CodeGuardian: A Survey and Retrieval-Augmented System Design for Automated Software Vulnerability Detection and Patch Generation Using Large Language Models

Mr. Aniruddha Kolpyakwar[1], Dhanashree Goverdhane[2], Sagar patil[3], Vedant Raut[4], Vandana Nathe[5]
[1]*Assistant Professor at School of Computer Science and Engineering, Department of Computer Science and Engineering Sandip University, Nashik, Maharashtra, India 422213*
[2,3,4,5]*Student at School of Computer Science and Engineering, Department of Computer Science and Engineering Sandip University, Nashik, Maharashtra, India 422213*

***Abstract*—There are more potential software vulnerabilities than can be discovered by hand or by traditional, static analysis techniques, and the amount of software code being written is greater than ever before. Recently, a new category of machine learning model the Large Language Models (LLMs) has been developed that shows great potential as a basis for automating discovery of security vulnerabilities and patch creation for correcting them. This paper summarizes a systematic survey of software vulnerability detection and generation techniques based on the foundational models of "large language model" (LLM) and introduces a novel design that combines code retrieval with vulnerability detection and attaches an iterative validation-based patch-production pipeline to this tool: CodeGuardian. It summarizes a systematic survey of software vulnerability detection and generation techniques based on the foundational models of "large language model" (LLM), and also introduces a novel design: pairing retrieval-augmented vulnerability detection with a validation-driven pipeline for patch production, which is synthesized as an iterative process: CodeGuardian. We have followed the protocol of PRISMA to include in our database all the studies found in the six digital libraries between 2019 and the beginning of 2026. We choose 55 studies with respect to graph-based detectors, fine-tuned and prompt-based models, retrieval-augmented and multi-agent systems, severity assessment, line-level localization, and automated program repair and organize them in a coherent taxonomy. We focus on four research questions and bring together their results from various angles and perspectives: What are the differences between the accuracy and reasoning in real use of the benchmarks, what are the limitations of the training data, and what is the importance of grounding knowledge. The results are interpreted and translated into design decisions for CodeGuardian and standards, datasets, and a long-term evaluation protocol to ensure the system is reliable. The survey raises key open issues of reliability, explainability, and validation, and specifies a research program for adapting security to the developer.**

***Index Terms*—Large language models, software vulnerability detection, automated program repair, patch generation, retrieval-augmented generation, secure software development, static application security testing, vulnerability assessment, secure code, software security.**

I. INTRODUCTION

In today's world, software has emerged as the nerve tissue that connects people, finance, healthcare, transportation, communications and even key national infrastructure. The areas of vulnerability on systems have expanded as they have become bigger and more interconnected, as have the surface of attack found in defects within their source code. Memory-safety vulnerabilities, injection flaws, insecure cryptographic uses, broken authentication, and other vulnerabilities continue to be introduced more quickly than they are discovered and patched, and just one previously unpatched weakness can lead to massive data breach or service outages. Even manual code review, rule-based SAST and dynamic fuzzing are useful, but are expensive to apply at scale to millions of lines of

quickly changing code. They radically over fetch developers too, and they miss on semantically subtle bugs. The crux of this study is whether more of this burden can be shifted to machine intelligence, doing so without compromising trustworthiness. What large language models (LLMs) fed on a massive amount of natural language and code have transformed this question. Unlike other detectors such as token- or pattern-based approaches, LLMs can learn about the semantic and contextual meaning of code to give them a sense of "what will happen" within the program, "what is moving" by way of data flow, and the "why" of the program development. In recent multi-task assessments of Open-Source models it has been found that one can efficiently use a single model to perform detection, classification and explanation together but performance is quite sensitive depending on the task and language [1].

The merits of compact, security-specific classifiers, like SecureFalcon, that can achieve competitive accuracy on carefully selected corpora have been shown [2]; a parallel train of thought holds that basic prediction is not enough the model's predictions must be explainable, which can lead to explainable detection architectures [3]. Importantly, models found to be defective can be directed to fix them: The models to detect defects have been extensively fine-tuned for automated program repair, thereby closing this loop between revealing defects and fixing them [4] This is tempered by very sobering facts. Reporting of effectiveness by deep-learning detectors is problematic, however, when they are re-evaluated on realistic, deduplicated data sets that are typical of production code, a large disconnect is found between laboratory and field performance [5]. Beyond the paradigm, there is growing interest in task-specific specializations, e.g., smart-contract auditing using LLM-powered multi-agent systems that discuss and compare results [6]; and structural representations of code continue to be promising and effective with sequence models [7]. These links can be clarified by a recent literature review that provides a map for future research directions for LLM-based detection and repair, where there still is a lack of consensus about evaluation, reliability and deployment [8].

Fragmentation is another factor that is a problem. Realistically, detection, severity assessment, locating, and fixing are managed by separate tools with different interfaces that require developers to do these overlap operations: setting up the scanner, the triage spreadsheet and the manual fix. Every hand-off becomes less contextual, knowledge retrieved for detection is infrequently passed on to the engineer who writes the patch, and the patch seldom gets verified with the original detector. Recent systems that are the most effective in the survey do just that: They ground detection in external knowledge, reason about program structure instead of flat text [7] and validate the program output and the suggested fixes before they are trusted [8]. This convergence encourages a seamless design and puts evidence into a single data stream that reaches inbound to validated remediation, the perfect shoes for which CodeGuardian strives.

This paper makes four contributions. First, it adopts a systematic methodology and uses the PRISMA approach for the selection and analysis of 55 representative studies and divides them into a taxonomy of five main groups: detection, assessment and localization, patch generation, secure code generation and evaluation. Second, it situates the survey in relation to the previous reviews to make the gap in the research clear and then clarifies four research questions based on that gap. Third, it summarizes cross-cutting lessons how the reasoning gap, fragility with data quality and grounding in knowledge all contribute to an actionable set of design principles. Fourth, it proposes a system design CodeGuardian combining retrieval-augmented detection with a validated, iterative patch-generation loop on the basis of these principles and defines evaluations standards, datasets and protocol on which this type of system should be judged. The rest of the paper is organized as follows: The literature survey starts with reviewing the methodology and research questions, introduces the taxonomy and thematic analysis and ends with discussions on limitations of the literature survey, including bibliometric trends, a quantitative synthesis and the social-welfare issues of the literature survey. Section 3 presents the proposed CodeGuardian system, followed by discussions on the advantages and social-welfare implications in Section 5 and social-welfare implications and a future research roadmap in Section 7, and Section 8 concludes.

II. LITERATURE SURVEY

The literature survey is shown in this section. The methodology employed in the review is explained and

the method, which is systematic but adapted to this specific review, is presented systematically following guidelines for systematic reviews (PRISMA) as follows: Research questions, search strategy and data sources, inclusion and exclusion criteria, process of selection that led to the resulting corpus, comparison with previous works on the issue, and threats to validity. We then classify the chosen study in taxonomy and analyse it thematically.

2.1 Research Questions

The review is organized around four research questions:

- RQ1. How are large language models currently applied to software vulnerability detection, assessment, and localization, and what architectural patterns recur?
- RQ2. To what extent can LLMs generate and validate correct security patches, and how is patch quality established?
- RQ3. What are the principal limitations in reasoning, data quality, and evaluation that undermine the reliability of current approaches?
- RQ4. How can retrieval-augmented, validation-driven architectures address these limitations in an integrated, developer-facing system?

The thematic analysis in this section and the proposed design in Section 3 answer RQ1, RQ2, and RQ4, while Section 4 consolidates the evidence relevant to RQ3.

2.2 Search Strategy and Data Sources

We located six digital libraries covering the venues that are most relevant to software engineering and security fields: IEEE Xplore, the ACM Digital Library, ScienceDirect, SpringerLink, Scopus, and Google Scholar. To perform the search, Boolean strings of “large language model” OR “LLM” OR “deep learning” AND “vulnerability detection” OR “vulnerability repair” OR “program repair” OR “secure code generation” OR “patch generation” were searched. Queries used on titles, abstracts and keywords. The window from January 2019 to early 2026 was selected to balance and represent both the deep learning detectors that came before the era of LLMs and the growing amount of work related to LLMs in 2024-2025. The database search was complemented by backward / forward search in the reference lists of the most suitable studies conducted in order to discover key/trigger papers. In addition to this, the database search, backward/forward search for the most relevant papers in the reference lists and citations was performed to uncover "key/trigger papers."

2.3 Inclusion and Exclusion Criteria

The study had to meet all the following criteria: it had to be peer-reviewed or baseline a well-established preprint with significant technical detail, it had to have centered on one of the following detection, assessment, localization, repair, or secure generation of software vulnerabilities; it had to have employed a language model, deep-learning, or retrieval-augmented technique; it had to have reported a concrete method or empirical evaluation. Studies in languages other than English, as short abstracts, posters and editorials without the writing of a technical contribution, as limited focus only on problems of security and/or duplication of previous versions of the same work, were rejected. Multiple papers reporting on the same group presenting the same findings were gathered in the most complete paper. They are not primary studies, but such authoritative standards or datasets scoring systems, weakness taxonomies, and public vulnerability databases are used when creating the survey and were used in the development of the proposed system, and were kept as references.

2.4 Study Selection

A summary of selection process is given in figure–1 It produced a total of 1,284 records in the databases searched and 36 more were located by snowballing. Following duplicate removal, 968 records were screened on title/abstract and 812 of these were ruled out as being clearly out of scope. All the remaining 156 articles received a full-text assessment, 101 of which were not LLM-based, did not have a focus on vulnerability and security, or did not make any technical contribution or did not permit any technical reports. This resulted in a final list of 55 references, consisting of 51 basic works and 4 authoritative norms/databases. The reported numbers reflect the screening funnel; their absolute size in the database varies depending on the date of the query and the entire search log can be obtained from the authors.

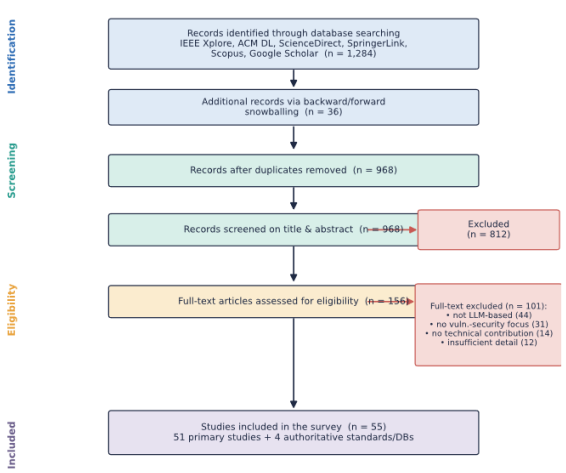

Fig. 1. PRISMA-style study-selection flow, from database identification through screening and eligibility to the final set of 55 included references.

2.5 Comparison with Existing Surveys and Research Gap

Several good surveys already cover portions of this landscape and the importance of understanding how it is different must be understood in order to define the contribution of the present work. The results of broad syntheses contain LLM techniques used for software security [9] and previous systematic reviews present machine-learning techniques, features, and datasets prior to the LLM boom [10]. Existing reviews consider the general landscape of LLM applications in cybersecurity across multiple tasks and threat models [11] or dive deep into the landscape of source-code detectors for deep-learning techniques, their datasets and gaps in reproducing their results [12]. Moving to the remediation, one survey is about remediation through patch generation LLM based perspective [13] and another one is about mapping the taxonomies, design paradigms, and applications of LLM-based automated program repair [14]. Both detection and repair are addressed for the road-ahead review, as well as challenges that remain open [8].

None of these reviews can even bring together and correlate detection, severity assessment, localization, validated patch generation, secure code generation and then translate the synthesis to a system design that one can actually implement and evaluate with a proper evaluation Procedure with a retrieval augmented system design. The current surveys mostly focus on either detection or repair and do not attempt to formulate an architecture. Reviewing the literature surveyed in this review, we do not know of any previous combination of these strands, and derive from them a pipeline validated and auditable, which is the novelty of this survey. The scope of representative surveys is compared with the present work in Table I.

Table I. Scope of representative surveys versus the present work. "Partial" denotes coverage that is incidental rather than a primary focus.

Survey	Detection	Repair	Secure gen.	Evaluation	Proposed architecture
Road ahead [8]	Yes	Yes	Partial	Yes	No
LLMs in SW security [9]	Yes	Partial	Partial	Yes	No
ML detection SLR [10]	Yes	No	No	Yes	No
LLMs in cybersecurity [11]	Yes	Partial	Partial	Yes	No
DL detection review [12]	Yes	No	No	Partial	No
APR patch survey [13]	No	Yes	No	Yes	No
LLM-APR survey [14]	No	Yes	Partial	Yes	No
This survey (CodeGuardian)	Yes	Yes	Yes	Yes	Yes

2.6 Threats to Validity

As with any review, there are a number of threats to this study. Selection bias can occur due to the selection of databases and search terms; although the databases we used and the snowballing technique minimized this, papers on the same ideas in other journals, abstracts, or abstracting indexes may have been overlooked. A tendency is to report significant performance increases which are the most psychologically appealing, the so-called publication bias where negative or null results tend to go under-reported; studies where performance drops when using

realistic data will be highlighted. Preprints and the most recent papers are at risk of indexing bias given there may be changing metadata and page numbers. Patents, industrial "white" papers, and non-English research are not included, and only the English-language publications are considered (mostly peer-reviewed publications). Finally, assigning a study to a taxonomy branch and a theme is subjective; we tried to be as consistent as possible with mutually exclusive definitions and cross-checking some cases on the borderline of taxonomy and theme classifications but we did not succeed in eliminating every possible source of inaccuracy; other readers may disagree on some points with us in this classification. The threats constrained the conclusions and did not discard them.

2.7 Taxonomy of LLM-based Vulnerability Analysis

Having set up a corpus, we group the work surveyed into nine thematic strands which span the field from foundational surveys and the extension of knowledge-based detection, to the various types of classes of weaknesses, the formal and detailed evaluation, the adaptation of patches and the sets of data that allow for the project to be measured all along the way. The resulting taxonomy is tidy up and summarized in Figure 2 is organized along 5 dimensions described below.

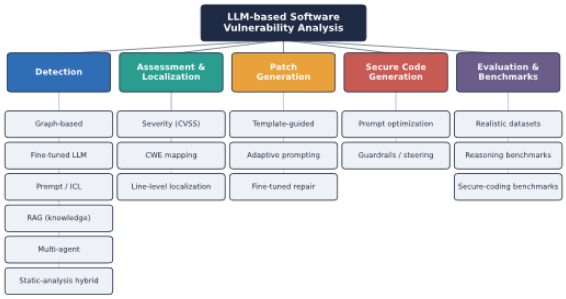

Fig. 2. Taxonomy of LLM-based software vulnerability analysis, spanning detection, assessment and localization, patch generation, secure code generation, and evaluation.

The taxonomy has five top-level branches as this is where the key decision points in a security workflow lie; all studies that were surveyed contain at least one of these BNs. Detection involves determining whether any weaknesses exist, and it wraps up all the techniques that constitute the bulk of the corpus graph-based, fine-tuned, prompt-based, retrieval-augmented, multi-agent, and static-analysis-hybrid. However, the assessment and localization focused questions are what is the severity of confirmed weakness, where is it, and include CVSS-like scoring, CWE mapping and line-level localization. There is-patch generation which questions about how to repair, representing template-guided, adaptive-prompting, and fine-tuned repair. Generating code is deliberately separated; a recent body of research indicates that models that generate code can create new vulnerabilities, and so can be a first-class problem of preemption. Evaluation and benchmarks comprise the fifth branch as the lesson learned in the survey is that measurement discipline is the key determining factor of the other four branches. The following subsections fill-in these branches and, to a certain degree, compare approaches where possible, but do not just describe approaches.

2.8 Surveys, Knowledge-Enhanced Detection, and the Threat Landscape

A first group of work steps returns to the task of systemizing the field. A survey paper of patch generation in automated program repair, where the authors state that simply using the LLM with premature prompts or ignoring all the fine-tuning is not enough [13]. In tandem with this, knowledge-enhanced detection has emerged as a special approach: Vul-RAG trains an LLM with a retrieval function that provides inferred knowledge as historical documents to the LLM inference to boost the accuracy and interpretability while retaining the generality of training, with no need to update the LLM training parameters [15]. The coverage is limited by the quality of the knowledge base, its main virtue being to provide some degree of groundedness in an explicit, updatable knowledge base. The inclusion of LLMs in software security is also the subject of broader synthesis efforts that highlight detection methods and insights that have proven common [9] and another systematic review that captures the pre-LLM machine learning algorithms, features and datasets [10]. To ward off these potential threats, two reviews raise red flags about the security and privacy risks posed by the use of LLMs, such as prompt injection, where LLMs manage to produce more suitable responses to harmful queries than to benign ones, and risks to training data from which LLMs are trained [16]; as well as about the role of LLM security-API misuses, which tend to be easy to miss and highly susceptible to exploitation

[17]. Together, these outputs show that deriving the ability to ground ideas in external knowledge via external tools and having a conscious awareness of the dangers LLMs pose is essential to trustworthy tools.

2.9 Detection Across Systems, Firmware, and Language-Specific Codebases

There's a second thread that takes detection to the wild frontier. While the underlying taint analysis relies on summaries of functions that may not be toolchain-independent, this taint model also enables analysis of command-injection vulnerabilities through opaque binaries, a drawback that is compensated by an LLM-driven analysis of library functions that traditional analyzers are unable to do [18]. VulTR generalizes detection by means of the multi-layer key-feature-enhancement network which progressively filters the key features that are most relevant to security, at the cost of a more complex training procedure [19]. Language-specific tuning also matters: SecureQwen specializes a large model for Python, benefitting from the peculiar features of Python language and library semantics: It boosts precision in Python, but not transfer to other languages [20]. In addition to binary detection, prompt-tuning with continual learning has also been used for vulnerability assessment, enabling a model to learn new types of vulnerabilities over time with only minor losses of previously acquired knowledge [21].

2.10 Prompting Frameworks, Graph Structure, and Early Repair

A third thread narrows down the invocation of LLMs. Specificity of a learnt extract is achieved via a signal from a specialized detector, which allows DLAP to learn to condition the model on the signal. A deep-learning-augmented prompting framework, DLAP, marries learned features with reasoning of a general model [22]. Following the same approach, GRACE also obtains accuracy improvement by augmenting the prompt with graph knowledge: it maps control and data flow relationships between the data, providing explicit information for the model, with the caveats that it involves additional overhead in prompt construction and requires high-quality auxiliary detector or auxiliary examples. LLM-based patch filtering is directly applied to data quality, to filter out unwanted noisy patch commits that contain CVE fixes [24]. For repairs, mining and pattern matching methods have been used to learn and apply patterns from past repairs to similar repairs [25] and transfer learning with Transformer models have been used to transfer knowledge from resource-rich projects to those with limited data [26]. The challenge of severity prediction is addressed using a bimodal multi-task approach which learns from code and textual descriptions of code [27]. The collection of this strand reveals that directives for the model, as well as additional input that inform it, can be as important as the model itself.

2.11 Practical Prediction, Localization, and Multimodal Fusion

The fourth strand looks at what practitioners, as practitioners, need. Recognizing that such a simple vulnerable/non-vulnerable classification is not sufficient, one stream of work aims to make better prediction feasible by combining detection, categorization and localization learning when a file is vulnerable and which vulnerability and location [28]. A multi-modality approach using a unified processing method uses a representation that combines token sequences, graphs and metadata for more convincing detection [29], but introducing the view fusion adds to model and data-engineering complexity. LocVul devises line-level localization as sequence-to-sequence problem to find out exact lines that should be considered for localization [30]. CodeSAGE combines multiple features, annotated by code, into a single graph with the attention mechanisms to capture attention in order to sharpen detection [31] and a generalized zero-shot classifier also extends coverage to weakness types that do not have annotations for during training a critical property as new CWE categories constantly appear [32].

2.12 Benchmarking and Systematic Evidence

A fifth strand undertakes the question of how effective any of this is. The only thing that makes single number comparisons in papers unreliable is that there is a considerable variation in the models and prompt strategies employed in a comprehensive benchmarking study, which uses a variety of LLMs under a common protocol [33]. A systematic literature review of LLMs in cybersecurity provides an overview of the evidence on a task-by-task, threat-by-threat basis, identifying both areas of strength and areas of weakness in the evidence [11]. A parallel

review of deep-learning-based source-code detection unifies the architectures, datasets and evaluation techniques, and identifies issues with reproducibility which hinder inter-study comparisons [12]. Together, such benchmarking points raise issues about comparing raw scores across papers and illustrate that reported performance is highly protocol dependent.

2.13 Specialized Weakness Classes

A sixth strand focuses on areas of low coverage, high impact defect types. To assess the capabilities of cryptographic-API-misuse detectors, the tools are challenged with realistic misuse by systematically injecting cryptographic-API misuse patterns, thereby showing the detectors weaknesses [34]. LLMs that reason about context in software repositories can be used to find hard-coded credentials, a seemingly benign yet pervasive vulnerability [35]. Web applications have a hybrid method of detecting SQL-injections, combining content matching and deep learning techniques for precision and generalization to obfuscated payloads [36]. These specialized detectors are of high-class level accuracy, but not as comprehensive as a general assistant would need to be.

2.14 Rigorous Evaluation and the Reasoning Gap

Some seventh strand questions the understanding of security in LLM. When measured in all the same important ways, one study shows gains in code language models dry up when measurements are carefully done, in a contaminated setting [37]. The added complexity and increased overhead of the analysis tool chain imposes a significant performance sacrifice, but IRIS is able to additionally improve the detection of whole-repository issues by combining an LLM with static analysis to reason over facts in the program, rather than over the raw text itself [38]. For repair, the LLM is only fed through template-guided program repair, which prevents it from making hallucinated edits while reducing the space of expressible fixes [39]. However, there are two notable evaluations that temper expectations: a unified framework revealed that LLM can be wrong when faced with complex tasks of detecting and addressing security threats [40] while another study on the nature of structural and semantic reasoning explicitly demonstrated that LLM's often fail to give a proper answer to the probing questions of SV-TrustEval-C because they may only be giving superficial answers, rather than deep understanding [41]. Another work analyzes when an LLM fails or succeeds in spotting security problems and finds out what conditions influence the performance of LLMs [42]. When taken together these evaluations form a grim conclusion: that while models may frequently be doing well for the wrong reason, there can be no substitute for checking the model with program facts.

2.15 Adaptive Patching, Steering, and Secure Generation

Safe Generation and Remediation is an 8th strand. APPATCH introduces automated adaptive prompting which adjusts prompts according to the defect; compared to static prompts, APPATCH patches real-world defects with higher reliability [43]. Training at the level of activation and prompt provides guidance to LLMs to steer the models in the right direction toward correct detection behavior [44]. PromSec optimises prompts to encourage LLM to provide functional but secure source code, not only reviewing but writing code, given the growing use of LLMs for the latter [45]. This is a sound concern in that a controlled user study shows that one might be able to write less secure programs by allowing a developer to be aided by AI [46] and similarly for systems programming it shows the same conclusion as the Lost at C study [47]. The results bring together the idea of securing generation and detecting/repairing generation, which is discontinuable.

2.16 Datasets for Training and Evaluation

An additional strand provides the empirical support. The MoreFixes mining project is introduced for scaling up the repository discovery process to mining commits that fix the CVE [48] to produce a large knowledge base of vulnerabilities and fixes, know them and their respective location. In order to assess the code generation methods based on machine learning, SecurityEval collects examples of vulnerabilities and offers a generic measure of secure code generation [49]. This persistent pressure to capture the reality of data and the dataset size is driven throughout this survey, toward CodeGuardian's approach of retrieval and verification over memorized labels.

2.17 Comparative Synthesis

The nine strands taken together tell of a field merging together to a handful of principles. Grounding into and

out of the structure and knowledge is always richer and more useful than using it as flat text [7, 15, 23, 31]; and explanation and localization steps are to become first class citizens, rather than late-thought-out side effects [3, 28, 30] and brittleness is to become problematic not with respect to convenient benchmarks, but rather with respect to realistic tests, that sustain reason for and reason about [5, 37, 40, 41]. On the remedial side, there is a growing move to having alternative output guided by a template and/or validation, where hard-won evidence has suggested that fluent output is different to correct or secure output [46], [47]. There is just one other frictional issue: these advances exist in isolated prototypes, are tested on different data, and there is hardly any shared tooling. In Table II, representative's primary studies are compared along task, technique, strengths, and limitations, thereby bringing to light this complementarity that CodeGuardian takes advantage.

Table II. Comparative analysis of representative primary studies across task, technique, strengths, and limitations.

Ref	Approach	Task	Technique	Strengths	Limitations
[7]	Multi-graph rep.	Detection	Fused code graphs	Captures structure sequence models miss	Graph construction cost; language-specific parsers
[15]	Vul-RAG	Detection	Knowledge-level RAG	Interpretable; no retraining; updatable	Bounded by knowledge-based coverage
[22]	DLAP	Detection	DL-augmented prompting	Combines detector recall with LLM reasoning	Depends on auxiliary detector quality
[23]	GRACE	Detection	Graph + in-context	Injects flow structure and examples	Prompt-construction overhead
[30]	LocVul	Localization	Seq-to-seq	Pinpoints vulnerable lines, not just files	Sensitive to labeling granularity
[38]	IRIS	Detection	LLM + static analysis	Reasons over verified facts; whole-repo	Heavier static-analysis toolchain
[39]	Template repair	Patching	Template-guided LLM	Curbs hallucinated edits	Limited to expressible templates
[43]	APPATCH	Patching	Adaptive prompting	Tailor's instructions per defect	Prompt search cost; needs validation
[45]	PromSec	Secure gen.	Prompt optimization	Elicits functional yet secure code	Optimization per prompt/context
[41]	SV-TrustEval-C	Evaluation	Reasoning benchmark	Exposes shallow-cue reliance	Focused on C; diagnostic, not a detector

III. PROPOSED SYSTEM: CODEGUARDIAN

Take the lessons from the survey and implement them in CodeGuardian's workflow. It's based on the following motto: A trustworthy security helper should rely on authoritative knowledge and report it precisely, quantify its impact, and never rely on a patch that hasn't been verified. The architecture, shown in figure 3, is divided into two segments: detection and remediation, joined by a knowledge base and by a self-refinement loop.

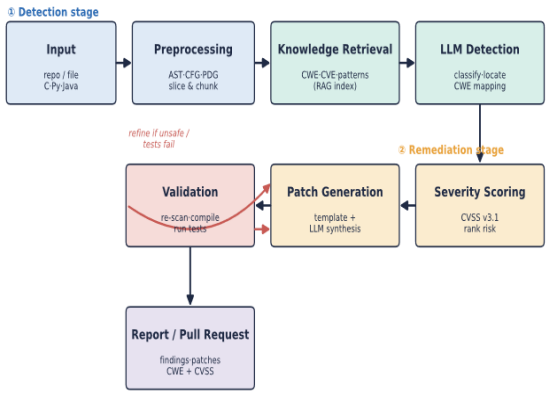

Fig. 3. CodeGuardian architecture. A retrieval-augmented detection stage feeds a remediation stage in which candidate patches are validated and iteratively refined before a report or pull request is emitted.

3.1 Design Principles and Component Justification

Each element of CodeGuardian is not simply a "desirable", but truly "necessary" according to a certain finding of the survey and this is explicitly stated in Table III. It is a conscious effort to make the design modular so that the individual components can be swapped out as the field progresses. Largely, complexity is determined by LLM inference itself, structural preprocessing and retrieval incur linear increases in complexity as per the input size, and the self-refinement loop preserves inference cost by number of iterations (with a small bound that can be configured). If a project doesn't have automated tests, validation becomes a checker which rescans and compiles the project's files in a state where the feature is being integrated into.

Table III. Justification of each CodeGuardian component by the survey finding that motivates it.

Component	Motivation (survey finding)	Supporting literature
Structural preprocessing (AST/CFG/PDG)	Graph and structural models outperform flat-text detection	[7], [23], [31]
Knowledge retrieval (RAG)	Grounding in explicit knowledge improves accuracy and interpretability	[15]
Detection with CWE mapping + localization	Actionable output requires which weakness and were, not just a label	[28], [30]
Static-analysis coupling	Reasoning gap: models exploit shallow cues without verified facts	[38], [40], [42]
Severity scoring (CVSS)	Risk-based triage focuses effort on the most exploitable defects	[50], [27]
Template-guided + adaptive patching	Constraining generation curbs hallucinated, unsafe edits	[39], [43]
Validation loop	AI-assisted code can silently introduce new flaws	[46], [47]
Secure-generation evaluation	Quality must be measured on functionality and security jointly	[45], [51]

3.2 Grounding in Authoritative Security Knowledge

Instead of relying only on the information it has extracted from a training set that is stored in a single state, CodeGuardian's decisions are based on valuable, versioned security knowledge. The severity is measured with Common Vulnerability Scoring System (CVSS) v3.1 with their vector-based metrics which rank findings base on the exploitability and impact in addition to the confidence of the model [50]. The retrieval index is seeded with the OWASP Top 10, representing the most critical category of web-application risks [52] and the CWE Top 25 Most Dangerous Software Weaknesses, which is a stable, hierarchical vocabulary for classifies defects [53]. The examples and remediation guidelines are taken from the National Vulnerability Database (NVD) that serves as the authoritative public catalog of vulnerabilities disclosed so far [54]. By embedding these resources as a retrieval-augmented layer, inspired by the knowledge-grounded approach of Vul-RAG [15] the model can explain each verdict's specific weakness pattern and enhance its accuracy and auditing capabilities.

3.3 Detection Stage

The source files and/or the whole repository are then mapped to structural representations (AST, CFG and PDG) and their resulting slices are semantically

organized to allow for the processing of long files in the model context without losing any data-flow information. Together, all three representations make evident the relationships in terms of how the program behave during execution, which a flat token sequence does not show. This structural preprocessing is similar to many token-based graph-centric detectors which outperform flat token models [7] [23] [31]. All the organized chunks are further paired with the most relevant retrieved knowledge, and then fed into the LLM detection module, which carries out three coupled tasks: detecting the presence or absence of a weakness, mapping it to a specific CWE category, and localizing it in the specific lines, akin to practical but actionable predictions in [28] and [30]. To reduce the gap in the reasoning process with respect to reported literature [40, 42] where LLM only works on the superficial text, the module is conditionally complemented with a lightweight static analyzer to support the LLM in its reasoning process by providing properties of the program, like borders, instead of text.

3.4 Remediation Stage and Self-Refinement Loop

Confirmed results from the findings feed into the remediation. Their severity is initially rated based on CVSS ensuring that limited developer effort goes towards the most serious defects. The patch-generation module then proposes a candidate fix by mimicking a repair shape with synthesizer-guided strategy, while incorporating adaptive prompting to progressively guide instructions towards the repair instance of the defect as in APPATCH [43], which constrains the model to repair shapes that are proven to be correct by the previous examples. Instead of releasing the first candidate, CodeGuardian cycles back: each patch is re-scanned by the detection stage, compiled and exercised with the test suite of the project. The iterative self-refinement is shown in figure 4. If a patch causes a failure by adding weak security or compromising functionality, the failure signal is used as an input to the generator which comes up with another proposed patch. Only patches that successfully pass validation are surfaced as a report or as an automated pull request, and thus are never sent to the developer.

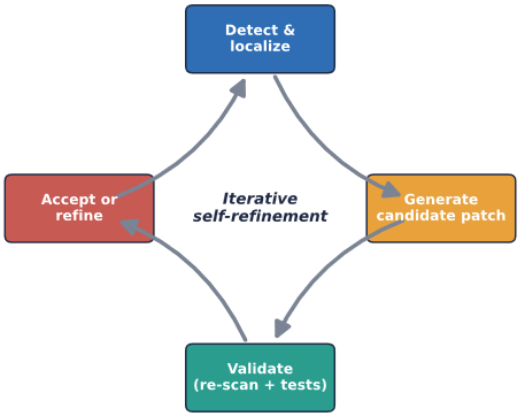

Fig. 4. The closed-loop validation workflow. Candidate patches are validated and refined iteratively until they are both secure and functionally correct.

3.5 Illustrative Example

The pipeline is able to deal with a canonical example as listed in 1. Additional details are added by the detection stage into the string-concatenated query and a corresponding pattern of weakness, remediation advice is retrieved, a high CVSS Severity is added by the detection stage and finally the remediation stage generates a parameterized query as a patch. Rewriting eliminates the weakness and maintains behavior as confirmed by validation. An example is provided for illustrative purposes only, and is not an actual experimental result.

```
# --- Detected (CWE-89: SQL Injection, CVSS High) ---
def get_user(uid):
    q = "SELECT * FROM users WHERE id = '" + uid + "'"
    return db.execute(q)


# --- CodeGuardian patch (validated) ---
def get_user(uid):
    q = "SELECT * FROM users WHERE id = %s"
    return db.execute(q, (uid,)) # parameterized query
```

Listing 1. A SQL-injection weakness detected and remediated by CodeGuardian.

3.6 Evaluation Protocol

The survey's primary point is that the cell should not rely on a lab score as a deployment criterion, so CodeGuardian is designed to be evaluated based on two protocols, as detailed below in Table IV. Detection is tested on realistic corpora, which are deduplicated (and traceable), with precision, recall and false-positive rate at both file and line level, using the traceable CVE-fix corpora (MoreFixes [48]) and secure code (SecurityEval [49]). Secure-generation and patch quality are evaluated using CyberSecEval [55] or CWEval [51] with a positive reward given to a fix only if it still is functionally correct. This design is clearly resistant to the most prevalent mode of failure reported by the survey as the field's most critical problem high benchmark scores which don't hold up in field deployment. Additional testing of the proposed CodeGuardian framework is beyond the scope of this survey and system-design paper and thus is left for future work.

Table IV. Standards, datasets, and benchmarks underpinning CodeGuardian's evaluation protocol.

Resource	Type	Purpose in CodeGuardian	Availability
MoreFixes [48]	Dataset (CVE fixes)	Detection/repair training and audit	Public
SecurityEval [49]	Dataset (secure code)	Evaluate security of generated/patched code	Public
CVSS v3.1 [50]	Scoring standard	Severity scoring and risk-based triage	Open standard
OWASP Top 10 [52]	Risk taxonomy	Web-risk categories in the knowledge base	Public
CWE Top 25 [53]	Weakness taxonomy	Stable vocabulary for classification	Public
NVD [54]	Vulnerability database	Concrete instances and remediation guidance	Public
CyberSecEval [55]	Benchmark	Secure-coding evaluation of the model	Open source
CWEval [51]	Benchmark	Joint functionality-and-security evaluation	Open source

3.7 Positioning Against Industrial Tools

The goal of CodeGuardian is to augment, not supplant the industry standard tools that are already in use. Platforms like Rule-based static analyzers like CodeQL, Semgrep, SonarQube, to name a few, and commercial products like Snyk, have mature integrin and have an appalling number of false-positives and false-negatives for a known pattern. However, they are based on hand-written rules and cannot detect semantically new and/or context-specific defects. Pairing developers with AI tools like GitHub Copilot speeds up coding, but can also cause an increase in insecure coding, just as was hypothesized in controlled trials [46],[47]. CodeGuardian differs in three ways: provides a knowledge-based CWE and NVD-based “auditable rationale” behind every finding, not an ambiguous rule hit; ties detection, severity, localization and re-scanning and testing of repair together in one bundle, rather than having to follow the remediation code up a single pathway; never exposes a fix that is not re-scanned and tested. It can be implemented concurrently with a normal scanner and the precision of the rule hits of the scanner can serve as further evidence for the retrieval layer.

IV. RESULTS AND DISCUSSION

This is a survey and system-design document and thereby we give only a bibliometric synthesis of the chosen literature, a qualitative aggregation of the used methods and a discussion on the implications of the evidence that we have used in the LUcket of RQ1–RQ3. The bibliometric measures were retrieved directly from the metadata of the 55 selected references: the values were acquired using the already entered metadata from the venue rather than from the original articles, and each study was only assigned to the dominant theme amongst the defined themes provided in Section2, in the case of borderline, the dominant theme was the main claim stated in the original study. The temporal distribution is in figure 5. Activity is concentrated in 2024 and 2025, and the majority of the works can be attributed to LLM-based vulnerability analysis, which is a still-forming and quickly evolving area where best practice continues to take shape.

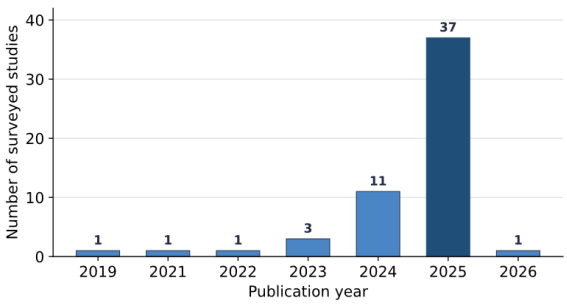

Fig. 5. Distribution of the 55 selected studies by publication year, extracted from venue metadata.

Figure 6 shows the breakdown of the corpus according to the research theme. That's shifting toward a more balanced overall distribution between detection and end to end, measurable remediation, with a significant and growing portion of work devoted to datasets, benchmarks, and standards, followed by surveys, assessment and localization, and patch generation. Some of the numbers are rounded off due to a small number of studies overlapping two themes and receiving allocations to the focus they are most clearly related to.

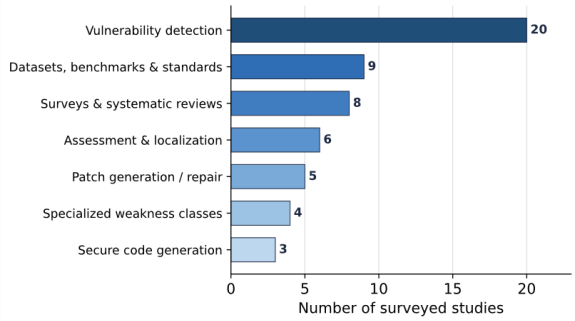

Fig. 6. Distribution of the 55 selected studies by primary research theme.

It's natural to ask if the results gathered from the different experiments can be aggregated into a single figure, in this case an average F1 or accuracy. Mistake fully, we do not do so, and that's a find which answers Q3 itself. The corpus reports results obtained on data types, definitions, labels (languages), languages, and metrics that differ in kind and an experiment that turns results into their realistic condition would show their scores to collapse making it misleading to average results over such incommensurable protocols. Rather, Table V brings together the corpus in the dimensions consistently subject to reporting, model families, target languages, common datasets, and evaluation metrics in a way that avoids positing an artificial point estimate of the methodological landscape.

Table V. Qualitative aggregation of the surveyed corpus. Heterogeneous protocols preclude a valid pooled performance figure.

Dimension	Commonly observed options across the corpus	Representative refs.
Model families	Encoder models (CodeBERT/GraphCodeBERT); decoder LLMs (GPT-family, CodeLlama, Qwen, Falcon-tuned); graph neural networks	[2], [20], [7]
Target languages	C/C++, Java, Python, Solidity, and web languages (SQL, PHP)	[20], [18], [6], [36]
Common datasets/benchmarks	Devign, BigVul, DiverseVul, PrimeVul, MoreFixes, SecurityEval, CyberSecEval, CWEval	[48], [49], [55], [51]
Tasks	Detection, CWE classification, line-level localization, severity assessment, repair, secure generation	[28], [30], [27]
Evaluation metrics	Accuracy, precision/recall/F1, MCC, false-positive rate, pass@k, CWE-match rate	[33], [5], [51]

Three of these are what we term as "cross-cutting findings" that cut across the corpus; and provide the answers to RQ1 and RQ3 to guide the conception of CodeGuardian Three of these are called “cross-cutting findings”, they cut across the corpus and directly guided the conception of CodeGuardian in answering RQ1 and RQ3. First, there is a reasoning gap between benchmark accuracy and genuine understanding of security: multiple rigorous studies reveal that there are models that exploit shallow cues (confidence-based) and that reported gains are reduced under conditions of contamination control [5, 37, 40, 41]. In response CodeGuardian bases detection on knowledge garnered from the retrieved information and known program properties instead of memorizing labels. Second, data quality is fragile: realistic datasets deflate performance [5] and even fix-commit corpora need particulary filters [24], [48] especially to be able to evaluate, discern, and push retrieval and validation over ever

larger training sets. Third, code generation and code detection are mutually intertwined with each other: assistants who write code can introduce defects without anyone's knowledge [46], [47] and a code remediation must be validated, as it could not be assumed [39] [43].

A recurring practical theme to the answering of part of RQ2 is the cost of a false positive. A detector that generates too many false alarms becomes useless, and the data show that while precision is the key to adoption in the industrial market, record breaking accuracy is not [33], [12]. CodeGuardian's retrieval layer supplements this by assigning each alert to an actual, documented weakness pattern ensuring there is no low evidence guess, while using the CVSS to rank alerts ensures that only those which have impact are treated. There's another "deciding factor" on the remediation side: A patch that fails to come through the validation loop, theoretically, never is shown. Put another way, CodeGuardian gives up a certain amount of recall for a great amount of precision and reliability: a bargain that many writers say is the correct one in terms of real-world deployment.

4.1 Limitations

In addition to the threats to validity related to the review process that were covered in Section 2.6, the following are two other limitations to note. First, the field is changing so rapidly that any snapshot is obsolete. Several conclusions made a year ago now are being redrawn, and some of the works referred to are quite recent, so their metadata could change. Second, while there is much evidence of the best things done in codes which can be accessed (for open-source and benchmark codes in particular), much of the strongest empirical evidence is in codes from other environments from which no reproductive access is provided (in closed proprietary, industrial codebases, the success shown by researchers has not been proven external to such code environments). The lack of these rules extends the paper's main point that claims of capability need to be accompanied by a disciplined and realistic validation process.

V. ADVANTAGES OF THE PROPOSED SYSTEM

Several real benefits are offered by CodeGuardian's architecture over classical tools and single monolithic LLM detectors. This means that each verdict can refer to the specific weakness pattern and NVD precedent that has been used to explain the verdict, which provides developers with an auditable reason for issuing the verdict, avoiding the "no big deal" alert fatigue that can be endured by rule-based scanners like CodeQL, Semgrep or SonarQube. Findings are prioritized according to the real world risk using the CWE Top 25 and the CVSS, so the most critical defects are remediated first before the tool has a high level of confidence. The seamless pipeline from detection to localization, to severity and validated repair eliminates the need for teams to pass a task between stand-alone detection and fixing tools. The most significant protection feature is the re-scanning, recompiling, and retesting of each patch before it can be surfaced, which prevents programmers from encountering the documented danger of assistants creating new bugs while seemingly fixing old ones. Last but not least, the structural preprocessor, retrieval layer and validator can be reused as newer or domain-specific models are brought in to meet the fast-growing changes in the field; this measure of course helps optimize a company's investment.

VI. SOCIAL WELFARE OF THE PROPOSED SYSTEM

Security tooling that is reliable and automated can have a high social impact which is widely spread. The impact of vulnerabilities in widely-used software is that those with a lower ability to manage them bear the brunt of the costs: small and medium-size businesses that lack well equipped security teams, public-interest and open-source projects that rely on volunteers to contribute to their development, end users of personal data whose sensitive information is staged under conditions they did not create. These are the categories most “tainted” injection, broken access control, and cryptographic failures are the very categories listed as most critical issues for Web applications [52]. CG's reduction in the knowledge and effort needed to detect and correct bugs could open up a new door to the level of protection typically seen in powerful companies. Rather, embedding in regular development, such as by using automated pull-requests, moves security as early as possible in the lifecycle and lowers the number of systems that can be exploited, a direction that the road-ahead review indicates will be central to its focus [8]. The main benefit of the focus on explainability is

pedagogical: the weakness pattern for every finding provides a great way to teach secure coding principles in the context of insecure code, which is what many AI-supported developers write as they think they are writing secure but, in fact, aren't [46], [47]. We are aware that the same powers can be weaponised, and that LLM-based tools carry their own privacy and security issues [16]; in such a context, CodeGuardian aims to be a defensive agent for developers who can only produce valid fixes, not exploit code.

VII. FUTURE ENHANCEMENT

There are a number of ways to extend CodeGuardian some short term and some long term and these are grouped together in a near-term / long-term research roadmap in Figure 7. There is a need for repository-scale reasoning (RSR) as an open challenge: though chunking and retrieval may help, it is the global relations between different files, which are truly inter-procedural, that require more intimate coupling to program analysis and longer-context models, added to designs like hybrid IRIS [38]. As automation has been shown to be beneficial for smart-contract auditing [6], multi-agent collaboration might allow for specialized agents to discuss detection, patching, and validation in order to minimize the individual agent bias and progress toward agentic repair. It would allow for the knowledge base and model to learn new CVEs without retraining the entire model, and human-in-the-loop feedback could be used to progressively fine-tune the system to the team's standards. Covering beyond the prevailing language of the present datasets with multi-language support. Federated and privacy-preserving training would also enable organizations to learn from each other's vulnerabilities, avoiding the disclosure of their proprietary code, particularly given the privacy concerns that LLMs present [16]. The agenda continues with robust measures against adversarial and obfuscated code, standardised security benchmarks and strides towards autonomous and verified remediation. Additional developer surveys of LLM-based program repair identified various ways of fixing code, including multi-hunk repairs and semantic repair, which could be integrated by CodeGuardian [14]. Both sides on each end stick to the core message of the paper, viz. "capability must be validated.

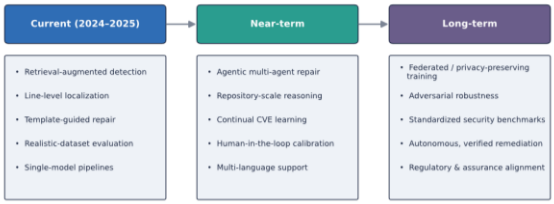

Fig. 7. A research roadmap for validation-driven, LLM-based vulnerability analysis, from current capabilities to long-term goals.

VIII. CONCLUSION

With the advent of large language models, software vulnerability analysis is no longer about a mere pattern match; it is about true semantic reasoning and the potential of tools that can detect and correct vulnerabilities. The paper surveyed and selected 55 references, created a taxonomy of detection, assessment and localization, patch generation, secure code generation and evaluation and was able to arrive at three enduring lessons drawn from this pool: that benchmark accuracy remains on the verge of understanding, training data is fragile and a firm reliance on authoritative security knowledge is decisive. From those lessons, we developed a unified pipeline for retrieval-augmented, structurally informed loss detection that is linked to a validated, self-refining loop for generating patches, CodeGuardian, and we defined the standards, datasets, and bi-evaluation protocol for what such a system should be evaluated on. We also recognized that the study has limitations: The corpus is a snapshot of a rapidly changing field, that of proprietary codebases is generally infrequent, and the thematic classification is one of judgments. The danger of the field is over-claiming and the remedy is disciplined validation at each step, the survey emphasizes, within a field that has these bounds. The frontier is moving from whether an LLM can help in security towards whether it can be trusted to. CodeGuardian is provided as a design to build, measure, and constantly re-evaluate. The proposed framework will be tested through implementation and empirical research in the future.

REFERENCES

[1]. X. Yin, C. Ni, and S. Wang, “Multitask-Based Evaluation of Open-Source LLM on Software

Vulnerability," IEEE Transactions on Software Engineering, vol. 50, no. 11, pp. 3071–3087, Oct. 2024, doi: 10.1109/TSE.2024.3470333.

[2]. M. A. Ferrag et al., "SecureFalcon: Are We There Yet in Automated Software Vulnerability Detection with LLMs?," IEEE Transactions on Software Engineering, vol. 51, no. 4, pp. 1248–1265, Apr. 2025, doi: 10.1109/TSE.2025.3548168.

[3]. Q. Mao, Z. Li, X. Hu, K. Liu, X. Xia, and J. Sun, "Towards Explainable Vulnerability Detection with Large Language Models," IEEE Transactions on Software Engineering, vol. 51, no. 10, pp. 2957–2971, Oct. 2025, doi: 10.1109/TSE.2025.3605442.

[4]. K. Huang, J. Zhang, X. Bao, X. Wang, and Y. Liu, "Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair," IEEE Transactions on Software Engineering, vol. 51, no. 4, pp. 904–928, Apr. 2025, doi: 10.1109/TSE.2025.3532759.

[5]. P. Chakraborty, K. K. Arumugam, M. Alfadel, M. Nagappan, and S. McIntosh, "Revisiting the Performance of Deep Learning-Based Vulnerability Detection on Realistic Datasets," IEEE Transactions on Software Engineering, vol. 50, no. 8, pp. 2163–2177, Aug. 2024.

[6]. Z. Wei et al., "Advanced Smart Contract Vulnerability Detection via LLM-Powered Multi-Agent Systems," IEEE Transactions on Software Engineering, vol. 51, no. 10, pp. 2830–2846, Oct. 2025, doi: 10.1109/TSE.2025.3597319.

[7]. F. Qiu, Z. Liu, X. Hu, X. Xia, G. Chen, and X. Wang, "Vulnerability Detection via Multiple-Graph-Based Code Representation," IEEE Transactions on Software Engineering, vol. 50, no. 8, pp. 2178–2199, Aug. 2024, doi: 10.1109/TSE.2024.3427815.

[8]. X. Zhou, S. Cao, X. Sun, and D. Lo, "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," ACM Transactions on Software Engineering and Methodology, vol. 34, no. 5, pp. 1–31, May 2025, doi: 10.1145/3708522.

[9]. Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights," ACM Computing Surveys, vol. 58, no. 5, pp. 1–35, Nov. 2025, doi: 10.1145/3769082.

[10]. N. Shiri Harzevili, A. Boaye Belle, J. Wang, S. Wang, Z. M. J. Jiang, and N. Nagappan, "A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning," ACM Computing Surveys, vol. 57, no. 3, pp. 1–36, Nov. 2024, doi: 10.1145/3699711.

[11]. J. Zhang et al., "When LLMs Meet Cybersecurity: A Systematic Literature Review," Cybersecurity, vol. 8, no. 1, Feb. 2025, doi: 10.1186/s42400-025-00361-w.

[12]. H. Su, Z. Xu, Y. Zhang, and Q. Tan, "Source Code Vulnerability Detection Based on Deep Learning: A Review," Cybersecurity, vol. 9, no. 1, Jan. 2026, doi: 10.1186/s42400-025-00518-7.

[13]. Y. Yang et al., "Patch Generation in APR: A Survey from the Perspectives of Utilizing LLMs and Using APR-Specific Information," ACM Transactions on Software Engineering and Methodology, vol. 35, no. 8, pp. 1–62, Jul. 2026, doi: 10.1145/3764584.

[14]. "A Survey of LLM-Based Automated Program Repair: Taxonomies, Design Paradigms, and Applications," arXiv.org, 2022. [Online]. Available: https://arxiv.org/html/2506.23749v1. Accessed: Jul. 19, 2026.

[15]. X. Du et al., "Vul-RAG: Enhancing LLM-Based Vulnerability Detection via Knowledge-Level RAG," ACM Transactions on Software Engineering and Methodology, Feb. 2026, doi: 10.1145/3797277.

[16]. S. Wang et al., "Unique Security and Privacy Threats of Large Language Models: A Comprehensive Survey," ACM Computing Surveys, vol. 58, no. 4, pp. 1–36, Oct. 2025, doi: 10.1145/3764113.

[17]. Z. Mousavi, C. Islam, M. A. Babar, A. Abuadbba, and K. Moore, "Detecting Misuse of Security APIs: A Systematic Review," ACM Computing Surveys, vol. 57, no. 12, pp. 1–39, Jul. 2025, doi: 10.1145/3735968.

[18]. J. Ye, X. Fei, X. de C. de Carnavalet, L. Zhao, L. Wu, and M. Zhang, "Detecting Command Injection Vulnerabilities in Linux-Based Embedded Firmware With LLM-Based Taint Analysis of Library Functions," Computers &

Security, vol. 144, Art. no. 103971, Sept. 2024, doi: 10.1016/j.cose.2024.103971.

[19]. H. He, S. Wang, Y. Wang, K. Liu, and L. Yu, "VulTR: Software Vulnerability Detection Model Based on Multi-Layer Key Feature Enhancement," Computers & Security, vol. 148, Art. no. 104139, Jan. 2025, doi: 10.1016/j.cose.2024.104139.

[20]. A. Mechri, M. A. Ferrag, and M. Debbah, "SecureQwen: Leveraging LLMs for Vulnerability Detection in Python Codebases," Computers & Security, vol. 148, Art. no. 104151, Jan. 2025, doi: 10.1016/j.cose.2024.104151.

[21]. J. Xue, X. Chen, J. Wang, and Z. Cui, "Towards Prompt Tuning-Based Software Vulnerability Assessment with Continual Learning," Computers & Security, vol. 150, Art. no. 104184, Mar. 2025, doi: 10.1016/j.cose.2024.104184.

[22]. Y. Yang et al., "DLAP: A Deep Learning Augmented Large Language Model Prompting Framework for Software Vulnerability Detection," Journal of Systems and Software, vol. 219, Art. no. 112234, Jan. 2025, doi: 10.1016/j.jss.2024.112234.

[23]. G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai, "GRACE: Empowering LLM-Based Software Vulnerability Detection with Graph Structure and In-Context Learning," Journal of Systems and Software, vol. 212, Art. no. 112031, Jun. 2024, doi: 10.1016/j.jss.2024.112031.

[24]. C. Dil, H. Chen, and K. Damevski, "Towards Higher Quality Software Vulnerability Data Using LLM-Based Patch Filtering," Journal of Systems and Software, vol. 230, Art. no. 112581, Dec. 2025, doi: 10.1016/j.jss.2025.112581.

[25]. X. Cao, J. Wang, and P. Wu, "Enhancing Vulnerability Repair Through the Extraction and Matching of Repair Patterns," Journal of Systems and Software, vol. 230, Art. no. 112528, Dec. 2025, doi: 10.1016/j.jss.2025.112528.

[26]. I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, and A. Chatzigeorgiou, "Transfer Learning for Software Vulnerability Prediction Using Transformer Models," Journal of Systems and Software, vol. 227, Art. no. 112448, Sept. 2025, doi: 10.1016/j.jss.2025.112448.

[27]. X. Du, S. Zhang, Y. Zhou, and H. Du, "A Vulnerability Severity Prediction Method Based on Bimodal Data and Multi-Task Learning," Journal of Systems and Software, vol. 213, Art. no. 112039, 2024, doi: 10.1016/j.jss.2024.112039.

[28]. C. Liu, X. Chen, X. Li, and Y. Xue, "Making Vulnerability Prediction More Practical: Prediction, Categorization, and Localization," Information and Software Technology, vol. 171, Art. no. 107458, Jul. 2024, doi: 10.1016/j.infsof.2024.107458.

[29]. W. Cai, J. Chen, J. Yu, W. Hu, and L. Gao, "A Software Vulnerability Detection Method Based on Multi-Modality with Unified Processing," Information and Software Technology, vol. 182, Art. no. 107703, Jun. 2025, doi: 10.1016/j.infsof.2025.107703.

[30]. I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, and A. Chatzigeorgiou, "LocVul: Line-Level Vulnerability Localization Based on a Sequence-to-Sequence Approach," Information and Software Technology, vol. 189, Art. no. 107940, Jan. 2026, doi: 10.1016/j.infsof.2025.107940.

[31]. G. Zhang, T. Yao, J. Qin, Y. Li, Q. Ma, and D. Sun, "CodeSAGE: A Multi-Feature Fusion Vulnerability Detection Approach Using Code Attribute Graphs and Attention Mechanisms," Journal of Information Security and Applications, vol. 89, Art. no. 103973, Mar. 2025, doi: 10.1016/j.jisa.2025.103973.

[32]. J. Hu, J. Guo, C. Luo, Y. Hu, M. Lanzinger, and Z. Li, "Enabling Generalized Zero-Shot Vulnerability Classification," IEEE Transactions on Dependable and Secure Computing, vol. 22, no. 4, pp. 3465–3482, Jul. 2025, doi: 10.1109/TDSC.2025.3532487.

[33]. K. Tamberg and H. Bahsi, "Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study," IEEE Access, vol. 13, pp. 29698–29717, 2025, doi: 10.1109/ACCESS.2025.3541146.

[34]. A. S. Ami, S. Marsden, K. Moran, D. Poshyvanyk, and A. Nadkarni, "Mutation-Based Evaluation of Cryptographic API Misuse Detectors," ACM Transactions on Privacy and Security, vol. 29, no. 2, pp. 1–35, Mar. 2026, doi: 10.1145/3796221.

[35]. C. Biringa and G. Kul, "Detecting Hard-Coded Credentials in Software Repositories via LLMs," Digital Threats: Research and Practice, vol. 6,

no. 3, pp. 1–16, Sept. 2025, doi: 10.1145/3744756.

[36]. Y. Chen, G. Liang, and Q. Wang, "Research on SQL Injection Detection Technology Based on Content Matching and Deep Learning," Computers, Materials & Continua, vol. 84, no. 1, pp. 1145–1167, 2025, doi: 10.32604/cmc.2025.063319.

[37]. Y. Ding et al., "Vulnerability Detection with Code Language Models: How Far Are We?" in Proc. 47th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), Apr. 2025, pp. 1729–1741, doi: 10.1109/ICSE55347.2025.00038.

[38]. Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities," in Proc. 13th Int. Conf. Learn. Representations (ICLR), 2025. [Online]. Available: https://openreview.net/forum?id=9LdJDU7E91

[39]. K. Huang, J. Zhang, X. Meng, and Y. Liu, "Template-Guided Program Repair in the Era of Large Language Models," in Proc. 47th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), Apr. 2025, pp. 1895–1907, doi: 10.1109/ICSE55347.2025.00030.

[40]. U. Saad, M. Han, S. Pujar, P. Hammond, A. Coskun, and G. Stringhini, "LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks," arXiv preprint arXiv:2312.12575, 2023. [Online]. Available: https://arxiv.org/abs/2312.12575. Accessed: Jul. 19, 2026.

[41]. Y. Li et al., "SV-TrustEval-C: Evaluating Structure and Semantic Reasoning in Large Language Models for Source Code Vulnerability Analysis," arXiv preprint arXiv:2505.20630, 2025. [Online]. Available: https://arxiv.org/abs/2505.20630. Accessed: Jul. 19, 2026.

[42]. A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, "Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities," arXiv preprint arXiv:2311.16169, 2023. [Online]. Available: https://arxiv.org/abs/2311.16169.

[43]. Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching." [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity25/sec25cycle1-prepub-1174-nong.pdf. Accessed: Jul. 19, 2026.

[44]. J. Li, L. Cui, J. Zhang, H. Fei, Y. Chen, and H. Zhu, "Steering Large Language Models for Vulnerability Detection," in Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP), Apr. 2025, pp. 1–5, doi: 10.1109/ICASSP49660.2025.10887736.

[45]. M. Nazzal, I. Khalil, A. Khreishah, and N. Phan, "PromSec: Prompt Optimization for Secure Generation of Functional Source Code with Large Language Models (LLMs)," in Proc. ACM SIGSAC Conf. Comput. Commun. Security, Dec. 2024, pp. 2266–2280, doi: 10.1145/3658644.3690298.

[46]. N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?" Nov. 2022, doi: 10.1145/3576915.3623157.

[47]. G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants," in Proc. USENIX Security Symp., 2023, pp. 2205–2222. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval. Accessed: Jul. 19, 2026.

[48]. J. Akhoundali, S. R. Nouri, K. Rietveld, and O. Gadyatskaya, "MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined Through Enhanced Repository Discovery," in Proc. 20th Int. Conf. Predictive Models Data Anal. Softw. Eng., 2024, pp. 42–51, doi: 10.1145/3663533.3664036.

[49]. M. L. Siddiq and J. C. S. Santos, "SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques," in Proc. 1st Int. Workshop Mining Softw. Repositories Appl. Privacy Security, 2022, pp. 29–33, doi: 10.1145/3549035.3561184.

[50]. "Common Vulnerability Scoring System Version 3.1 Specification Document Revision 1," FIRST. [Online]. Available:

https://www.first.org/cvss/v3-1/cvss-v31-specification_r1.pdf. Accessed: Jul. 20, 2026.

[51]. J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, "CWEval: Outcome-Driven Evaluation on Functionality and Security of LLM Code Generation," arXiv preprint arXiv:2501.08200, 2025. [Online]. Available: https://arxiv.org/abs/2501.08200. Accessed: Jul. 19, 2026.

[52]. OWASP, "OWASP Top Ten," Jul. 2025. [Online]. Available: https://owasp.org/www-project-top-ten/. Accessed: Jul. 20, 2026.

[53]. MITRE, "CWE Top 25 Most Dangerous Software Weaknesses 2024," 2024. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. Accessed: Jul. 19, 2026.

[54]. National Institute of Standards and Technology (NIST), "National Vulnerability Database (NVD)," 2019. [Online]. Available: https://nvd.nist.gov/. Accessed: Jul. 20, 2026.

[55]. M. Bhatt et al., "Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models," arXiv preprint arXiv:2312.04724, 2023. [Online]. Available: https://arxiv.org/abs/2312.04724. Accessed: Jul. 19, 2026.