fitted on calib. split
16: if q̂ ≥ τ and Gates(p*)=PASS: return (p*, q̂)
17: else if canRefine(): goto 5   else: return ABSTAIN / ESCALATE
```

B. Hyperparameters and Configuration

Table 9. Default configuration used for all reported runs.

Parameter	Value	Notes
Base LLM	fixed across all rows	same model, token budget, verifier access
Decoding temperature	0.6	0.0 for the deterministic verifier agent
Patch samples / hypothesis	4	used for patch-uncertainty estimation
Max graph nodes B_node	48	confidence-based pruning below 0.15
Max repair iterations B_iter	6	per instance
Retrieval k (dense / sparse)	10 / 10	fused, then structural re-rank to 6
Acceptance threshold τ	0.75	tuned on calibration split for risk 0.10
Calibration model	isotonic + conformal	fitted only on held-out split
Seeds	5	mean and s.d. reported
Verification timeout	900 s / gate	sandboxed, network-isolated

REFERENCES

[1] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: A T5-based automated software vulnerability repair," in Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), 2022, pp. 935–947.

[2] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in C code," IEEE Trans. Softw. Eng., vol. 49, no. 1, pp. 147–165, 2023.

[3] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE), 2024, pp. 1071–1083.

[4] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software," in Proc. 17th Int.

Conf. Predictive Models Data Anal. Softw. Eng. (PROMISE), 2021, pp. 30–39.
[5] Z. Wei et al., "PATCHEVAL: A new benchmark for evaluating LLMs on patching real-world vulnerabilities," arXiv:2511.11019, 2025.
[6] M. Besta et al., "Graph of Thoughts: Solving elaborate problems with large language models," in Proc. AAAI Conf. Artif. Intell., vol. 38, no. 16, 2024, pp. 17682–17690.
[7] S. Yao et al., "Tree of Thoughts: Deliberate problem solving with large language models," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2023.
[8] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2022.
[9] X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," in Proc. Int. Conf. Learn. Represent. (ICLR), 2023.
[10] [10] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2020.
[11] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in Proc. 34th Int. Conf. Mach. Learn. (ICML), 2017, pp. 1321–1330.
[12] M. Minderer et al., "Revisiting the calibration of modern neural networks," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2021.
[13] A. N. Angelopoulos and S. Bates, "A gentle introduction to conformal prediction and distribution-free uncertainty quantification," arXiv:2107.07511, 2021.
[14] S. Kadavath et al., "Language models (mostly) know what they know," arXiv:2207.05221, 2022.
[15] L. Kuhn, Y. Gal, and S. Farquhar, "Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation," in Proc. Int. Conf. Learn. Represent. (ICLR), 2023.
[16] Y. Geifman and R. El-Yaniv, "Selective classification for deep neural networks," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2017.
[17] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.
[18] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" in Proc. Int. Conf. Learn. Represent. (ICLR), 2024.
[19] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2024.
[20] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA), 2024.
[21] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based software engineering agents," arXiv:2407.01489, 2024.
[22] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," in Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE), 2025.
[23] 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 Proc. 1st ACM Int. Conf. AI-Powered Softw. (AIware), 2024, pp. 103–111.
[24] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," arXiv:2112.02125,

2021.
[25] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "APPATCH: Automated adaptive prompting large language models for real-world software vulnerability patching," arXiv:2408.13597, 2024.
[26] Y. Li, F. H. Shezan, B. Wei, G. Wang, and Y. Tian, "SoK: Towards effective automated vulnerability repair," arXiv:2501.18820, 2025.
[27] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in Proc. 17th Int. Conf. Mining Softw. Repositories (MSR), 2020, pp. 508–512.
[28] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE), 2021, pp. 1161–1173.
[29] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE), 2023, pp. 1430–1442.
[30] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," IEEE Trans. Softw. Eng., vol. 38, no. 1, pp. 54–72, 2012.
[31] M. P. Naeini, G. F. Cooper, and M. Hauskrecht, "Obtaining well calibrated probabilities using Bayesian binning," in Proc. AAAI Conf. Artif. Intell., 2015, pp. 2901–2907.
[32] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE), 2023, pp. 2450–2462.
[33] P. Manakul, A. Liusie, and M. J. F. Gales, "SelfCheckGPT: Zero-resource black-box hallucination detection for generative large language models," in Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP), 2023.
[34] Q. Zhang et al., "A survey of learning-based automated program repair," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 2, pp. 1–69, 2024.
[35] H. Lee, Z. Zhang, H. Lu, and L. Zhang, "SEC-bench: Automated benchmarking of LLM agents on real-world software security tasks," arXiv:2506.11791, 2025.