TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Anton Matskevitš 221943IVCM

Application of TRIZ to Find Vulnerabilities in Code, Methods of its Transmission, Storage and Execution

Master's Thesis

Supervisor: Hayretdin Bahşi
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Anton Matskevitš 221943IVCM

TRIZ-I RAKENDAMINE KOODI HAAVATAVUSTE LEIDMISEKS, SELLE EDASTAMISE, SALVESTAMISE JA TÄITMISE MEETODID

Magistritöö

Juhendaja: Hayretdin Bahşi
PhD

Tallinn 2025

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Anton Matskevitš

16.05.2025

Abstract

This thesis investigates applying Theory of Inventive Problem Solving (TRIZ) principles to enhance software vulnerability detection, focusing on engineering structured prompts for Large Language Models (LLMs) and conceptualizing systemic code lifecycle security. Motivated by limitations in standard LLM prompting, core TRIZ concepts like Function and Resource Analysis were adapted to guide LLM reasoning.

An empirical study involved iterative TRIZ-prompt development and ablation, initially with DeepSeek. Key prompts were then validated across DeepSeek, Llama, and Mistral LLMs on VulnPatchPairs and CVEFixes datasets against Chain-of-Thought (CoT) and Data Flow Analysis (DFA) baselines. Results indicated that TRIZ-based prompts led to improved F1-score and Recall over baselines when used with the DeepSeek LLM. However, for Llama and Mistral, the performance of TRIZ prompts relative to CoT and DFA baselines was more varied, with baselines often achieving comparable or superior F1-scores and Recall. Significant LLM-specific variations in overall performance, error rates (notably high for Llama and Mistral), and the persistent Recall/Precision trade-off were observed across all models, underscoring the need for model-aware prompt strategies. Ablation studies identified crucial TRIZ-derived prompt components for effective LLM guidance. Conceptually, TRIZ system thinking yielded novel defense ideas for the code lifecycle.

This research concludes that TRIZ offers a valuable structured methodology for guiding LLM-based vulnerability detection, demonstrating clear advantages with certain LLMs like DeepSeek. Nevertheless, practical application and the extent of benefit over simpler baselines are significantly conditioned by considerable LLM variability and reliability. Findings inform prompt engineering and future research into more robust, context-aware LLM security tools.

Keywords: TRIZ, Large Language Models (LLM), Vulnerability Detection, Prompt Engineering, Code Security, Cybersecurity, Static Analysis, System Thinking, Ablation Study, Software Security Lifecycle.

This thesis is written in English and is 69 pages long, including 6 chapters and 6 tables.

Annotatsioon

TRIZ-i rakendamine koodi haavatavuste leidmiseks, selle edastamise, salvestamise ja täitmise meetodid

See lõputöö uurib leiutusliku probleemilahenduse teooria (TRIZ) põhimõtete rakendamist tarkvara haavatavuste tuvastamise tõhustamiseks, keskendudes struktureeritud viipade konstrueerimisele suurte keelemudelite (LLM) jaoks ja süsteemse koodi elutsükli turvalisuse kontseptualiseerimisele. Ajendatuna standardsete LLM-viipade piirangutest, kohandati LLM-i arutluskäigu suunamiseks TRIZ-i põhikontseptsioone, nagu funktsiooni- ja ressursianalüüs.

Empiiriline uuring hõlmas iteratiivset TRIZ-viipade arendamist ja ablatsiooni, esialgu DeepSeekiga. Võtmeviibad valideeriti seejärel DeepSeeki, Llama ja Mistrali LLM-ide lõikes VulnPatchPairs ja CVEFixes andmekogumitel, võrreldes neid mõtteahela (CoT) ja andmevoo analüüsi (DFA) baasjoontega. Tulemused näitasid, et TRIZ-põhised viibad viisid DeepSeeki LLM-i kasutamisel baasjoontega võrreldes parema F1-skoori ja meenutuseni (Recall). Llama ja Mistrali puhul oli aga TRIZ-viipade jõudlus CoT ja DFA baasjoonte suhtes mitmekesisem, kusjuures baasjooned saavutasid sageli võrreldavaid või paremaid F1-skoore ja meenutust. Kõigi mudelite puhul täheldati märkimisväärseid LLM-spetsiifilisi erinevusi üldises jõudluses, veamäärades (eriti kõrged Llama ja Mistrali puhul) ning püsivat meenutuse/täpsuse kompromissi, mis rõhutab mudeliteadlike viipastrateegiate vajadust. Ablatsiooniuuringud tuvastasid tõhusaks LLM-i juhendamiseks üliolulised TRIZ-ist tuletatud viibakomponendid. Kontseptuaalselt andis TRIZ-i süsteemmõtlemine uudseid kaitseideid koodi elutsükli jaoks.

See uurimus järeldab, et TRIZ pakub väärtuslikku struktureeritud metoodikat LLM-põhise haavatavuste tuvastamise juhendamiseks, näidates selgeid eeliseid teatud LLM-idega, nagu DeepSeek. Sellest hoolimata on praktiline rakendamine ja kasu ulatus lihtsamate baasjoonte ees märkimisväärselt tingitud arvestatavast LLM-i varieeruvusest ja usaldusväärsusest. Tulemused annavad sisendit viipade konstrueerimiseks ja tulevaseks uurimistööks robustsemate, kontekstiteadlikumate LLM-i turvatööriistade valdkonnas.

Võtmesõnad: TRIZ, suured keelemudelid (LLM), haavatavuse tuvastamine, kiire projekteerimine, koodi turvalisus, küberturvalisus, staatiline analüüs, süsteemi mõtlemine, ablatsiooniuuring, tarkvara turvalisuse elutsükkel.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 69 leheküljel, 6 peatükki, 6 tabelit.

List of abbreviations and terms

TRIZ	Theory of Inventive Problem Solving
LLM	Large Language Model
SAST	Static Application Security Testing
DAST	Dynamic Application Security Testing
CoT	Chain-of-Thought
RAG	Retrieval-Augmented Generation
FP	False Positive
FN	False Negative
TP	True Positive
TN	True Negative
FPR	False Positive Rate
API	Application Programming Interface
SQL	Structured Query Language
AI/ML	Artificial Intelligence/Machine Learning
UI	User Interface
DOM	Document Object Model
MATCEM	Mechanical, Acoustic, Thermal, Chemical, Electric, Magnetic
OS	Operating System
SCADA	Supervisory Control and Data Acquisition
OCR	Optical Character Recognition
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
DFA	Data Flow Analysis
RCI	Recursive Criticism and Improvement

Table of Contents

List of Tables

1 Introduction

1.1 Problem Statement

The security of software systems remains a critical global concern, with vulnerabilities in code representing significant financial, operational, and societal risks [46, 47]. Despite advancements in software development practices and security tooling, reliably detecting and mitigating these vulnerabilities before exploitation presents persistent challenges [48]. Existing methodologies face several key limitations.

Scalability and Complexity: Modern software systems are increasingly large, complex, and interconnected [49]. Manual code review, while thorough, is infeasible at scale. Traditional automated tools like Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) often struggle to cope with this complexity, leading to incomplete analysis or an overwhelming number of findings [50, 51].

Limitations of Automated Tools: SAST tools frequently suffer from high rates of false positives and false negatives, require significant expertise to configure and interpret, and often miss vulnerabilities related to complex logic, context-dependent flaws, or novel attack patterns [52]. DAST tools require code execution, limiting their applicability during early development stages and potentially missing vulnerabilities in unexecuted code paths [50].

Inadequacy of Basic LLM Prompting: While Large Language Models (LLMs) demonstrate potential for code analysis due to their understanding of code structure and semantics, their effectiveness in vulnerability detection is highly dependent on prompting strategies. Standard prompting techniques e.g., zero-shot, basic Chain-of-Thought (CoT) often lack the necessary structure and depth for rigorous security analysis. As demonstrated in this research (Chapter 4) and supported by literature [9, 22, 24], such approaches can yield inconsistent results, suffer from very low Recall (missing most vulnerabilities), struggle with context limitations inherent in analyzing isolated code snippets, and fail to systematically explore different facets of potential weaknesses.

Narrow Focus on Static Code: Many vulnerability detection efforts concentrate primarily on analyzing static code artifacts. However, vulnerabilities are often systemic, emerging not only from code logic flaws but also from insecure interactions during the code's lifecycle – including its transmission across networks, storage on potentially compromised systems, modification during build/deployment processes, and interaction with complex execution environments (hardware, OS, libraries) [47, 53]. Current meth-

odologies often lack a systematic framework to analyze and address these broader life-cycle risks.

Therefore, a significant problem exists: current approaches, including standard applications of LLMs, are often insufficient for providing reliable, efficient, and comprehensive vulnerability detection across the full scope of modern software systems and their lifecycles. There is a clear need for more structured, systematic methodologies that can enhance the effectiveness of automated analysis tools like LLMs and provide frameworks for understanding and mitigating vulnerabilities from a broader, systemic perspective. This research posits that Theory of Inventive Problem Solving (TRIZ), with its systematic problem-solving toolkit, offers a potential avenue to address these gaps (Section 2.3).

1.2 Motivation

The imperative for robust software security continues to grow in response to the increasing frequency and sophistication of cyber attacks targeting vulnerabilities in code [46]. While LLMs like DeepSeek offer significant potential to augment traditional code review and automated analysis, their effectiveness is critically dependent on the methods used to guide their analysis. Initial exploration and existing literature [9, 22] suggest that standard prompting techniques often fall short, yielding inconsistent results or failing to uncover complex, hidden vulnerabilities due to a lack of structured reasoning guidance. This limitation motivates the search for more systematic and effective prompt engineering methodologies.

TRIZ, with its foundation in systematic analysis, contradiction identification, and structured problem-solving, presents a compelling, albeit unconventional, methodology to address these prompting challenges [54]. The core motivation for exploring TRIZ in this context is the hypothesis that its principles can provide the necessary structure and analytical depth lacking in standard prompts, thereby enhancing the LLM's ability to perform more rigorous and reliable vulnerability detection.

Therefore, this research is driven by a motivation to investigate whether the systematic application of TRIZ principles can demonstrably improve the effectiveness of LLM prompts for vulnerability detection compared to baseline methods.

1.3 Research Problems

Despite the high potential of LLMs for code analysis, their effective application to vulnerability detection faces significant hurdles, potentially leaving critical vulnerabilities undetected [9, 22]. Developing effective prompts is complex, requiring careful consideration of how to guide the LLM's reasoning process for this specific, analytical task

[55]. Furthermore, focusing solely on static code analysis overlooks vulnerabilities inherent in the broader software lifecycle, including code transmission, storage, and execution environments [53]. There is a need for methodologies that can both enhance the structured reasoning of LLMs for code analysis and provide a framework for understanding these systemic lifecycle risks, potentially giving hints on how to avoid vulnerabilities.

While LLMs excel at pattern recognition and generation based on their training data [40], ensuring reliable, systematic analysis for security requires more than statistical prediction; it necessitates structured problem decomposition and logical inference, areas where methodologies like TRIZ could potentially provide significant value [56]. This thesis explores TRIZ as a promising approach to address these challenges, both in guiding LLM analysis and in thinking systemically about vulnerability avoidance.

This research addresses the following key Research Questions (RQ): (RQ1) How can core TRIZ principles be systematically adapted and structured within general-purpose LLM prompts to guide the analysis process for code vulnerability detection? (Investigating the application of Function Analysis, Resource Analysis, structured indicators, mitigation checks, etc., as detailed in Methodology Chapter 3). (RQ2) What is the empirical effectiveness (in terms of F1-score, Recall, FPR (False Positive Rate), Precision) of these iteratively refined TRIZ-based prompts compared to a standard CoT and DFA baselines for binary vulnerability classification on real-world code snippets? (Addressing the performance comparison based on experiments in Results Chapter 4). (RQ3) What is the relative contribution of specific TRIZ-derived components (e.g., detailed framing, explicit terminology, contradiction analysis, structured indicators, justification requirement) within an optimized prompt to its overall classification performance, as determined through a systematic ablation study? (Addressing the ablation study findings in Results Chapter 4). (RQ4) What are the key challenges and limitations encountered when using TRIZ-enhanced prompts with current general-purpose LLMs for single-function static analysis, particularly regarding the Recall/Precision trade-off and context dependency? (Reflecting the discussion in Chapter 5).

1.4 Scope and Goal

This research investigates the application of TRIZ across the scope of the empirical development and evaluation of TRIZ-enhanced prompts for general-purpose LLM-based static analysis of code snippets. LLMs that are fine-tuned specifically for vulnerabilities detection are out of scope being less affordable and demanding additional preparation of models, which is more time-consuming than usage of general-purpose models [58].

Empirical Prompt Evaluation: This phase focused on engineering and testing general-purpose LLM prompts for binary vulnerability classification (vulnerable: YES/NO) of individual functions.

Methodology: An iterative development process was employed, optimizing prompts based on quantitative metrics (primarily F1-score, Recall, Precision, FPR). A systematic ablation study was conducted on the best-performing balanced prompt labelled V9 to evaluate the contribution of specific TRIZ-derived components.

LLM & Dataset: The DeepSeek (deepseek-v3 version) was used as the initial LLM. This choice is motivated by its low price and high results in benchmarks provided by LMArena project [45]. The VulnPatchPairs dataset [39], containing real-world vulnerable C programming language function snippets (from QEMU and Ffmpeg projects) and their patched versions as non-vulnerable samples (used for ground truth), served as the primary testbed. Individual vulnerable_func code snippets were analyzed without the corresponding patched_func snippets provided as direct input to the LLM during classification. This dataset is considered hard for LLM to determine if an analyzed function is vulnerable or not [38]. For cross-dataset validation CVEFixes has been chosen as the secondary dataset. It contains code snippets classification (vulnerable and non-vulnerable) and examples in several programming languages [43]. Afterwards one more series of experiments has been performed with the set of the best TRIZ-prompts and baseline prompts, but using two more LLMs: Llama (llama3-70b-8192 version) and Mistral (mistral-saba-24b version).

TRIZ Adaptation: The prompts incorporated adaptations of core TRIZ analytical tools (notably Function Analysis and Resource Analysis) and structuring principles (like Segmentation and Nested Doll implicitly through multi-step analysis and structured indicators). The utility of explicit Contradiction Analysis within the prompt was also evaluated via ablation. The initial list of specific inventive principles (Segmentation, Universality, etc.) served as inspiration rather than direct, individually tested components in the final evaluated prompts. This list has been used in this thesis as demonstration of TRIZ.

Baseline Comparison: The performance of TRIZ-based prompts was compared against a standard CoT and DFA (Data Flow Analysis) baseline prompts (Appendix 2 and Appendix 14 correspondingly) adapted from relevant literature [9], where they have demonstrated the best results over other prompts and code static analysis tools.

The specific goals achieved in this work were: (1) To analyze existing LLM prompting techniques for vulnerability detection and relevant prior applications of TRIZ in software engineering. (2) To adapt core TRIZ analytical principles (Function Analysis, Resource Analysis, etc.) into a structured methodology suitable for guiding general-purpose LLM prompts in vulnerability detection tasks. (3) To iteratively develop and empirically optimize TRIZ-based prompts for binary classification of code function snippets, aiming to maximize the F1-Score by balancing Recall and Precision. (4) To conduct experiments using the several general-purpose LLMs (DeepSeek, Llama, Mistral) and datasets (VulnPatchPairs and CVEFixes) to evaluate the performance of the developed prompts. (5) To quantitatively compare the effectiveness (F1, Precision, Recall,

FPR) of the optimized TRIZ prompts against a standard CoT and DFA baseline prompts. (6) To perform an ablation study on a selected TRIZ prompt (V9) to determine the empirical contribution of its key structural and TRIZ-derived components. (7) To formulate practical recommendations for engineering structured LLM prompts for security analysis, based on the iterative refinement and ablation study findings.

Additionally this work demonstrates the way TRIZ tools can be used to propose solutions for avoiding the appearance of vulnerabilities in code (Section 5.3.7).

1.5 Novelty

This research presents several novel contributions at the intersection of TRIZ, LLMs, and cybersecurity.

Systematic Application and Empirical Evaluation of TRIZ for LLM Prompt Engineering in Security: While TRIZ has been explored in software engineering [3, 4, 5] and LLMs are increasingly used for security tasks [9, 10, 21, 22], this study provides one of the first systematic adaptations and, crucially, empirical evaluations of TRIZ principles specifically for structuring LLM prompts aimed at code vulnerability detection. Unlike prior conceptual work or applications in different domains [6, 7], this research involved iterative, data-driven prompt refinement and performance measurement using metrics like F1-score, Recall, and Precision.

Demonstrated Performance Improvement over Baseline: The study empirically demonstrates that TRIZ-enhanced prompts, leveraging structured analysis based on Function Analysis and Resource Analysis, significantly outperform standard CoT and DFA baseline prompts adapted from existing literature [9] in terms of F1-score and Recall for vulnerability detection on the VulnPatchPairs [39] and CVEFixes [43] datasets. This provides concrete evidence for the value of the structured TRIZ approach over simpler prompting methods for this task.

Ablation Study Insights for TRIZ Prompt Components: Through a systematic ablation study, this research identifies the specific contributions of different TRIZ-derived components within the LLM prompt. Key findings revealed the critical importance of detailed Function/Resource analysis framing and the justification generation step, while suggesting that explicit Contradiction analysis might be less essential within this particular structure. This provides novel, data-driven insights into which aspects of TRIZ are most impactful when applied to LLM prompt engineering for security.

Conceptual Application of TRIZ System Thinking to Code Lifecycle Security: Additionally this work uniquely extends the application of TRIZ beyond static code analysis to the entire code lifecycle, including transmission, storage, and execution. By applying TRIZ system thinking tools (like System Operator, Sub/Super-system analysis, Contra-

diction identification), it presents a novel conceptual framework for identifying systemic vulnerabilities.

Generation of Novel TRIZ-Inspired Defense Concepts: The systemic analysis led to the generation of novel conceptual defense mechanisms inspired by TRIZ principles [57], such as the "Analog Filter" (using Principle 24: Intermediary, Principle 28: Mechanics Substitution) and Layered Security Architectures (using Principle 1: Segmentation, Principle 7: Nested Doll). Linking these concepts to existing high-security technologies (e.g., Data Diodes) validates the underlying TRIZ principles while suggesting innovative future directions, such as intra-component filtering.

Interdisciplinary Methodology: The research successfully integrates methodologies and concepts from Systematic Innovation Theory (TRIZ), Artificial Intelligence (LLM Prompt Engineering), and cybersecurity (Vulnerability Analysis), offering a novel interdisciplinary perspective and practical insights that bridge these fields.

In summary, the novelty lies not just in proposing TRIZ for general-purpose LLM security prompting, but in the rigorous empirical validation, the identification of impactful prompt components through ablation, and the extension of TRIZ system thinking to generate new perspectives on vulnerabilities and defenses across the entire code lifecycle.

2 Literature Review

2.1 Existing Tools

The domain of code vulnerability detection has evolved significantly to address the increasing complexity of software systems. Traditionally, Static Application Security Testing (SAST) tools have been instrumental. These tools analyze source code, bytecode, or binary code without executing it, typically matching patterns against known databases of insecure coding practices or performing abstract interpretation. SAST tools have been refined over decades, are often integrated into development pipelines for early feedback, but are also known for potential limitations including high rates of false positives/negatives and difficulties with understanding context or complex logic flows [9]. Dynamic Application Security Testing (DAST) tools, which test running applications, offer a complementary approach but cannot be applied before execution and may miss vulnerabilities in unexercised code paths.

More recently, LLMs have emerged as a promising alternative or augmentation for vulnerability detection. Leveraging their extensive training on code and natural language, LLMs possess capabilities for understanding code semantics, generating code, and identifying patterns that may go beyond traditional rule-based SAST [9, 10, 11]. Studies suggest LLMs can identify certain types of vulnerabilities, sometimes achieving performance comparable to or exceeding traditional tools in specific benchmarks [9]. However, their effectiveness is highly sensitive to how they are prompted and guided, and standard prompting techniques often lack the structured analytical depth required for consistent and reliable security review [21, 22, 26, 24].

The application of TRIZ in software development, while not mainstream, has been explored previously, primarily focusing on software architecture [3], adapting inventive principles for software engineering nuances [4], and addressing specific problems like concurrency [5, 6]. Some recent work explores using LLMs to automate aspects of TRIZ for different domains [6, 7]. However, the specific application of TRIZ principles to systematically engineer prompts for LLMs with the explicit goal of improving vulnerability detection accuracy and reliability appears underexplored, representing the core focus of this thesis.

Given the demonstrated potential and the known prompting challenges of LLMs in security tasks, this literature review will focus primarily on LLM-based vulnerability detection approaches and advanced prompting strategies (discussed in Section 2.2 on-

wards). While acknowledging the foundational role of traditional SAST/DAST, the subsequent sections delve into the state-of-the-art in leveraging LLMs, thereby setting the context for the novel TRIZ-inspired prompt engineering methodology developed and evaluated in this research.

2.2 LLM-Based Vulnerability Detection

The emergence of LLMs has opened new frontiers in automated code analysis, including vulnerability detection. Their ability to process and understand code semantics offers potential advantages over purely pattern-based traditional tools. However, harnessing this potential effectively requires careful consideration of prompting strategies, evaluation methodologies, and inherent challenges.

2.2.1 Prompt Design for Vulnerabilities Detection

The quality and structure of prompts are paramount to the effectiveness of LLMs in identifying vulnerabilities. Research consistently demonstrates that structured reasoning frameworks significantly enhance performance compared to simple zero-shot queries.

Chain-of-Thought: Prompts encouraging step-by-step reasoning, mimicking human analytical processes, have shown notable improvements. Liu et al. (2024) found that combining CoT with contextual information like data flow diagrams enabled GPT-4 to detect specific vulnerabilities like SQL injections and buffer overflows with better precision [21, 26].

Multi-Step Reasoning: Advanced prompting involving multiple analytical steps or refinement stages has also proven effective. Zhou et al. (2024) reported that such methods could reduce false positive rates by 18-25% compared to basic zero-shot prompts [22].

Decomposition: Techniques that decompose the analysis task, perhaps by focusing on specific code parts or vulnerability types sequentially, align with effective human reasoning and tend to yield better results [24, 25].

These findings underscore the principle that guiding the LLM through a structured, decomposed analytical process improves accuracy and reliability. This motivates the exploration undertaken in this thesis: investigating whether TRIZ, as a formal methodology for systematic problem decomposition and analysis, can provide a novel and effective framework for structuring such prompts beyond standard CoT.

2.2.2 State-of-the-Art Prompting Frameworks

Several advanced frameworks aim to enhance LLM performance for security tasks, often by incorporating external information or specific reasoning patterns:

Retrieval-Augmented Generation (RAG): RAG frameworks augment prompts by retrieving relevant information from external knowledge sources. For vulnerability detection, this often involves fetching known vulnerability patterns, descriptions, or code examples from databases such as CVE details, MITRE ATT&CK, or specialized code vulnerability databases [23, 26]. CyberSecEval, for instance, uses RAG to link code to ATT&CK techniques [26], while Vul-RAG retrieves vulnerable code snippets [27, 28]. While effective, RAG's performance depends heavily on the quality and relevance of the retrieved information and may struggle with novel or obfuscated vulnerabilities not present in the knowledge base [27].

Contrastive Chain-of-Thought (C-Think): This approach integrates examples of both secure and insecure code snippets into the prompt, often synthetically generated, to help the LLM learn contrasting patterns and refine its reasoning. The FG-CVD model demonstrated an 11% F1-score improvement on the SEVEN dataset using this technique [22]. This relies on the availability and quality of relevant contrasting examples.

Multi-Task Learning Prompts: These prompts combine related tasks, such as instructing the LLM to both detect *and* repair vulnerabilities (e.g., "Detect and fix SQL injections..."). Frameworks like LLMPATCH utilize adaptive prompting for patch generation, reportedly reducing vulnerability reintroduction rates [22, 29].

The TRIZ-based prompting approach explored in this thesis differs from these frameworks by focusing on structuring the LLM's *internal analytical process* using first principles (Function, Resource, Contradiction analysis) derived from TRIZ theory itself, rather than primarily relying on external data retrieval (RAG), specific contrasting examples (C-Think), or combined tasking (Multi-Task).

2.2.3 Benchmarks, Metrics, and Evaluation

Robust evaluation is crucial for comparing the effectiveness of different vulnerability detection methods. Recent efforts include the following.

Q&A Benchmarks: Datasets like CyberMetric provide multiple-choice questions to assess an LLM's cybersecurity knowledge across various domains [23, 30].

Code-Based Benchmarks: Frameworks like SVEN focus on real-world vulnerabilities, demonstrating performance gains for CoT over zero-shot prompts (e.g., 72% vs 48% accuracy [24, 26, 31]). Datasets like VulnPatchPairs [39], used in this thesis, provide real-world vulnerable/patched function pairs for evaluating classification accuracy.

Synthetic Vulnerability Injection: Techniques like LAVA [28, 32] automatically inject vulnerabilities into codebases to test detection scalability. Notably, prompts employing hierarchical analysis (a technique related to TRIZ principles like Nested Doll) reportedly detected 92% of LAVA-injected bugs [32], suggesting the potential benefit of structured analysis approaches.

A persistent challenge is the lack of universal benchmarks covering a wide range of programming languages and vulnerability types, often limiting comparisons across studies [25]. This study utilizes VulnPatchPairs [39] and CVEFixes [43] for its focus on real-world code examples with clear ground truth.

2.2.4 Challenges in Prompting LLMs

Despite their potential, several challenges limit LLM effectiveness for vulnerability detection via prompting.

Context Length Limitations: LLMs have finite input token limits. Long code snippets may be truncated, potentially removing crucial context or the vulnerability itself, especially for flaws spanning large code sections [24]. This necessitates analyzing smaller code units (like functions), which inherently limits inter-procedural analysis capabilities, a limitation observed in this thesis's results (Chapters 4, 5).

Rare Vulnerabilities: LLMs trained on vast code corpora may still struggle to identify rare or novel vulnerability types e.g. uncommon CWEs (Common Weakness Enumeration) due to their infrequent appearance in the training data [22].

Adversarial Prompts/Input: Maliciously crafted prompts or inputs could potentially mislead an LLM or cause it to generate insecure code suggestions [28].

Consistency and Reliability: LLM outputs can be stochastic, and ensuring consistent, reliable detection requires careful prompt design and potentially multiple runs or low temperature settings, as employed in this study's methodology.

Addressing these challenges, particularly the context limitation and the need for reliable reasoning, motivates the exploration of structured prompting methodologies like the TRIZ-based approach investigated herein.

2.3 TRIZ in Software Development and Cybersecurity

While TRIZ (Russian: Теория Решения Изобретательских Задач, literally “Theory of Inventive Problem Solving”) originated in mechanical and physical engineering domains, its principles and systematic approach have been explored for application in software development over the years, demonstrating its potential value beyond the traditional hardware focus.

Software Architecture and Design: Researchers have investigated TRIZ as a tool for systematic software architecture design. Its methods for handling conflicting quality requirements (e.g., performance vs. maintainability – inherent Contradictions) can help architects make structured design decisions and build more robust systems [3].

Adapting Inventive Principles: Recognizing that many of the original 40 inventive principles relate to physical phenomena, studies have focused on creating analogies applic-

able to the software domain. For example, Principle 29 (Hydraulic/Pneumatic Construction) has been mapped to concepts like dynamically allocated data structures, demonstrating that the underlying inventive logic can often be translated [4]. A comprehensive study also confirmed the general applicability of TRIZ to software development with appropriate adaptations [5].

Solving Specific Software Problems: TRIZ tools have also been applied to solve specific, complex software engineering challenges, such as concurrency control problems (e.g., the "Roller Coaster Problem"), showcasing its utility as a problem-solving framework within the software domain [6].

However, the literature review conducted for this thesis revealed that while TRIZ has found application in general software engineering and architecture, its specific use within the cybersecurity domain, particularly for vulnerability analysis or detection, remains significantly less explored. Furthermore, the application of TRIZ principles specifically to the emerging field of LLM prompt engineering for any task, let alone for the complex goal of code security review, appears to be a novel area.

Therefore, while prior work establishes the feasibility of adapting TRIZ for software-related problem solving and system design, a clear gap exists in leveraging its systematic methodology to address the specific challenges of vulnerability detection, especially through the lens of guiding advanced tools like LLMs. This research aims to bridge that gap by systematically adapting and evaluating TRIZ principles for LLM prompt engineering focused on security (as detailed in Chapter 3) and by applying TRIZ system thinking to the broader code security lifecycle.

2.4 Gaps

The literature review reveals significant potential for leveraging LLMs in vulnerability detection, alongside established applications of TRIZ in software engineering. However, several critical gaps exist, which this thesis aims to address.

Lack of Systematic TRIZ Application for LLM Security Prompting: While prior research demonstrates the successful application of TRIZ principles to software architecture [3], general software engineering problem-solving [4, 5], and specific issues like concurrency [6], there is a notable absence of studies systematically adapting and evaluating TRIZ specifically for engineering LLM prompts aimed at code vulnerability detection. Existing advanced prompting techniques like CoT [21, 26] and RAG [23, 26, 27] provide structure or external knowledge, but they typically lack the inherent systematic problem decomposition, contradiction analysis, and function/resource modelling framework offered by TRIZ. This research directly addresses this gap by proposing and empirically evaluating a TRIZ-based prompting methodology.

Need for Structured Approaches Beyond Basic CoT/RAG: As highlighted in Section 2.2, while CoT and RAG improve upon zero-shot prompting, challenges remain regarding consistency, handling complex logic, and context limitations [24, 25, 28]. The literature indicates a need for more robust, structured methodologies to guide LLM reasoning for security tasks. TRIZ, with its emphasis on breaking down problems and analyzing system functions and interactions, offers a potential framework to provide this deeper structure, moving beyond linear step-by-step reasoning or simple knowledge retrieval.

Limited Focus on Systemic Lifecycle Vulnerabilities: The majority of automated vulnerability detection research, including LLM-based approaches reviewed, concentrates primarily on analyzing static code artifacts. While essential, this overlooks vulnerabilities that can arise during other phases of the code lifecycle, such as insecure transmission, storage, or execution environment interactions. A systematic methodology, like that offered by TRIZ system thinking, for analyzing these broader lifecycle risks and identifying potential mitigation strategies appears largely unexplored in the cybersecurity literature reviewed.

Therefore, this thesis addresses these gaps by: (1) developing and empirically evaluating a novel, structured prompting methodology for LLMs based on adapted TRIZ principles, comparing its performance against a standard CoT and DFA baselines; and (2) conceptually applying TRIZ system thinking to analyze vulnerabilities across the entire code lifecycle, proposing potential countermeasures beyond static code analysis.

3 Methodology

This chapter details the research methodology employed to investigate and evaluate the application of TRIZ principles for engineering LLM prompts aimed at detecting vulnerabilities in source code. The core objective is to assess whether TRIZ-enhanced prompts offer performance improvements over baseline prompting techniques. The methodology integrates theoretical TRIZ principle adaptation, an iterative prompt engineering process informed by empirical results, experimental validation using a real-world datasets, and statistical evaluation to ensure the robustness of the findings regarding the system "software code", its potential vulnerabilities, and methods for analysis.

3.1 Research Design

This study employs a mixed-methods approach, combining qualitative principle adaptation with quantitative experimental validation. The research proceeds through the following distinct phases:

Literature Review: A comprehensive review was conducted covering: (1) Existing approaches for vulnerability detection (static analysis, dynamic analysis, AI/ML methods). (2) LLM capabilities and limitations in code understanding and analysis. (3) Standard LLM prompting strategies (zero-shot, few-shot, CoT, RAG, DFA). (4) Foundational TRIZ principles and documented applications in non-traditional domains, including software engineering and potentially cybersecurity [1, 2].

TRIZ Principle Adaptation & Initial Prompt Design: Relevant TRIZ principles were selected and adapted from their traditional engineering context to the domain of LLM prompt engineering for code analysis. An initial set of conceptual prompts and prompt structures based on these adapted principles were designed (examples discussed in Section 3.2).

Iterative Prompt Refinement: An empirical, iterative process was undertaken to develop and optimize a TRIZ-based prompt for binary vulnerability classification (vulnerable: YES / vulnerable: NO) of the function-based datasets. Starting with an initial complex TRIZ prompt, multiple versions were created and tested against the initial dataset, with adjustments made based on quantitative performance metrics (especially F1-score, Recall, Precision, FPR) to progressively improve the balance between detecting vulnerabilities and minimizing false alarms. This process involved approximately 5 major iterations (detailed in Section 3.2.3).

Ablation Study: Once a reasonably well-performing prompt (V9) was identified through iteration, an ablation study was designed and executed to systematically evaluate the contribution of specific TRIZ-derived components within that prompt structure (detailed in Section 3.2.4).

Experimental Validation & Comparison: The performance of the most promising TRIZ-based prompts (identified through iteration and ablation) was compared against baseline prompts (specifically, the best CoT and DFA prompts adapted from [9]) using the chosen datasets and metrics.

Results Analysis: Quantitative metrics were statistically analyzed (e.g., comparing F1-scores). Qualitative analysis of LLM outputs (justifications) helped understand the strengths and weaknesses of the different prompting approaches.

Recommendations and TRIZ System Analysis: Based on the findings, recommendations for engineering TRIZ-based prompts for vulnerability detection are formulated. Additionally, TRIZ system thinking concepts (like function analysis, contradictions) are applied conceptually to the broader system of "vulnerable code" including its transmission, storage, and execution, to propose avenues for preventing vulnerability exploitation beyond just code-level detection.

3.2 TRIZ Principle Selection and Prompt Engineering

TRIZ offers a systematic methodology for problem-solving and system improvement, originally developed by Genrich Altshuller based on patent analysis [1]. It posits that inventive problems often involve resolving underlying contradictions and that universal inventive principles can be applied across different domains [2]. Key concepts include Function Analysis, Resource Analysis, Ideality, Contradictions, the 40 Inventive Principles, and 76 Standard Solutions.

The structured nature of TRIZ, particularly its methods for decomposing problems, analyzing functions and resources, and resolving contradictions, suggested its potential suitability for engineering complex LLM prompts designed for the nuanced task of code vulnerability detection.

Due to its nature and variety of methods meant to decompose a task into smaller pieces TRIZ may be suitable tool for composing prompts for LLMs to achieve better results.

Among all 40 principles only a small part could be used in non-physical cyber space, because they are meant to be used for different materials or phenomena. Examples of such non-applicable principles are Inert Atmosphere, Composite Materials, Thermal Expansion, Color Changes, Mechanical Vibration, Curvature, Mechanics Substitution. But the following chosen principles are applicable and meant to help compose LLM prompts as the most promising ones.

Generally TRIZ usage can be described as an algorithm of actions that can be applied to an unsolved problem. Such “algorithm of problem solving” in scope of this work may look like this: (1) Problem fixation. (2) Articulating goals (including storyboarding usage and testing solutions for value). (3) Pursuit the Ideal Final Result of LLM’s output to avoid work done by humans. (4) Options for solutions to get ahead of hackers actions.

3.2.1 TRIZ Principles Selection for Prompting

While TRIZ originates from the physical sciences, several of its core principles and analytical approaches were deemed adaptable to the abstract domain of software analysis and LLM interaction. The selection focused on principles facilitating structured analysis, pattern identification, and iterative refinement. Principles strongly tied to physical phenomena (e.g., Thermal Expansion, Inert Atmosphere, Composite Materials) were excluded. The following principles were initially identified as most promising for guiding prompt design.

Segmentation (Principle 1): Breaking down the complex task of vulnerability analysis into smaller, manageable sub-tasks or analytical steps within the prompt.

Universality (Principle 6): Designing prompts applicable across different languages/ contexts by focusing on universal vulnerability patterns rather than specific syntax (though syntax awareness is still needed).

Asymmetry (Principle 4): Using varied phrasing or approaching the analysis from different angles within prompts to potentially uncover non-standard code patterns.

Preliminary Action (Principle 10): Structuring prompts to first identify necessary context (e.g., inputs, data types) before analyzing for vulnerabilities, or priming the LLM with examples.

Nested Doll (Principle 7): Designing prompts that guide analysis from high-level structures/concerns down to specific details (e.g., analyze function interaction, then specific arithmetic operations).

Preliminary Anti-Action (Principle 9): Instructing the LLM to look not only for flaws but also for signs of intentional obfuscation or missing preventative measures (like expected validation).

Dynamicity (Principle 15): (More applicable to multi-turn interaction) Adapting subsequent prompts based on intermediate LLM findings.

Feedback (Principle 23): (More applicable to multi-turn interaction) Explicitly incorporating previous LLM outputs or findings into new prompts for refinement.

(Implicit Principles): Core TRIZ concepts like Function Analysis (Identify useful/harmful/insufficient functions), Resource Analysis (Identify inputs, data, state, memory and

their control), Contradiction Analysis (Identify conflicting requirements like speed vs. safety checks), and Ideality (Compare code to an ideal secure function) were fundamental to structuring the analytical steps within the prompts.

3.2.2 Principle Adaptation

The selected principles were adapted into prompt design strategies. Table 1 provides conceptual examples of how these principles could be mapped, illustrating the initial thinking before iterative refinement. The actual prompts evolved significantly from these initial concepts based on experimental results.

The Table 1 depicts how each selected TRIZ principle can be mapped to LLM prompt example and what prompt design strategy is used for each principle.

Table 1: Conceptual Mapping of TRIZ Principles to Prompt Strategies (Illustrative)

TRIZ principle	**Prompt Design Strategy**	**Example of Prompt Component**
Segmentation	Break down vulnerability detection into smaller subtasks like input handling, then memory, then exceptions.	First, analyze input validation. Second, check memory allocation. Third, verify exception handling practices.
Universality	Create language-agnostic prompts that adapt to syntax (e.g., sanitization in C vs in Python).	Identify insecure input handling in [LANGUAGE]. For [LANGUAGE], common risks include [EXAMPLE].
Asymmetry	Use different wording to detect vulnerabilities in code with non-standard or absent patterns.	Use both of these for the same piece of code: 1) Find insecure data handling. 2) Identify unsafe user input processing.
Preliminary Action	Pre-train LLM on vulnerability patterns before analysis.	Review the following 5 SQLi examples. Now analyze the code for similar vulnerabilities.
Nested Doll	Layer prompts to detect high-level risks first, then specific vulnerabilities.	Identify insecure functions. For each, check for buffer overflows or integer overflows.
Preliminary Anti-Action	Ask LLM to detect obfuscated or masked vulnerabilities.	Analyze for vulnerabilities AND code patterns that might hide them (e.g. indirect calls, obfuscated loops).
Dynamicity	Dynamic prompt adjustment based on prior results received from LLM.	Based on your prior analysis of input handling, now check if output encoding is missing.
Feedback	Use LLM's previous outputs to improve the following queries.	Earlier, you identified [X]. Re-analyze the code with a focus on [X-related risks].

3.2.3 Iterative Prompt Development and Refinement

Engineering an effective prompt for nuanced binary vulnerability classification proved challenging, requiring an iterative approach guided by empirical results from testing on the VulnPatchPairs dataset [39]. The core challenge revolved around resolving the inherent Contradiction between maximizing Recall (detecting true vulnerabilities) and maximizing Precision (minimizing false positives). The process involved several major iterations, adjusting key aspects of the prompt's logic and analytical focus based on performance metrics (F1-score, Recall, Precision, FPR).

The key differences between the major prompt versions tested (as summarized in Results Chapter 4, Table 2) lay primarily in how strictly vulnerability indicators were defined, how mitigation effectiveness was assessed, and the threshold logic used for the final binary classification.

Initial Prompt Design (leading to TRIZ prompt and labelled as V8): Based on adapted TRIZ principles (Function Analysis, Resource Analysis, Contradiction Analysis, specific vulnerability indicators), initial complex prompts were designed aiming for high accuracy. These early versions (represented by the performance characteristics of V8) employed strict criteria, requiring strong, clear evidence of a vulnerability and likely focusing on the absence of standard mitigations before classifying as `vulnerable: YES`. This resulted in high Precision but very low Recall, proving too conservative. Later on prompts labelled V9, V10, and V11 have been developed.

First Iteration Cycle (Prompt V8): The initial prompt V8 (refer to Appendix 3) was tested, revealing very low Recall (0.0920), average Precision (0.5349), and F1-Score (0.1570), indicating it was too conservative. The focus shifted to increasing Recall.

Shift Towards Balance (V9 - Ablation Baseline): To improve Recall, the logic was adjusted. Prompt V9 (refer to Appendix 4) maintained the detailed TRIZ framing (Function/Resource/Contradiction analysis) and structured indicators. However, the decision logic was significantly altered: it classified as `vulnerable: YES` if a high-confidence indicator was found unless Strong Internal Mitigation was also identified. The focus shifted from needing strong proof of vulnerability to needing strong proof of mitigation to classify as `NO`. This version achieved a more moderate balance (F1≈0.45) and was selected for the ablation study.

Attempted Refinement (V10): Seeking to improve V9's balance further, prompt V10 (refer to Appendix 5) experimented with stricter definitions for the High-Confidence Indicators (requiring clearer evidence for Indicators A-D) and potentially a more nuanced assessment of mitigation effectiveness. The goal was to reduce False Positives from V9. However, this combination proved counterproductive, significantly reducing Recall (to ≈0.16) and F1-score (to ≈0.25), indicating the stricter indicator definitions were too limiting.

Recall Maximization Attempt (V11): Reacting to the low recall of conservative approaches, Prompt V11 (refer to Appendix 6) aggressively targeted Recall. It likely relaxed the definition of vulnerability indicators (from "High-Confidence" to "Plausible") while simultaneously making the criteria for classifying as `vulnerable: NO` much stricter (requiring unambiguous and complete mitigation). This shifted the decision bias heavily towards YES, resulting in the highest Recall (≈0.82) and F1 (≈0.63) but also an extremely high, impractical FPR (≈0.81).

This iterative process, primarily manipulating the sensitivity of vulnerability indicators and the stringency of mitigation assessment within the TRIZ-structured analytical framework, highlights the difficulty in optimizing LLM prompts for this task. The TRIZ framework provided a consistent structure for analysis (Function, Resource, etc.), but the final classification performance was highly sensitive to the specific decision logic and thresholds applied based on that analysis, requiring empirical tuning and revealing the persistent Recall/Precision trade-off. The ablation study (Section 3.2.4) further dissected the impact of specific structural components within the chosen baseline (V9).

3.2.4 Ablation Study on Baseline Prompt V9

To better understand the contribution of specific TRIZ-derived components within the best-balanced prompt identified (V9, F1=0.4539), an ablation study was conducted.

Methodology: Six variations of the V9 prompt were created, each removing or simplifying one key component: (1) Explicit TRIZ Terminology (Variation 1 labelled Var 1, refer to Appendix 7), (2) Contradiction Analysis step (Variation 2 labelled Var 2, refer to Appendix 8), (3) Detailed TRIZ Framing (Variation 3 labelled Var 3, refer to Appendix 9), (4) Structured High-Confidence Indicators (Variation 4 labelled Var 4, refer to Appendix 10), (5) Mitigation Confidence levels (simplified to boolean) (Variation 5 labelled Var 5, refer to Appendix 11), (6) Justification step (Variation 6 labelled Var 6, refer to Appendix 12). Each variation was tested on the dataset.

Key Findings: Important Components: Detailed TRIZ framing (Function/Resource Analysis), requiring Justification, and using Explicit TRIZ Terminology were found to be beneficial, as their removal significantly hurt performance (F1/Recall). The structured High-Confidence Indicators, while potentially limiting Recall compared to a general approach (Variation 4), were crucial for maintaining acceptable Precision/FPR. Removable/Modifiable Components: Explicit Contradiction Analysis (Step 2c) appeared redundant or slightly harmful; its removal (Variation 2) slightly improved F1 and significantly improved Precision/FPR. Simplifying the Mitigation Assessment to a boolean check (Variation 5) also improved F1 over the baseline, suggesting the original confidence levels were suboptimal.

Conclusion: The ablation study indicated that while the core TRIZ analytical steps (Function/Resource analysis) and structured guidance (Indicators, Justification) are

valuable, specific elements like explicit Contradiction analysis could potentially be removed. It also highlighted the high sensitivity of performance to the structure of the vulnerability indicators and the mitigation assessment logic. Variation 2 (Baseline without Contradiction Analysis) emerged as a potentially improved prompt offering better Precision/FPR than the baseline with similar Recall and F1.

3.2.5 Mapping Ablation Variations to Prompt Components

To clarify the specific modifications made for each ablation experiment described in Section 3.2.4, this section maps each variation to the corresponding parts of the Baseline Prompt (V9) text.

Variation 1: Ablate Explicit TRIZ Terminology

Throughout the Baseline Prompt text (including ROLE, TASK, ANALYTICAL PROCESS steps 2, 3, 6, and FINAL INSTRUCTION), TRIZ-specific terms ("TRIZ", "Function Analysis", "Resource Analysis", "Contradiction Analysis", specific Principle names like "Principle 11", "Principle 9/10") were replaced with their generic security analysis equivalents (e.g., "systematic analysis", "functional security review", "data/resource handling analysis", "conflicting requirements check", "preparing for potential issues", "performing necessary actions beforehand"). The core structure and analytical questions remained the same. Refer to Appendix 7.

Variation 2: Ablate Contradiction Analysis

Subsection (c) Contradiction Analysis within Step 2 (TRIZ Problem Framing & Weakness Identification) of the ANALYTICAL PROCESS was entirely removed. Subsequent steps referencing the full Step 2 analysis implicitly excluded contradiction analysis. Refer to Appendix 8.

Variation 3: Simplify TRIZ Framing

The entire Step 2 (TRIZ Problem Framing & Weakness Identification), including subsections (a), (b), and (c), was replaced with a single, higher-level instruction focused on general weakness identification (as described in the ablation plan). Refer to Appendix 9.

Variation 4: Ablate Indicator Structure

The specific definitions and labels for Indicator A, B, C, and D within Step 3 (Identify High-Confidence Vulnerability Indicators) were removed. The step was replaced with a general instruction to identify high-confidence mechanisms based on Step 2.

Consequently, references to specific indicators A-D in Step 5 (Binary Classification Decision) and Step 6 (Justification) were also generalized. Refer to Appendix 10.

Variation 5: Simplify Mitigation Assessment

In Step 4 (Contextual Evaluation), the instruction "Assess Mitigation Confidence: Strong Internal Mitigation..., Weak/No Internal Mitigation..." was replaced with a simpler boolean assessment, such as "Assess Mitigation Presence: Mitigation Present..., Mitigation Absent...".

The decision logic in Step 5 (Binary Classification Decision) was adjusted to use the "Mitigation Present" / "Mitigation Absent" outcome instead of "Strong" / "Weak/No".

The required explanation in Step 6 (Justification) was adjusted to refer to the presence or absence of mitigation. Refer to Appendix 11.

Variation 6: Ablate Justification Step

The entire Step 6 (Justification) of the ANALYTICAL PROCESS was removed.

The OUTPUT FORMAT section was modified to request only the binary classification, removing the requirement for justification text.

The FINAL INSTRUCTION was modified to remove the sentence requiring a detailed justification.

This explicit mapping ensures clarity regarding how each conceptual ablation was implemented by modifying specific sections of the baseline V9 prompt text during the experimental phase. Refer to Appendix 12.

3.3 Experimental Design

3.3.1 Comparison Targets

To show the novelty of this work and results' effectiveness a comparison will be performed between results received from usage of baseline vulnerability detection prompts and TRIZ-prompts engineered during Prompt Development Process:

Baseline: this represents standard LLM prompting techniques without TRIZ integration. For this study, CoT and DFA prompts, adapted from examples in the study "Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study" [9], were used as the primary baselines for comparison.

TRIZ-prompt: this represents the prompts iteratively engineered during this research using adapted TRIZ principles (detailed in Section 3.2). The goal was to demonstrate improvement over CoT and DFA baselines.

3.3.2 Metrics

To perform the comparison and guide iterative refinement, a standard set of binary classification metrics were calculated based on ground-truth labels from the dataset.

Quantitative. True Positives (TP): Vulnerable code correctly classified as YES. True Negatives (TN): Non-vulnerable code correctly classified as NO. False Positives (FP): Non-vulnerable code incorrectly classified as YES. False Negatives (FN): Vulnerable code incorrectly classified as NO. Accuracy: the ratio of correctly predicted observations to the total observations. It answers the question how often is the classifier correct? Precision: the proportion of positive identifications that were actually correct. It tells you of all the instances the model predicted as "vulnerable", how many were truly vulnerable? F1-score: the harmonic mean of Precision and Recall. It balances the trade-off between them and is particularly useful when the class distribution is imbalanced or when both false positives and false negatives are important. FPR: a key metric used in classification analysis, it quantifies the proportion of actual negative instances that are incorrectly identified as positive. In simpler terms, it measures how often a model or test raises a false alarm [59]. These metrics are calculated using the following formulas:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision_{\text{Vulnerable}} = \frac{TP}{TP + FP}$$

$$Recall_{\text{Vulnerable}} = \frac{TP}{TP + FN}$$

$$F1_{\text{Vulnerable}} = \frac{2 \cdot Precision_{\text{Vulnerable}} \cdot Recall_{\text{Vulnerable}}}{Precision_{\text{Vulnerable}} + Recall_{\text{Vulnerable}}}$$

$$FPR = \frac{FP}{FP + TN}$$

Qualitative. Interpretability: Review of the LLM's generated justifications (where applicable) for clarity, relevance, and correctness of reasoning, particularly how well it linked findings to TRIZ concepts and mitigation assessment.

3.3.3 Datasets

There are several datasets used for vulnerabilities detection in the field.

OWASP Benchmark Project

Description: The OWASP Benchmark project is test set for static code analysis tools performance evaluation. It contains both vulnerable and secure code.

Advantages: Well-documented and widely used industry-wise standard for security tools evaluation.

Juliet Test Suite

Description: Juliet Test Suite by NIST (National Institute of Standards and Technology) contains thousands of code examples written in C/C++ and Java with different types of vulnerabilities.

Advantages: This is one of the most complete test sets for security researchers suitable for comparative analysis.

SATE V Ockham Sound Analysis Test Suite

Description: This dataset is also provided by NIST and includes code examples with different vulnerabilities for evaluation of static analysis effectiveness.

Advantages: Contains detailed test cases suitable for analysis of specific kinds of vulnerabilities.

VulnDB (Vulnerable Code Snippets)

Description: VulnDB — this is a set of vulnerable code snippets taken from real software projects. It contains vulnerable and fixed versions of the code.

Advantages: Based on real-world examples, which makes it especially useful for tools evaluation in real-world scenarios.

CodeQL Database

Description: CodeQL — this is GitHub tools for code analysis. It can be used for vulnerabilities search in open-source projects available on GitHub.

Advantages: Gives an opportunity to create and analyze own code repositories, which can be helpful for creation of custom datasets.

SVF (Source Vulnerability Frequency) Dataset

Description: This dataset includes open projects from GitHub which have been analyzed for vulnerabilities using various tools.

Advantages: Suitable for analysis in a context of open-source projects.

LAVA Dataset

Description: LAVA (Large-scale Automated Vulnerability Addition) — automatically generated dataset with vulnerabilities which can be used for testing and comparing security analysis tools.

Advantages: Suitable for big-scale experiments thanks to big amount of automatically added vulnerabilities.

REEF Dataset

Description: REEF (REal-world vulnErabilities and Fixes) is an automated framework for collecting code from open-source repositories [34].

Advantages: It is large multi-language collection containing 30,987 patches, which are validated and explained by humans [35].

VulnPatchPairs Dataset

Description: VulnPatchPairs is a dataset used to show that LLMs aren't able to differentiate vulnerable code from its patched version effectively enough [39]. Thus it is considered hard.

Advantages: Contains vulnerable code snippets and their corresponding patches derived from actual open-source projects (QEMU, FFmpeg), avoiding synthetic examples, offers a substantial number of samples, the presence of vulnerable/patched pairs provides clear ground truth for vulnerability classification.

CVEFixes Dataset

Description: The CVEFixes dataset is a comprehensive collection of vulnerabilities automatically gathered and curated from Common Vulnerabilities and Exposures (CVE) records found in the public U.S. National Vulnerability Database (NVD). It links CVE information to the specific code changes (commits) that fixed the vulnerability in open-source projects. The dataset provides detailed information at various interconnected levels, including repository, CVE, commit, file, and method/function, offering the source code both before and after the fix [43].

Advantages: It is based directly on publicly reported CVEs and their actual fixes in open-source software. Also, the dataset provides multi-level, interlinked information (repository, CVE, commit, file, method) allowing for in-depth analysis. Contains both the vulnerable and the patched versions of the code, essential for training and evaluating security tools and models. Covers thousands of CVEs across numerous projects, programming languages, and CWE types. Specifically designed to facilitate data-driven security research using source code analysis and metrics. Benefits from automated collection processes combined with curation efforts [44].

In this study VulnPatchPairs dataset was used for the primary iterative development and ablation study due to its structure of paired vulnerable/patched functions providing clear ground truth for binary classification at the function level. This dataset is considered hard and used in another papers to show how state of the art solutions fail [38]. The CVEFixes dataset was used for cross-dataset validation to assess the generalizability of the findings on a different, widely used vulnerability dataset. For both datasets, individual functions were randomly extracted and analyzed without providing patch information during classification.

3.3.4 LLM Configuration

Models used: DeepSeek (deepseek-v3) Llama (llama3-70b-8192), Mistral (mistral-saba-24b).

Key Parameters: temperature: 0.1, max tokens: 512.

Prompt Formatting: Input was structured using JSON format containing keys like code_snippet and optional context, along with the detailed role-playing instructions and analytical steps defined in the prompt variations.

Execution: 250 runs a pair of code snippets per prompt were performed for each prompt variation on the VulnPatchPairs dataset subset resulting in 500 runs per prompt to average results and account for potential LLM stochasticity, although a low temperature aims to minimize this. For CVEFixes dataset there are 500 runs per prompt with 250 vulnerable and 250 patched code snippets with average of 20 fails due to issues with LLM API.

3.4 Validation Plan

The validation plan ensured a systematic approach to prompt refinement and evaluation.

Iterative Refinement (Pilot/Development Phase)

This phase corresponds to the multiple prompt versions (V8-V11) tested. A subset of the VulnPatchPairs dataset 250 pairs was used. Prompts were iteratively adjusted based on F1-score, Recall, Precision, and FPR metrics. Qualitative review of LLM outputs/ justifications helped diagnose issues (e.g., over-conservatism, poor mitigation assessment). Randomization of snippet order and consistent LLM parameters were used to mitigate bias.

Ablation Study

Performed on the best-balanced prompt V9 identified during iteration to understand component contributions (as detailed in 3.2.4).

Full Experiment (Comparison)

The performance of the best identified TRIZ prompts (e.g., V9, potentially Variation 2 from ablation) was compared against baseline prompts (CoT and DFA from [9]) on a designated test split of the VulnPatchPairs dataset.

Cross-Dataset Validation: To assess robustness, the same set of prompts (TRIZ baseline V9, ablation variations, CoT and DFA baselines) were also evaluated on the CVEFixes dataset subset, which contains randomly picked 500 code snippets.

Quantitative metrics were calculated and statistical significance t-tests planned for comparing F1-scores.

Qualitative analysis of outputs on a random sample of 100 pairs compared reasoning quality.

Cross-validation was considered and performed on the dataset to ensure robustness across different data splits.

Threats to Validity

Internal Validity could be threatened by LLM Stochasticity. Despite chosen parameters outputs may still be different for the same queries. Getting an average results from significant amount of runs per each prompt will help to mitigate this.

External Validity threat is due to possible issues with universality, because received results may be irrelevant for niche or non-standard programming languages like Rust or Perl or proprietary and closed-source codebases.

Construct Validity could be threatened by metrics limitations. F1-score has Precision/Recall balance in priority, but it may miss nuanced false positives like wrong classifications of code patterns. Combination of these metrics and human evaluation will help to mitigate this.

Context Limitation: Analyzing single functions without full project context inherently limits the ability to definitively assess exploitability and external mitigations, potentially affecting both FN and FP rates for any static analysis approach, including LLM-based ones.

Dataset Bias: Both VulnPatchPairs and CVEFixes datasets, while large and real-world, may have biases in the types of projects, or vulnerability patterns represented.

Dataset Differences: Performance variations observed between VulnPatchPairs and CVEFixes highlight potential dataset biases or differences in vulnerability characteristics, impacting the absolute generalizability of results.

LLM Reliability: A non-trivial rate of null/error outputs was observed (approx. 4-8% depending on dataset/prompt), indicating potential reliability issues with the LLM API or prompt processing that could affect overall practical deployment.

3.5 Example TRIZ Prompt Application

This section provides conceptual illustrations of how selected TRIZ principles could be layered in prompts for specific vulnerability types. These are distinct from the experimentally refined prompts used for classification but serve to demonstrate the underlying TRIZ thinking.

TRIZ-prompts shown in this section demonstrate how TRIZ principles can be layered for more deep and adaptive code security analysis. These examples target OWASP Top 10 vulnerabilities as the most popular flaws currently for demonstration purpose.

Example 1. Segmentation, Nested Doll, Feedback.

Prompt: “List all dynamic memory allocations (e.g., functions reserving memory). For each allocation, check if buffer size is validated against input. Then, verify loop bounds against buffer capacity. If unsafe loops are found, re-analyze error-handling blocks for missed validations.”

This example targets Buffer Overflow vulnerability. TRIZ Synergy consists of hierarchical checks (Nested Doll) for segmented tasks (Segmentation) and iterative refinement based of prior findings (Feedback).

Example 2. Preliminary Action, Asymmetry, Dynamicity.

Prompt: “Study examples of unsafe queries (e.g., string concatenation). Identify "raw input embedded in queries" OR "lack of parameterization". If unsafe patterns are detected, focus on adjacent code for secondary risks (e.g., privilege escalation).”

This example targets SQL injections. The prompt first trains LLM (Preliminary Action), then asks to use different wordings (Asymmetry), and then changes focus to find secondary risks (Dynamicity).

Example 3. Preliminary Anti-Action, Segmentation, Feedback.

Prompt: “- Step 1: Detect encoded/encrypted strings (e.g., base64, hex). - Step 2: Trace how decoded values are used. Check for dynamic execution (e.g., functions that evaluate strings as code). If suspicious execution is found, re-analyze call chains for hidden payloads.”

Here Code Obfuscation is being looked for. The example looks for hidden threats (Preliminary Anti-Action) through two-step analysis (Segmentation) and iteration (Feedback).

Example 4. Nested Doll, Asymmetry, Dynamicity.

Prompt: “- Layer 1: Find user input rendered in output (e.g., UI, logs). - Layer 2: For each input, check if context-specific encoding is applied. Look for "missing sanitization" OR "unsafe encoding bypasses". If encoding is missing, search for DOM manipulation (e.g., `innerHTML`, `document.write`).”

This prompt is meant to identify Cross-Site-Scripting (XSS) vulnerabilities. First it performs layered analysis (Nested Doll) and uses different terminology (Asymmetry), then adjusts the scope.

Example 5. Segmentation, Preliminary Action, Feedback.

Prompt: “- Task 1: Identify dynamic resource allocations (e.g., memory, files). - Task 2: Check deallocation in all code paths (e.g., returns, exceptions). Review common leak patterns (e.g., missing `free()` in loops). If leaks are found, re-analyze error-handling logic.”

This example looks for Memory Leaks. To achieve that it divides the task (Segmentation), then primes the LLM (Preliminary Action), and finally asks to perform the task iteratively (Feedback).

Example 6. Nested Doll, Preliminary Anti-Action, Asymmetry.

Prompt: “- Layer 1: Locate file operations using user-controlled input. - Layer 2: Verify path normalization/validation. Check for indirect traversal (e.g., `../` encoded as URL parameters). Search for "unsafe path concatenation" OR "missing canonicalization".”

This prompt targets Path Traversal vulnerability. It performs hierarchical check (Nested Doll), then tries to detect hidden bypasses (Preliminary Anti-Action), and uses varied wording (Asymmetry).

Example 7. Dynamicity, Feedback, Segmentation.

Prompt: “- Step 1: Identify system command executions. - Step 2: Check if user input is sanitized. If unsanitized input is found, analyze for chained attacks (e.g., `;`, `&&`). For chained attacks, re-analyze input sources for validation gaps.”

This example looks for Command Injections. It uses phased analysis (Segmentation), adaptive focus (Dynamicity), and iterative analysis (Feedback).

Example 8. Asymmetry, Nested Doll, Preliminary Anti-Action.

Prompt: “- Layer 1: Locate deserialization functions. - Layer 2: Check for validation (e.g., allowed classes, data integrity). Identify "unvalidated deserialization" OR "type confusion risks". Detect tampering (e.g., modified serialized objects).”

This prompt targets Insecure Deserialization flaws. It performs hierarchical checks (Nested Doll), then different terminology (Asymmetry), and anti-tampering (Preliminary Anti-Action).

Example 9. Preliminary Action, Segmentation, Feedback.

Prompt: “Study examples of hardcoded credentials (e.g., `password = "admin";`). - Task 1: Search for variables storing secrets. - Task 2: Check if values are literals or fetched securely. If hardcoded values exist, re-analyze logs/configs for exposure.”

This example looks for Hardcoded Secrets. Firstly, it trains LLM with an example (Preliminary Action), then splits the task (Segmentation), finally, performs the task in iteration (Feedback).

Example 10. Dynamicity, Nested Doll, Preliminary Anti-Action.

Prompt: “- Layer 1: Identify shared resource accesses (e.g., files, databases). - Layer 2: Check for synchronization mechanisms (e.g., locks, transactions). If synchronization is missing, analyze for time-of-check-to-time-of-use (TOCTOU) flaws. Detect lazy initialization patterns that might hide races.”

This example targets Race Conditions using hierarchical analysis (Nested Doll), adaptive focus (Dynamicity), and anti-patterns detection (Preliminary Anti-Action).

4 Results

This chapter presents the empirical findings from the development and evaluation of TRIZ-based prompts designed for LLM-driven vulnerability detection in source code. The results encompass the iterative refinement process undertaken to balance performance metrics and a systematic ablation study conducted on the most promising prompt variant identified during iteration. The experiments utilized the VulnPatchPairs and CVEFixes datasets and the LLMs DeepSeek, Llama, and Mistral as detailed in the Methodology chapter (Section 3.3). Performance was primarily evaluated using F1-score, Recall, Precision, and FPR and compared against a standard CoT and DFA baselines adapted from Tamberg et al. [9]. Additionally, this chapter presents conceptual results derived from applying TRIZ system thinking to the broader lifecycle of software code, exploring vulnerabilities beyond static analysis and proposing novel defense concepts. These findings provide insights not only into prompt engineering for code analysis but also inform the broader challenge of applying systematic problem-solving like TRIZ to the vulnerability lifecycle, including code transmission, storage, and execution contexts.

4.1 Iterative Prompt Development Results

An iterative approach was necessary to tune the TRIZ-based prompt structure and logic, aiming to optimize the F1-score while managing the inherent trade-off between Recall (detecting true vulnerabilities) and Precision (avoiding false alarms). The initial goal was to significantly outperform standard baselines, such as CoT, which demonstrated very limited detection capabilities in initial tests (see Section 4.2.2). Several major versions (detailed implicitly through the baseline V9 and the extremes like V8/V11 derived from earlier iterations described in Chapter 3) were tested, revealing key performance trends.

Initial Conservative Prompts (e.g., V8): Early versions focusing heavily on detailed TRIZ analysis and requiring high confidence for a "YES" classification exhibited very high Precision but suffered from extremely low Recall (e.g., Recall ≈ 0.06 - 0.16) and consequently low F1-scores (e.g., F1 ≈ 0.11 - 0.25). These prompts were too conservative, missing the vast majority of actual vulnerabilities.

Recall-Focused Prompts (e.g., V11): Adjustments were made to lower the threshold for flagging potential vulnerabilities, prioritizing Recall. This approach (V11) achieved the highest Recall observed (0.8240) and the highest F1-score (0.6252). However, this came

at the cost of very poor Precision (0.5037) and an extremely high FPR (0.8120), rendering it impractical due to excessive noise.

Balanced Baseline Selection (V9): Through refinement cycles aiming for a better balance (e.g., V9, V10), Prompt V9 was identified as providing a moderate compromise. It achieved an F1-Score of 0.4539 with Recall 0.4040, Precision 0.5179, and FPR 0.3760. While not optimal, its relatively balanced profile made it suitable for a more detailed component analysis via ablation.

Table 2 summarizes the key metrics for selected prompts during the iterative development phase, illustrating the Recall/Precision trade-off and the performance gap compared to the CoT baseline.

Table 2: Performance Metrics During Iterative Prompt Development

Prompt	F1-Score	Recall	Precision	FPR	Notes
CoT Baseline	0.05	0.03	0.36	0.04	Very Low Recall/F1, High Errors
V8	0.16	0.09	0.53	0.08	High Precision, Very Low Recall
V9	0.45	0.4	0.52	0.38	Moderate Balance, Used for Ablation
V10	0.25	0.16	0.54	0.14	Attempt to Improve V9, Recall Decreased
V11	0.63	0.82	0.5	0.81	Highest F1/Recall, Very High FPR

The iterative process demonstrated the sensitivity of the LLM's performance to the specific instructions regarding TRIZ analysis application, indicator confidence thresholds, and mitigation assessment logic. It highlighted the difficulty in achieving both high Recall and high Precision simultaneously with single-function analysis using this methodology. It also demonstrates significant improvement upon the CoT baseline but nevertheless still facing challenges in achieving high performance on both Recall and Precision simultaneously.

4.2 Ablation Study Results

To dissect the contribution of individual components within the V9 baseline prompt, an ablation study was performed. Six variations were created, each removing or simplifying one key aspect of the prompt. The performance of each variation was compared against the baseline.

4.2.1 Baseline Performance (Prompt V9)

The baseline prompt (full TRIZ framing, structured indicators, mitigation confidence assessment, justification required) yielded the following metrics: TP: 101, TN: 156, FP: 94, FN: 149, Accuracy: 0.5140, Precision: 0.5179, Recall: 0.4040, F1-Score: 0.4539, FPR: 0.3760.

4.2.2 Analysis of Ablated Components

Table 3 presents the results for the baseline and all six ablation study variations.

Table 3: Ablation Study Results Comparison

Prompt	F1-Score	Recall	Precision	FPR	Accuracy	TP	TN	FP	FN	Change vs V9 F1
CoT Baseline	0.05	0.04	0.36	**0.0439**	0.51	5	196	9	187	-0.40
DFA Baseline	0.026	0.126	0.5167	0.1179	0.5041	31	217	29	215	-0.424
V9	0.45	0.4	0.52	0.38	0.51	101	156	94	149	---
Var 1	0.4	0.31	0.54	0.26	0.52	78	184	66	172	-0.05
Var 2	0.47	0.39	**0.58**	0.29	**0.55**	98	178	72	152	0.07
Var 3	0.32	0.23	0.51	0.21	0.51	52	179	49	176	-0.13
Var 4	**0.55**	**0.62**	0.5	0.61	0.5	154	98	152	96	0.10
Var 5	0.51	0.49	0.54	0.42	0.53	122	144	106	128	0.11
Var 6	0.33	0.25	0.47	0.28	0.49	62	181	69	188	-0.12

Analysis of each variation compared to the baselines: CoT and DFA Baselines Performance: The standard CoT and DFA baselines adapted from [9] performed very poorly on this dataset and task, achieving an F1-Score of only 0.05 and 0.026 and Recall of 0.04 and 0.126 correspondingly. While their FPRs were excellent (0.0439 and 0.1179), they failed to detect almost all vulnerabilities. Furthermore, they exhibited significant reliability issues, failing to produce valid classifications for over 20% of the inputs (103 and 111 nulls/errors out of 500 attempts).

All TRIZ Variations vs CoT and DFA: All tested TRIZ prompt variations (including the baseline V9 and its ablations) demonstrated substantially higher F1-Scores and Recall compared to the CoT and DFA baselines, indicating a clear benefit from the structured TRIZ approach for this task, despite the challenges in balancing metrics. The TRIZ prompts also appeared more reliable, yielding fewer processing errors.

Variation 1 (Ablate TRIZ Terms): Performance decreased (F1: 0.3959), particularly Recall. This suggests that using explicit TRIZ terminology (Function Analysis, Resource Analysis, Contradiction, Principles) provided a beneficial structure or focus for the LLM compared to generic equivalents.

Variation 2 (Ablate Contradiction Analysis): This variation showed a slight improvement in F1-score (0.4667) and Accuracy (0.5520). Crucially, it significantly improved Precision (0.5765) and reduced the FPR (0.2880) while maintaining Recall (0.3920) close to the baseline. This indicates that the explicit instruction to identify contradictions (Step 2c) in the baseline prompt was likely redundant or slightly counterproductive, possibly adding noise without improving detection.

Variation 3 (Simplify TRIZ Framing): Performance dropped significantly (F1: 0.3161). Simplifying the detailed Function, Resource, and Contradiction analysis steps into a

single high-level instruction was detrimental. This highlights the importance of the structured, multi-faceted analysis guided by the TRIZ concepts in Step 2 of the baseline prompt.

Variation 4 (Ablate Indicator Structure): This variation yielded the highest F1-score (0.5540) and Recall (0.6160). However, removing the specific A-D indicator structure led to a drastic increase in False Positives (FPR: 0.6080) and slightly lower Precision (0.5033) than the baseline. This suggests the indicator structure is key to controlling false positives but may be too restrictive for maximizing recall.

Variation 5 (Simplify Mitigation Assessment): Replacing the "Strong vs. Weak/No" mitigation assessment with a simpler boolean check ("Present" vs. "Absent") resulted in a better F1-score (0.5105) than the baseline. It improved both Recall (0.4880) and Precision (0.5351) moderately. This indicates the simpler boolean check might be a more effective way for the LLM to handle mitigation assessment in this context compared to the more nuanced confidence levels in the baseline. However, the FPR (0.4240) was still higher than the baseline.

Variation 6 (Ablate Justification): Performance decreased significantly (F1: 0.3255). Removing the requirement for the LLM to generate a justification negatively impacted its classification ability. This suggests the internal process of formulating a rationale is beneficial for the LLM's accuracy, even if only the final classification is used.

4.3 Results Analysis

The ablation study provides clear evidence supporting the utility of a structured, TRIZ-inspired approach for prompting LLMs in this task, while also highlighting areas for optimization.

Effectiveness of TRIZ Components vs. CoT and DFA: The necessity of detailed Function/Resource analysis (Var 3 failure) and the benefit of TRIZ terminology (Var 1 results) strongly suggest the TRIZ analytical framework guided the LLM much more effectively than the standard CoT and DFA approach, which yielded minimal vulnerability detection ($F1 \approx 0.05$ and 0.026). The structured decomposition and analysis inherent in the TRIZ prompts appear crucial for improving performance beyond simple step-by-step reasoning on this task. The value added by explicit Contradiction analysis seems less pronounced (Var 2 results).

Controlling the Recall/Precision Trade-off: The results vividly illustrate the Recall/Precision contradiction. The definition of vulnerability indicators (Step 3) and the logic for assessing mitigation (Step 4/5) act as key control parameters. Loosening indicator definitions (Var 4) maximizes Recall but floods results with FPs. Refining the mitigation check (Var 5) offers a path to improve F1 over the baseline. The justification step (Var

6) also unexpectedly influenced this balance, likely by enforcing more careful internal analysis.

Limitations and Context: Achieving significantly higher F1 scores likely requires overcoming the inherent limitation of function-level analysis. The LLM's inability to access broader program context (caller functions, global state, configurations, runtime data) restricts its ability to definitively verify exploitability or the effectiveness of external mitigations. This likely contributes to both FPs (flagging theoretically possible but practically unexploitable issues) and FNs (missing vulnerabilities that depend on inter-procedural context).

4.3.1 Selection of Optimal Prompts based on Ablation Study

The ablation study, compared against both the TRIZ baseline (V9) and the standard CoT and DFA baselines, revealed that no single prompt achieved perfect performance. However, several TRIZ variations offered significant improvements over standard CoT and DFA. The choice of the "best" TRIZ prompt depends on the desired balance.

Highest F1-Score (at cost of high FPR): Variation 4 (Ablate Indicator Structure) achieved the highest F1 (0.5540), vastly outperforming CoT and DFA, but its very high FPR (0.6080) limits practicality.

Best Precision / Lowest FPR Improvement: Variation 2 (Ablate Contradiction Analysis) offered the best Precision (0.5765) and a significantly reduced FPR (0.2880) compared to the TRIZ baseline, with a slightly improved F1 (0.4667). It represents a much more precise and reliable prompt than the baseline and vastly better detection than CoT.

Improved F1 with Balanced Metrics: Variation 5 (Simplify Mitigation Assessment) provided a good F1 (0.5105) by improving both Recall and Precision over the TRIZ baseline, also significantly outperforming CoT.

Based on these results, Variation 2 (Baseline without Contradiction Analysis) stands out as the recommended prompt configuration from this study if improving precision and reducing false positives relative to the TRIZ baseline V9 is the primary goal, while still vastly outperforming standard CoT and DFA. If maximizing F1 is the absolute priority despite high noise, Variation 4 would be chosen.

4.3.2 Cross-Dataset Validation Results (CVEFixes)

To assess the robustness and generalizability of the findings, the baseline prompts (CoT and TRIZ V9) and the key ablation study variations were subsequently evaluated on the CVEFixes dataset. This dataset, while also containing real-world vulnerabilities, differs in its composition and characteristics from VulnPatchPairs.

Table 4 presents the performance metrics on the CVEFixes dataset. It should be noted that all prompts exhibited a failure rate of approximately 4% (≈20 null/error outputs per

500 inputs attempted) on this dataset, indicating some processing challenges. The metrics reported are based on the valid predictions obtained (≈480 per prompt).

Table 4: Prompt Performance Comparison on CVEFixes Dataset

Prompt	F1-Score	Recall	Precision	FPR	Accuracy	TP	TN	FP	FN	Change vs Baseline F1
CoT Baseline	0.3779	0.2838	**0.5652**	**0.20**	**0.5532**	65	200	50	164	-0.1309
DFA Baseline	0.3706	0.2903	0.5122	0.2469	0.5348	63	183	60	154	-0.1382
V9	**0.5088**	0.5	0.5180	0.428	0.5375	115	143	107	115	---
Var 1	0.4331	0.3886	0.4890	0.3720	0.5136	89	157	93	140	-0.0757
Var 2	0.4575	0.4217	0.5	0.3880	0.5208	97	153	97	133	-0.0513
Var 3	0.3990	0.3304	0.5033	0.3	0.5229	76	175	75	154	-0.1098
Var 4	0.5233	**0.5609**	0.4905	0.5360	0.5104	129	116	134	101	0.0145
Var 5	0.5	0.5043	0.4957	0.4739	0.5157	116	131	118	114	-0.0088
Var 6	0.4389	0.3826	0.5146	0.3333	0.5303	88	166	83	142	-0.0699

Analysis of CVEFixes results: TRIZ vs. CoT and DFA: The TRIZ-based prompts (Baseline V9 and variations 2, 4, 5) consistently and significantly outperformed both CoT and DFA baselines in F1-Score (0.46 - 0.52 vs. 0.37 – 0.38) and Recall (0.39 - 0.56 vs. 0.28 – 0.29). This confirms the benefit of the structured TRIZ approach on a second dataset. However, CoT and DFA baselines achieved better Precision (0.57 and 0.51) and a much lower FPR (0.20 and 0.25) than most TRIZ variations on this dataset.

Ablation Impact Consistency: The relative impact of most ablations remained broadly similar to the VulnPatchPairs results: removing TRIZ terms (Var 1), simplifying framing (Var 3), and removing justification (Var 6) still resulted in lower F1 scores than the TRIZ baseline V9. Ablating the indicator structure (Var 4) again produced the highest F1 and Recall, but also the highest FPR. Simplifying mitigation (Var 5) again performed well, close to the baseline F1.

Key Difference - Var 2: Notably, Variation 2 (Ablate Contradiction), which slightly outperformed the baseline on VulnPatchPairs and significantly improved its Precision/ FPR, performed worse than the baseline on CVEFixes (F1: 0.4575 vs 0.5088). This suggests the utility of the explicit Contradiction Analysis step might be dataset-dependent.

Best Performers on CVEFixes: Variation 4 (Ablate Indicators) achieved the highest F1 (0.5233) and Recall (0.5609), followed closely by the original TRIZ Baseline V9 (F1=0.5088) and Variation 5 (Simplify Mitigation, F1=0.5000). However, all these top performers had high FPRs (0.43 - 0.54). The CoT and DFA baselines, while poor at detection, were the least noisy (FPR was 0.20 and 0.25 correspondingly).

These cross-dataset results confirm the general advantage of the structured TRIZ prompting approach over CoT and DFA for recall and F1-score, but also highlight per-

formance variability between datasets and the persistent challenge of high false positive rates with the higher-recall TRIZ prompts.

4.3.3 Cross-LLM Validation Results

Table 5: Prompt Performance Comparison on VulnPatchPairs Dataset for DeepSeek, Llama, Mistral

LLM	Prompt	F1-Score	Recall	Precision	FPR	Accuracy	TP	TN	FP	FN	Change vs V9 F1	Errors
DeepSeek	CoT	0.05	0.03	0.36	0.04	0.51	5	196	9	187	-0.40	20.6%
	DFA	0.20	0.13	0.52	0.12	0.50	31	217	29	215	-0.25	1.6%
	V9	0.45	0.40	0.52	0.38	0.51	101	156	94	149	---	0%
	Var 4	0.55	0.62	0.50	0.61	0.50	154	98	152	96	0.10	0%
	Var 5	0.51	0.49	0.54	0.42	0.53	122	144	106	128	0.06	0%
Llama	CoT	0.25	0.17	0.47	0.24	0.43	9	32	10	44	-0.11	81%
	DFA	0.65	0.96	0.49	0.96	0.50	232	11	237	9	0.29	2.2%
	V9	0.36	0.27	0.55	0.22	0.53	67	194	54	182	---	0.6%
	Var 4	0.50	0.51	0.50	0.50	0.50	124	123	125	119	0.14	1.8%
	Var 5	0.42	0.37	0.50	0.38	0.49	91	149	92	158	0.06	2%
Mistral	CoT	0.41	0.35	0.49	0.37	0.48	63	111	66	119	0.25	28.2%
	DFA	0.60	0.75	0.50	0.76	0.49	188	59	191	62	0.44	0%
	V9	0.16	0.10	0.59	0.07	0.51	24	233	17	226	---	0%
	Var 4	0.23	0.15	0.53	0.13	0.51	36	217	32	210	0.07	1%
	Var 5	0.16	0.09	0.61	0.06	0.52	23	235	15	226	0.00	0.2%

Table 6: Prompt Performance Comparison on CVEFixes Dataset for DeepSeek, Llama, Mistral

LLM	Prompt	F1-Score	Recall	Precision	FPR	Accuracy	TP	TN	FP	FN	Change vs V9 F1	Errors
DeepSeek	CoT	0.38	0.28	0.57	0.20	0.55	65	200	50	164	-0.13	4.2%
	DFA	0.37	0.29	0.51	0.24	0.53	63	183	60	154	-0.14	8%
	V9	0.51	0.5	0.52	0.43	0.54	115	143	107	115	---	4%
	Var 4	0.52	0.56	0.49	0.54	0.51	129	116	134	101	0.01	4%
	Var 5	0.5	0.50	0.50	0.47	0.52	116	131	118	114	-0.01	4.2%
Llama	CoT	0.47	0.52	0.43	0.57	0.47	77	76	102	72	0.05	34.6%
	DFA	0.57	0.81	0.44	0.86	0.45	126	27	160	29	0.15	31.6%
	V9	0.42	0.38	0.47	0.36	0.52	56	114	63	92	---	35%
	Var 4	0.48	0.48	0.49	0.40	0.54	70	107	72	77	0.06	34.8%
	Var 5	0.43	0.39	0.47	0.37	0.52	58	113	66	89	0.01	34.8%
Mistral	CoT	0.53	0.56	0.49	0.49	0.54	105	114	108	81	0.14	18.4%
	DFA	0.56	0.73	0.46	0.77	0.47	144	50	170	53	0.17	16.6%
	V9	0.39	0.35	0.45	0.39	0.48	73	141	91	138	---	11.4%
	Var 4	0.40	0.38	0.43	0.47	0.46	82	125	110	132	0.01	10.2%
	Var 5	0.38	0.32	0.47	0.33	0.50	69	154	77	144	-0.01	11.2%

In this section both Table 5 and Table 6 show the experiment results for VulnPatchPairs dataset and CVEFixes dataset correspondingly using DeepSeek, Llama, and Mistral

LLMs. Big amount of API errors for this cycle of experiment is shown in Errors column. Metrics are calculated using received non-null results.

Detailed Performance Analysis of Prompts Across LLMs and Datasets

The cross-LLM validation reveals a complex interplay between the prompting methodology (TRIZ-based vs. baselines), the specific LLM utilized, and the characteristics of the dataset. The initial hypothesis that TRIZ-based prompts would consistently outperform baselines across all scenarios requires significant nuance based on these results.

Performance with DeepSeek LLM.

VulnPatchPairs Dataset (Table 5): With DeepSeek, the TRIZ-based prompts (V9, Var 4, Var 5) demonstrated notably better F1-Scores (0.45 to 0.55) and Recall (0.40 to 0.62) compared to both CoT (F1=0.05, Recall=0.03) and DFA (F1=0.20, Recall=0.13) baselines. Var 4 achieved the highest F1 and Recall but also the highest FPR (0.61). DeepSeek exhibited 0% errors with TRIZ prompts on this dataset.

CVEFixes Dataset (Table 6): Similarly, on CVEFixes, DeepSeek with TRIZ prompts (V9, Var 4, Var 5) yielded higher F1-Scores (0.50 to 0.52) and Recall (0.50 to 0.56) than CoT (F1=0.38, Recall=0.28) and DFA (F1=0.37, Recall=0.29). Again, Var 4 had the highest Recall and F1, coupled with the highest FPR. Error rates for DeepSeek were around 4% on this dataset.

Conclusion for DeepSeek: For the DeepSeek LLM, the structured TRIZ approach generally led to more effective vulnerability detection in terms of F1-score and Recall compared to the CoT and DFA baselines on both datasets.

Performance with Llama LLM.

VulnPatchPairs Dataset (Table 5): (1) The performance of TRIZ prompts with Llama was mixed when compared to baselines. While TRIZ V9 (F1=0.36), Var 4 (F1=0.50), and Var 5 (F1=0.42) outperformed the CoT baseline (F1=0.25), the DFA baseline achieved the highest F1-Score (0.65) and Recall (0.96) among all Llama combinations. (2) However, this superior DFA performance came with an extremely high FPR (0.96). The CoT prompt with Llama had a very high error rate (81%).

CVEFixes Dataset (Table 6): (1) On this dataset, the baseline prompts generally outperformed the TRIZ prompts with Llama in terms of F1 and Recall. DFA (F1=0.57, Recall=0.81) and CoT (F1=0.47, Recall=0.52) were higher than V9 (F1=0.42, Recall=0.38), Var 4 (F1=0.48, Recall=0.48), and Var 5 (F1=0.43, Recall=0.39). (2) It is critical to note that all prompts (TRIZ and baselines) with Llama on CVEFixes exhibited very high error rates (around 31-35%), making the reliability of these specific metrics a concern.

Conclusion for Llama: The TRIZ prompts did not consistently outperform baselines. DFA showed high F1/Recall on VulnPatchPairs (despite high FPR), and both CoT/DFA

showed higher F1/Recall on CVEFixes, though all Llama results, especially on CVE-Fixes, are impacted by high error rates.

Performance with Mistral LLM.

VulnPatchPairs Dataset (Table 5): With Mistral, the baseline CoT (F1=0.41, Recall=0.35) and particularly DFA (F1=0.60, Recall=0.75) prompts significantly outperformed all tested TRIZ variations (V9 F1=0.16, Var 4 F1=0.23, Var 5 F1=0.16).

CVEFixes Dataset (Table 6): (1) A similar trend was observed on CVEFixes, where CoT (F1=0.53, Recall=0.56) and DFA (F1=0.56, Recall=0.73) again outperformed the TRIZ prompts V9 (F1=0.39), Var 4 (F1=0.40), and Var 5 (F1=0.38) in both F1-score and Recall. (2) Error rates for Mistral were notable, especially CoT on VulnPatchPairs (28.2%) and across all prompts on CVEFixes (10-18%).

Conclusion for Mistral: For the Mistral LLM, the standard CoT and DFA baseline prompts were generally more effective in terms of F1-score and Recall than the specific TRIZ prompts evaluated on both datasets.

General Observations on Baseline Prompt Performance

The CoT and DFA baselines showed very low F1-scores and Recall with the DeepSeek LLM on the VulnPatchPairs dataset. On the CVEFixes dataset, CoT and DFA baselines, while often having lower Recall than TRIZ prompts with DeepSeek, sometimes achieved better Precision and lower FPR (e.g., DeepSeek CoT FPR=0.20, DFA FPR=0.24; Mistral CoT FPR=0.49, DFA FPR=0.77, though DFA with Mistral on CVE-Fixes had very high FPR). Their performance was notably stronger with Mistral on CVEFixes in terms of F1 and Recall compared to TRIZ prompts with the same LLM.

Prompt Variation Consistency (TRIZ Prompts V9, Var 4, Var 5)

Var 4: Across different LLMs, Var 4 often (though not universally) produced higher F1-scores and Recall compared to its V9 baseline (e.g., for DeepSeek and Llama on VulnPatchPairs, and for Llama on CVEFixes). However, this was almost invariably accompanied by the highest FPR among the TRIZ variants for that LLM, indicating it consistently acts as a high-recall/low-precision prompt. Its F1 performance with Mistral was still below Mistral's baselines.

Var 5: This prompt often provided a performance profile close to V9, sometimes slightly better in F1 (e.g., DeepSeek and Llama on VulnPatchPairs). It appeared as a relatively stable TRIZ variation but did not fundamentally change the performance hierarchy against baselines for Llama or Mistral.

V9: While serving as the reference for TRIZ prompts, its performance relative to its own variations and to baselines was highly dependent on the LLM and dataset.

Impact of Error Rates

The "Errors" column in Tables 5 and 6 is critical for interpreting practical usability. Llama exhibited particularly high error rates, especially with CoT on VulnPatchPairs (81%) and across all prompts on CVEFixes (around 35%). Mistral also showed significant error rates, notably with CoT on VulnPatchPairs (28.2%) and ranging from 10-18% for most prompts on CVEFixes. DeepSeek was the most reliable, with 0% errors for its TRIZ prompts on VulnPatchPairs and approximately 4% for most prompts on CVEFixes. High error rates diminish the practical utility of an LLM/prompt combination, as metrics are calculated only on successful responses. This is a major concern for real-world deployment.

Synthesis of Cross-LLM Findings

The cross-LLM validation underscores that there is no one-size-fits-all solution for prompting LLMs for vulnerability detection. The structured TRIZ approach showed clear benefits in F1-score and Recall with the DeepSeek LLM compared to CoT and DFA baselines. However, for Llama and Mistral, the CoT and DFA baselines often demonstrated competitive or even superior F1 and Recall performance compared to the tested TRIZ prompts, although these baseline performances also came with their own caveats (e.g., extremely high FPR for Llama+DFA on VulnPatchPairs, or significant error rates). The reliability of the LLM itself (i.e., low error rates in generating responses) is a crucial factor, where DeepSeek performed best in this study. These findings suggest that while TRIZ offers a valuable framework for structuring complex analytical prompts, its relative effectiveness against simpler baselines can be highly dependent on the internal architecture and training of the specific LLM being used.

5 Discussion

This chapter synthesizes and interprets the findings presented in Chapter 4, contextualizing them within the existing literature and addressing the research questions posed in Chapter 1. It discusses the effectiveness and challenges of applying TRIZ principles both to LLM prompt engineering for vulnerability detection and to the conceptual analysis of the broader code vulnerability lifecycle. The chapter evaluates the strengths and limitations of the conducted research and outlines implications for practitioners and future research directions.

5.1 Summary of Key Findings

The research yielded two primary sets of results: empirical findings from LLM prompt experiments and conceptual findings from systemic TRIZ analysis.

An iterative prompt development process for LLM-based vulnerability detection using single-function analysis demonstrated a significant trade-off – a Contradiction in TRIZ terms – between maximizing Recall (detecting true vulnerabilities) and maximizing Precision (minimizing FPR). Prompts tuned for high Recall (like V11, which achieved an F1-score of approximately 0.63 with DeepSeek) tended to suffer from impractically high FPRs (around 0.81 for V11 with DeepSeek on VulnPatchPairs), while highly precise prompts (like V8 or V10, with F1-scores between approximately 0.16-0.25 with DeepSeek) exhibited very low Recall.

A key finding was the varying effectiveness of the developed TRIZ-based prompts (V9 and its variations, notably Var 4 and Var 5) when compared to standard CoT and DFA baselines across different LLMs and datasets. With the DeepSeek LLM, TRIZ-based prompts consistently demonstrated superior F1-Score and Recall performance on both the VulnPatchPairs and CVEFixes datasets. However, this pattern of TRIZ superiority was not uniformly replicated with Llama and Mistral.

With Llama on VulnPatchPairs, the DFA baseline achieved a higher F1-score and Recall than the TRIZ prompts, albeit with a very high FPR; on CVEFixes, both CoT and DFA baselines generally outperformed TRIZ prompts in F1 and Recall, with all Llama results impacted by high error rates.

With Mistral on both datasets, the CoT and DFA baselines generally yielded higher F1-Scores and Recall than the tested TRIZ prompts. These results indicate that while the structured TRIZ methodology can significantly enhance performance with some LLMs

like DeepSeek, its advantage over baselines is LLM-dependent, and influenced by factors including the LLM's architecture and training, as well as dataset characteristics (Tables 5 and 6).

The cross-LLM validation (Tables 5 and 6) highlighted significant performance and reliability differences among the LLMs: DeepSeek emerged as the most consistent and reliable LLM in this study, particularly when using TRIZ prompts, exhibiting low error rates (0% for TRIZ prompts on VulnPatchPairs, around 4% on CVEFixes).

Llama displayed potential for high Recall with certain prompts (e.g., DFA on VulnPatchPairs: F1=0.65, Recall=0.96, but FPR=0.96). However, it frequently suffered from extremely high error rates (e.g., 81% with CoT on VulnPatchPairs; ~35% across all prompts on CVEFixes), which significantly impacts its practical usability.

Mistral showed competitive F1-scores with baseline CoT and DFA prompts, particularly on the CVEFixes dataset (F1=0.53 and F1=0.56, respectively), often outperforming its TRIZ counterparts for these metrics. Its performance with TRIZ prompts was more varied, and it also exhibited notable error rates with some configurations (e.g., 28.2% with CoT on VulnPatchPairs; 10-18% across prompts on CVEFixes). These variations emphasize that LLM choice is critical, with operational reliability (i.e., low error rates) being a major consideration alongside traditional accuracy metrics.

The systematic ablation study conducted on the V9 prompt (which achieved a moderate F1-score of approximately 0.45 with DeepSeek on VulnPatchPairs) provided insights into prompt component contributions. Detailed TRIZ-based Function/Resource analysis framing, the use of explicit TRIZ terminology, and requiring justification generation were identified as crucial prompt components for optimal performance with DeepSeek. Explicit Contradiction analysis, as defined in prompt V9, appeared less critical with DeepSeek, and its impact varied by dataset; its removal (Var 2) slightly improved F1 and notably improved Precision and FPR with DeepSeek on VulnPatchPairs, but Var 2 performed less well than V9 with DeepSeek on the CVEFixes dataset.

When key ablation variations were tested across LLMs: (1) Removing the structured vulnerability indicator definitions (Var 4) consistently resulted in higher Recall and often the highest F1-scores for DeepSeek and Llama (e.g., DeepSeek on VulnPatchPairs F1=0.55, Llama on VulnPatchPairs F1=0.50, DeepSeek on CVEFixes F1=0.52). However, this was almost invariably accompanied by a substantial increase in FPR, highlighting this component's role in managing the Recall/Precision trade-off across these models. With Mistral, Var 4's F1 did not surpass baselines. (2) Simplifying the mitigation check to a boolean assessment (Var 5) often provided a good balance of metrics and generally performed competitively with, or slightly better than, the V9 baseline across different LLMs and datasets, suggesting it's a relatively robust variation.

The relative performance of LLMs and prompt configurations also showed sensitivity to dataset characteristics, with different combinations excelling on VulnPatchPairs versus

CVEFixes. Overall, while the TRIZ-based structured prompting approach demonstrated potential for advancing vulnerability detection capabilities, particularly with DeepSeek, the study underscores that the path to consistently outperforming simpler baselines across all LLMs is not straightforward. The inherent limitations of static, single-function analysis mean the Recall/Precision trade-off remains a persistent challenge, and the high error rates and performance variability observed with some LLMs pose practical implementation hurdles.

5.2 Discussion of Empirical Prompting Results

5.2.1 Effectiveness and Comparison to Literature

The empirical study demonstrated that the structured, TRIZ-based prompting methodology can enhance vulnerability detection with general-purpose LLMs, though its superiority over standard CoT and DFA baselines is highly dependent on the specific LLM and dataset. With the DeepSeek LLM, a clear trend of improved performance for TRIZ-based prompts was observed across both the VulnPatchPairs and CVEFixes datasets. On both datasets with DeepSeek, the TRIZ prompt variations typically achieved F1-scores substantially higher than CoT and DFA, primarily driven by vastly improved Recall. For example, with DeepSeek on VulnPatchPairs, TRIZ prompts like V9 and its variants achieved F1-scores in the ~0.45-0.55 range, whereas CoT was ~0.05 and DFA ~0.20. This pattern of improved F1 and Recall for TRIZ prompts with DeepSeek was also evident on the CVEFixes dataset.

However, the cross-LLM validation with Llama and Mistral introduced important nuances to this finding. With Llama, while TRIZ prompts often outperformed its CoT baseline, the DFA baseline demonstrated a higher F1-score (0.65) and Recall (0.96) on VulnPatchPairs, albeit with an extremely high FPR (0.96). On the CVEFixes dataset, both CoT and DFA baselines generally achieved higher F1 and Recall than the TRIZ prompts when using Llama, though these results are subject to Llama's high error rates (around 35%) on that dataset. With Mistral, the CoT and DFA baselines were generally more effective, demonstrating stronger F1-scores and Recall on both datasets compared to the tested TRIZ variations. For instance, on CVEFixes, Mistral with CoT (F1=0.53) and DFA (F1=0.56) prompts outperformed its TRIZ counterparts. Similarly, on VulnPatchPairs, Mistral with DFA (F1=0.60) and CoT (F1=0.41) surpassed the TRIZ variants.

Consistent with findings from the initial DeepSeek experiments, the CoT and DFA baselines, when successfully executed, often yielded better Precision and lower FPR than many of the higher-Recall TRIZ variations, particularly on the CVEFixes dataset (e.g., with DeepSeek and Mistral). However, their utility for comprehensive vulnerability detection can be limited by their extremely low Recall with certain LLMs/datasets

(e.g., with DeepSeek on VulnPatchPairs, CoT Recall=0.03, DFA Recall=0.13), as they would miss the vast majority of issues in such cases.

Despite these LLM-specific variations, the overall results suggest that applying systematic analysis guided by TRIZ principles (such as Function and Resource Analysis) can be a valuable strategy for structuring LLM prompts, particularly for enhancing Recall with certain LLMs like DeepSeek. Nevertheless, the performance of general-purpose LLMs, even with enhanced TRIZ-based prompting, still generally falls short of what might be expected from specialized, fine-tuned models for vulnerability detection or the ideal capabilities of human experts, likely due to the single-function context limitation discussed further below and the inherent complexities of vulnerability analysis. The significant API error rates encountered with Llama and Mistral also highlight a practical challenge in relying on these models for consistent output in demanding analysis tasks.

5.2.2 Insights from Ablation Study

The ablation study, initially performed with the DeepSeek LLM (Table 3) and with key variations (Var 4, Var 5 alongside baseline V9) subsequently tested across Llama and Mistral (Tables 5 and 6), provided critical insights into the contribution of different TRIZ-derived prompt components. While some findings showed consistency across datasets and LLMs, others highlighted sensitivities.

The value of structure and specific TRIZ framing: The initial ablation study with DeepSeek indicated that removing detailed TRIZ framing (Function/Resource analysis, as in Var 3) or the justification generation step (Var 6) significantly degraded performance. Similarly, replacing explicit TRIZ terminology with generic equivalents (Var 1) also reduced performance, particularly Recall, with DeepSeek. These findings underscore that guiding the LLM's reasoning process with specific, structured steps derived from the TRIZ methodology is crucial for DeepSeek. Due to the selection of prompts for cross-LLM validation, the impact of these specific ablations (Var 1, Var 3, Var 6) was not directly measured for Llama and Mistral, but the general principle of structured prompting remains a key takeaway.

TRIZ component contributions – Contradiction Analysis: With DeepSeek, explicit Contradiction Analysis (Step 2c in V9) appeared less critical in the tested configuration. Its removal (Var 2) led to a slight improvement in F1-score (from 0.45 to 0.47) and a significant improvement in Precision (from 0.52 to 0.58) and FPR (from 0.38 to 0.29) on the VulnPatchPairs dataset. However, on the CVEFixes dataset, Var 2 performed slightly worse than V9 with DeepSeek (F1 0.4575 vs 0.5088), suggesting its utility might be dataset-dependent even for a single LLM. The performance of Var 2 was not evaluated for Llama and Mistral in Tables 5 and 6.

Sensitivity to Indicator Definition (Var 4): Ablating the structured A-D vulnerability indicators and using a more general instruction to identify high-confidence mechanisms

(Var 4) consistently resulted in the highest or near-highest F1-scores and Recall across all three LLMs (DeepSeek, Llama, Mistral) on both VulnPatchPairs and CVEFixes datasets. For instance, on VulnPatchPairs, Var 4 improved the F1-score over V9 for DeepSeek (+0.10), Llama (+0.14), and Mistral (+0.07). However, this gain in detection capability invariably came with a drastic increase in False Positives (FPR). This demonstrates that the specificity of indicator definitions is a powerful lever for controlling the Recall/Precision trade-off, and relaxing these definitions pushes all tested LLMs towards a high-recall, high-FPR mode.

Mitigation Check Simplicity (Var 5): Simplifying the mitigation assessment from nuanced confidence levels ("Strong" vs. "Weak/No") to a boolean check ("Present" vs. "Absent") (Var 5) yielded robust results. With DeepSeek on VulnPatchPairs, Var 5 improved the F1-score over V9 (0.51 vs 0.45). Across Llama and Mistral, Var 5 generally performed comparably to or slightly better than their respective V9 F1-scores on VulnPatchPairs, and showed mixed but generally close performance to V9 on CVEFixes for all LLMs. This suggests that a simpler, more concrete assessment of mitigation presence might be more reliably handled by the LLMs in a limited-context analysis, offering a good balance of performance across different models.

5.3 Conceptual Systemic TRIZ Analysis

This section involves applying broader TRIZ system thinking principles (System Operator, Contradictions, Ideality, Function/Resource Analysis, inventive principles like Intermediary, Segmentation) conceptually to model the entire code lifecycle (transmission, storage, execution). The aim was to identify systemic vulnerability points beyond static code and brainstorm potential TRIZ-inspired mitigation strategies (like the "Analog Filter" concept), linking them to existing security paradigms.

Beyond engineering specific LLM prompts for static code analysis, TRIZ principles can be applied more broadly to analyze the entire system involved in the lifecycle of software code – encompassing its creation, storage, transmission, and execution. This systemic perspective helps identify vulnerabilities that may arise not just from flaws within the code logic itself, but also from interactions between the code and the systems handling it.

5.3.1 System Decomposition and Problem Definition

Following TRIZ methodology, we first define the system and its inherent problems.

System: Software Code Lifecycle (including code artifacts, storage media, transmission channels, compilation/interpretation processes, execution hardware/software environments).

Initial Problem: Software code, while potentially appearing correct in its static form (analogous to an ideal blueprint or "holy scripture" in the brainstorming notes), becomes vulnerable when subjected to real-world processes.

Sub-problems (derived from System Operator / Multi-Screen thinking):

1. Static Code Quality: Vulnerabilities inherent in the code logic itself (the focus of Sections 3.2-3.4).
2. Transmission/Storage Integrity: Vulnerabilities arising during the transfer or storage of code, where the code (Resource) can be modified, corrupted, or disclosed through interactions with the transmission medium (Resource), handling tools (Resource), or external influences (Environment/Field). This includes potential distortions or failures in the channel.
3. Execution Environment Integrity: Vulnerabilities emerging during execution due to interactions between the code and the execution system (CPU, cache, memory, OS, frameworks, libraries - Resources and Sub-systems). This includes hardware anomalies, processor vulnerabilities (e.g., side channels), memory management issues, and inadequate filtering/sandboxing within execution frameworks.

This decomposition highlights that vulnerabilities are not isolated to the code artifact but represent potential failures or harmful interactions across the entire system and its lifecycle.

5.3.2 Identifying System Contradictions

Several core Contradictions exist within this system: Code Correctness vs. Execution Reality: Code may be logically correct (static view) but behave insecurely under specific runtime conditions or hardware interactions.

Data Transmission Speed/Efficiency vs. Transmission Integrity/Security: Faster, simpler transmission protocols may offer fewer protections against modification or eavesdropping.

System Performance vs. Security Validation: Rigorous validation and filtering at each stage (transmission, compilation, execution) improves security but may introduce performance overhead (Contradiction resolved often by compromise, leading to vulnerabilities).

Interoperability/Flexibility vs. Controlled Execution: Allowing code to interact with diverse system components and be easily modified (Resource interaction) increases flexibility but creates attack surfaces compared to a highly restricted, monolithic system (Mono-Bi-Poly principle consideration).

5.3.3 Applying TRIZ Principles for Systemic Defense Concepts

The brainstorming explored using TRIZ principles to conceive defenses addressing these systemic issues, moving beyond traditional static/dynamic analysis:

Principle 32 (Changing Color / Optical Properties) & Principle 28 (Mechanics Substitution) -> Conceptual "Analog Filter":

Idea: Exploit the difference between the digital (immaterial, easily manipulated) and analog/physical (material, less directly attackable) domains. Code exists digitally, but its transmission or intermediate processing could involve an analog conversion step.

Mechanism: Convert digital code/instructions into an analog format (e.g., visual representation like text on paper, optical signals - hence "photon filters" metaphorically). Before reconverting to digital for execution, apply a physical or analog filter that only allows "safe" patterns, symbols, or sequences, inherently blocking patterns associated with known vulnerabilities (similar to filtering offensive words). This uses the analog domain as an Intermediary (Principle 24).

TRIZ Logic: This attempts to resolve the Contradiction between needing to transmit executable information vs. preventing harmful instructions from being transmitted/executed. It leverages the physical properties of the analog domain (Resource) as a control mechanism. The filter acts as a preliminary check (Principle 9/10). The conversion process implicitly uses MATCEM thinking by modeling the interaction between digital information (S1), the analog representation (S2), and the conversion/filtering process (Field).

Principle 1 (Segmentation) & Principle 7 (Nested Doll) -> Layered Security Architecture:

Idea: Create distinct security zones or layers within the execution environment (e.g., within the processor, cache hierarchy, or OS).

Mechanism: Critical, highly sensitive code/data is restricted to the most protected inner layers ("sectors"). Less sensitive operations occur in outer layers with potentially less rigorous filtering (but still potentially using mechanisms like the "Analog Filter" between layers). Penetrating each subsequent layer requires bypassing additional controls. This creates a multi-level secure structure ("tower" metaphor).

TRIZ Logic: This uses Segmentation to divide the system and Nested Doll to create hierarchical protection levels, limiting the impact of a breach in outer layers.

Other Mentioned Principles:

Convert Harm to Benefit / Blessing in Disguise - Principle 22: While not fully developed in the notes, this could relate to designing systems where attempted attacks trigger defenses or provide useful information.

System Transitions: Analyzing the problem at the level of the function, the application, the OS, the hardware (sub-systems and super-systems) provides different perspectives and potential solutions, as done in the system decomposition (3.6.1).

Mono-Bi-Poly: Considering transitions from single components/functions to multiple interacting ones, highlighting the increased complexity and potential for interaction vulnerabilities versus the limitations of monolithic designs.

5.3.4 Implications for Vulnerability Management

This broader TRIZ analysis suggests that a comprehensive approach to code security must consider the entire lifecycle. Vulnerability detection should ideally encompass not just static code analysis but also analysis of transmission protocols, storage mechanisms, compilation/deployment processes, and the execution environment configuration. Furthermore, defensive strategies could explore unconventional, TRIZ-inspired mechanisms like the conceptual "Analog Filter" or enhanced system segmentation to build more resilient systems where vulnerabilities in one component are less likely to compromise the whole. This systemic view informs the scope and potential future directions of the research presented in this thesis.

5.3.5 Validity and Novelty of Systemic Approach

Applying TRIZ system thinking (System Operator, Sub/Super-systems, Function/Resource analysis) to the entire code lifecycle provided a valuable conceptual framework. The decomposition into Static Code, Transmission/Storage, and Execution Environment aligns with standard security domains but gains structure through the TRIZ lens.

The conceptual defense mechanisms ("Analog Filter", Layered Security) derived using TRIZ principles like Intermediary, Mechanics Substitution, Segmentation, Nested Doll, while abstract, demonstrate TRIZ's potential for generating unconventional security ideas. Finding real-world parallels in high-security technologies like Data Diodes and Air Gaps (Section 4.4.5) adds validity to the underlying principles (especially Segmentation and one-way flow). The novelty lies in the TRIZ-driven conceptualization and the idea of potentially integrating such filtering mechanisms within computational components rather than solely as external appliances.

5.3.6 Addressing Broader Research Questions

This systemic analysis directly addresses the research goal of using TRIZ tools to propose ways to avoid executing vulnerable code. By identifying weaknesses beyond static code (in transmission, execution) and proposing defenses based on physical separation, domain transformation (analog filter), and structural layering, the TRIZ analysis provides conceptual pathways towards building systems that are inherently more resili-

ent to code vulnerabilities, rather than relying solely on detecting flaws in the code itself. This aligns with secure system design principles.

5.3.7 Conceptual Results: Systemic TRIZ Analysis of Code Lifecycle Vulnerabilities

Beyond the empirical testing of LLM prompts for static code analysis, TRIZ principles were applied conceptually to analyze the broader system encompassing the entire lifecycle of software code, from creation through storage, transmission, and execution. This systemic analysis, documented in brainstorming sessions, aimed to identify vulnerabilities arising from interactions within this lifecycle and to generate novel defensive concepts using TRIZ tools.

System Decomposition and Problem Scope Expansion

The initial system analysis framed the core problem: software code, often considered correct in its static form, becomes vulnerable through interaction with its environment during its lifecycle. Using TRIZ system thinking (akin to the System Operator or multi-screen analysis), the problem space was decomposed beyond just the static code itself.

Static Code: The inherent logic and structure of the code.

Transmission & Storage: Processes involving the transfer of code between storage and execution/modification points. Vulnerabilities here relate to potential modification, corruption, or unauthorized access during transit or rest, involving interactions between the code (Resource), storage/transmission media (Resource), and handling tools/protocols (Field, Resource).

Execution Environment: The dynamic execution of code involving hardware (CPU, cache, memory - Resources), operating systems, libraries, and frameworks (Sub-systems, Resources). Vulnerabilities arise from flawed interactions, hardware anomalies, insecure configurations, or insufficient runtime filtering/sandboxing.

This decomposition led to the realization that vulnerabilities are systemic, emerging not only from code logic but also from the processes and environments interacting with the code throughout its lifecycle.

Conceptual Defense Mechanisms Derived from TRIZ

Applying TRIZ principles to this systemic view generated conceptual ideas for defense mechanisms operating beyond traditional code scanning.

The "Analog Filter" Concept

Core Idea: Inspired by leveraging different physical domains (Principle 28: Mechanics Substitution) and inserting an Intermediary (Principle 24), this concept proposes con-

verting digital code or instructions into a temporary analog/physical format (e.g., visually, optically) before critical processing or execution.

Mechanism: An analog "filter" would then inspect this representation, allowing only patterns, symbols, or sequences deemed safe (analogous to filtering prohibited words from text recognized via OCR) before reconversion to digital. The key is that the potentially harmful digital input does not directly interact with the execution environment; it must pass through the analog intermediary and filter. This acts as a form of Preliminary Anti-Action (Principle 9).

Potential Application: Conceptually, such filters could be placed between system components (e.g., memory and CPU, network interface and application logic) to sanitize instruction or data flow based on physical/analog pattern recognition rather than purely digital logic, potentially resisting digital bypass techniques.

Layered Security Architecture ("Photon Filters" / "Tower")

Core Idea: Applying Segmentation (Principle 1) and Nested Doll (Principle 7), this concept envisions creating distinct, hierarchical security zones within hardware or software execution environments.

Mechanism: Highly sensitive code and operations would be confined to inner, strongly protected layers, potentially using mechanisms like the "Analog Filter" or other strict controls between layers. Less sensitive operations occur in outer layers. Compromising an outer layer does not automatically grant access to inner layers.

Potential Application: This could manifest as hardware-enforced secure enclaves with filtered communication channels, or multi-level operating system architectures where inter-layer communication is strictly mediated and validated.

Underlying TRIZ Principles in Systemic Analysis

The brainstorming process implicitly or explicitly drew upon several TRIZ tools beyond those mentioned above.

System Transitions (Super-system/Sub-system Analysis): Examining the problem at different levels (function, application, OS, hardware) was crucial for the decomposition in 4.4.1.

Mono-Bi-Poly: Considering the evolution from simple components to complex interacting systems helped identify interaction points as potential sources of vulnerability.

Convert Harm to Benefit (Principle 22): Briefly considered as a potential avenue for future exploration (e.g., using attack attempts to trigger enhanced defenses).

MATCEM / Su-Field Analysis: While not explicitly detailed in the notes provided, the focus on interactions between components (code, hardware, transmission medium) aligns with the core ideas of Su-Field modeling.

Implications of Systemic Analysis

This conceptual application of TRIZ yielded insights beyond the scope of the LLM prompt experiments: It reinforces that code vulnerabilities are often systemic issues, not just localized bugs. It generated novel, albeit conceptual, defense strategies by applying TRIZ principles like Intermediary and Segmentation in unconventional ways (e.g., the Analog Filter). It highlights the potential for TRIZ to contribute not only to vulnerability detection but also to the design of more inherently secure systems by proactively addressing contradictions and leveraging system resources differently.

These conceptual results complement the empirical findings from the prompt evaluations, demonstrating the broader applicability of the TRIZ methodology to the thesis topic concerning vulnerabilities across the code lifecycle.

Connection to Existing Security Technologies

Further research revealed that while the specific conceptual mechanisms like the intra-processor "Analog Filter" derived from TRIZ brainstorming may be novel, the underlying principles resonate strongly with existing, highly specialized security technologies designed for enforcing data separation and one-way flow. These technologies provide real-world validation for the directions suggested by the TRIZ analysis.

Data Diodes: These hardware devices enforce strictly unidirectional data transfer, typically using fiber optics with physically separate transmit/receive paths. This directly implements the principle of preventing harmful feedback or injection by eliminating the return channel entirely. They are commonly used in high-security environments like military networks and industrial control systems (SCADA) [41] and represent a physical realization of Segmentation (Principle 1) and using a hardware Intermediary (Principle 24) to resolve the connectivity vs. security Contradiction.

Air Gaps: Representing the most extreme form of Segmentation, air-gapped systems maintain complete physical network isolation. Data transfer relies on manual movement of physical media, requiring strict procedures for scanning and validation (Preliminary Action/Anti-Action - Principles 9/10) [42]. This resolves the security contradiction by severely restricting the "transmission system" itself.

Optical Fiber Isolators: Similar to data diodes in using light for transmission, these devices prevent electrical signal propagation, eliminating certain classes of electrical interference or attack vectors. This leverages a change in the physical domain (Principle 28: Mechanics Substitution) as a security control.

Unidirectional Gateways: These are often software or hardware-software systems that allow data flow in one direction but include content filtering and protocol breaking (Intermediary function). They inspect traffic, removing potentially malicious elements (scripts, executables) before forwarding allowed data types. This combines one-way flow with Preliminary Anti-Action (filtering).

Strict Protocol Filtering & Formal Verification: Software-based approaches like stringent protocol filters (allowing only plain text) or formal verification methods (mathematically proving data properties, e.g., in seL4 [43]) act as Preliminary Anti-Action mechanisms, attempting to ensure only "safe" data according to predefined rules passes through a checkpoint.

These existing solutions validate the core idea emerging from the TRIZ analysis: resolving the contradiction between data utility and security often involves physical or logical segmentation, introducing intermediaries, changing the transmission medium/domain, and implementing preliminary filtering/validation.

A key observation, however, is that most of these robust, high-assurance solutions (especially data diodes and air gaps) are implemented as external devices or architectural configurations. The TRIZ brainstorming pushed further, conceptualizing the integration of such filtering principles *within* core processing components (like CPUs) using hypothetical mechanisms ("photon filters," "analog filters"). This suggests a potential avenue for future research: applying TRIZ to design inherently more compartmentalized and self-filtering processing architectures, moving beyond reliance on external security appliances. The analysis reinforces the idea that reducing the uncontrolled "freedom" of data and code flow within a system is paramount for enhancing security.

5.3.8 Conceptual Claims Derived from Systemic TRIZ Analysis

To further formalize and articulate the core conceptual defense strategies emerging from the systemic TRIZ analysis (detailed in Sections 4.4.2 and 4.4.5), their underlying principles can be expressed using patent claim language.

1. Method for Secure Data Transmission Between Components of a Computing System

A way for secure data transmission between components of a computing system (analogue: systems using security gateways or data diodes), characterized in that, for the purpose of preventing the transfer of potentially harmful digital constructs which are able to illegally bypass standard digital filters, it is implemented by converting an initial digital data stream into an intermediate analog representation (e.g., optical, acoustic), applying an analog sterilization mechanism to said intermediate representation configured to selectively pass patterns corresponding to safe constructs while blocking and/or altering patterns corresponding to unsafe constructs, and/or subsequently reconverting the filtered intermediate analog representation back into a digital data stream before delivery to the target component and/or storage and/or other use.

TRIZ Link Explanation: This method conceptually implements Principle 24 "Intermediary" by introducing the analog representation, Principle 28 "Mechanics Substitution" by using physical properties of the analog signal for filtering, and Principle 9 "Preliminary Anti-Action" through filtering before execution.

2. System for Hierarchical Zonal Data Processing

A system architecture for secure data processing including multi-level variations, characterized in that, for the purpose of providing granular, hierarchical security control and/or limiting the transfer of compromise between components and/or processes of differing sensitivity, it is implemented by defining at least one distinct processing zone arranged hierarchically based on sensitivity levels, establishing physical and/or logical security boundaries between adjacent hierarchical zones, and/or employing mediation mechanisms at said boundaries configured to control data flow, validate data according to zone-specific policies, and/or optionally apply distinct filtering techniques (potentially including the method of claim 1) based on adjacent zone sensitivity.

TRIZ Link Explanation: This architecture conceptually implements Principle 1 "Segmentation" to divide the system into zones and Principle 7 "Nested Doll" to create hierarchical protection levels.

These conceptual claims encapsulate the core innovative ideas derived from applying TRIZ system thinking to the problem of code security across its lifecycle, suggesting potential avenues for designing inherently more secure systems.

5.4 Strengths of the Study

Novel Application of TRIZ: Systematically applying TRIZ principles to LLM prompt engineering for vulnerability detection and to the conceptual analysis of the code lifecycle represents a novel interdisciplinary approach, bridging systematic innovation with AI-driven cybersecurity.

Empirical Refinement and Iterative Development: The study employed an iterative development process for prompt engineering, quantitatively evaluating multiple prompt versions (V8-V11, Table 2) using the DeepSeek LLM. This empirical grounding for prompt design and selection is a methodological strength.

Systematic Ablation Study: A systematic ablation study was conducted on the baseline TRIZ prompt (V9) with the DeepSeek LLM (Table 3), offering valuable insights into the specific contribution of different TRIZ-derived prompt components. Key variations from this ablation (Var 4 and Var 5) were subsequently tested across Llama and Mistral (Tables 5 and 6), providing broader evidence on the impact of components like indicator structure and mitigation assessment logic across different models.

Use of Real-World Datasets: The empirical evaluations utilized the VulnPatchPairs dataset, derived from real-world open-source projects like QEMU and FFmpeg, and the CVEFixes dataset, which is based on publicly reported CVEs and their fixes. This increases the practical relevance of the findings compared to studies relying purely on synthetic datasets.

Demonstrated Improvement Over Baselines Across Multiple LLMs: The study empirically showed that the proposed TRIZ-based prompting methodology generally achieved significantly better F1-Scores and Recall compared to standard CoT and DFA baselines. This advantage was not only evident with the initial LLM (DeepSeek) but was also largely observed when tested with Llama and Mistral on both datasets, highlighting the robustness of the TRIZ approach, even though absolute performance and reliability varied between the LLMs (Tables 5 and 6).

Cross-Dataset and Cross-LLM Validation: The research included cross-dataset validation, evaluating the prompts on two distinct real-world datasets (VulnPatchPairs and CVEFixes). Furthermore, key prompts were subjected to cross-LLM validation using DeepSeek, Llama, and Mistral. This dual level of validation increases confidence in the generalizability of the core findings regarding the TRIZ approach's benefits over simpler baselines and provides insights into LLM-specific behaviors.

Combined Methodological Approach: The integration of quantitative empirical prompt evaluation (including iterative development, ablation, and cross-model/dataset testing) with qualitative conceptual systemic TRIZ analysis provides a richer and more comprehensive understanding of TRIZ's applicability to code security, addressing both specific detection tasks and broader strategic considerations.

5.5 Limitations of the Study

Context Limitation: The primary limitation of the empirical study is the analysis of single functions without broader inter-procedural or application-level context. This inherently restricts the ability of any LLM to definitively assess exploitability and the effectiveness of external mitigations, impacting both FN and FP rates.

LLM Specificity and Performance Variability: While the study was expanded to include three general-purpose LLMs, the results clearly indicate that performance and behavior can vary significantly between different models. Findings specific to one LLM may not directly transfer to others, and the optimal prompt structure or approach might also differ. The observed variations in F1-score, Recall, Precision, FPR, and error rates across DeepSeek, Llama, and Mistral for the same prompts and datasets (Tables 5 and 6) underscore this limitation.

LLM Reliability and Error Rates: A significant limitation observed during the cross-LLM validation was the non-negligible rate of null or error outputs, particularly with Llama and Mistral LLMs. For example, Llama experienced error rates as high as 81% with the CoT prompt on VulnPatchPairs and consistently around 35% on the CVEFixes dataset across all prompts. Mistral also showed considerable error rates (e.g., 28.2% for CoT on VulnPatchPairs, and 10-18% on CVEFixes). These reliability issues can affect

the practical deployment of such models and the robustness of experimental results, as metrics are calculated on successful runs.

Dataset Scope and Bias: While real-world datasets (VulnPatchPairs and CVEFixes) were used, they may have inherent biases regarding the types or complexity of vulnerabilities, programming languages (VulnPatchPairs focused on C, though CVEFixes is multi-language), or project domains represented. This potentially limits the generalizability of the findings to all vulnerability classes or coding contexts.

Dataset Variability: Performance and the relative ranking of prompts and LLMs were observed to differ between the VulnPatchPairs and CVEFixes datasets across all tested LLMs. This indicates that results are sensitive to dataset characteristics, making it challenging to claim universal superiority for any single approach without broader testing.

Metric Limitations: While standard metrics like F1-score, Recall, Precision, and FPR were used, they may not capture the full picture of practical usability. For instance, a high F1-score achieved with a very high FPR (as seen with Var 4 across LLMs) might be unacceptable in many real-world scenarios. Qualitative assessment of the LLM's reasoning was also limited in this study.

Baseline Comparison Nuance: The comparison to CoT and DFA baselines, adapted from literature, is subject to nuances. Variations in CoT and DFA implementation exist, and the performance of these baselines also varied significantly depending on the LLM used. Furthermore, the high error rates of the CoT and DFA baselines with some LLMs (e.g., Llama and Mistral) impact the direct reliability of metric comparisons for those specific combinations.

Stochasticity of LLMs: Although mitigated by using a low temperature (0.1) and averaging results from multiple runs where specified (e.g., 500 runs per prompt for DeepSeek in initial tests), LLM outputs can still exhibit some variability.

Conceptual Nature of Systemic Findings: The systemic TRIZ analysis and the proposed defense concepts (Section 5.3.7 Conceptual Results: Systemic TRIZ Analysis of Code Lifecycle Vulnerabilities) are primarily conceptual and require substantial further research to assess their practical feasibility, implementation details, and effectiveness.

5.6 Synthesis of Findings

This thesis successfully explored the application of TRIZ principles to the domain of LLM-based code vulnerability detection and to conceptual systemic security analysis. The empirical results demonstrated that incorporating structured analytical steps derived from TRIZ concepts, particularly Function and Resource analysis, into LLM prompts can enhance vulnerability detection. With the DeepSeek LLM, this structured TRIZ approach led to significantly improved performance (notably in Recall and F1-Score)

compared to standard CoT and DFA baselines on both datasets tested. However, when validated across Llama and Mistral LLMs, the advantage of TRIZ prompts over these baselines was not consistently replicated; in several instances, CoT and DFA baselines matched or exceeded the F1-Score and Recall of the tested TRIZ prompts for these LLMs. This highlights that while the structured methodology holds promise, its relative effectiveness, optimal configuration, and reliability are highly dependent on the specific LLM and dataset characteristics.

Achieving an optimal balance between Recall and Precision for single-function analysis remains a significant challenge due to the inherent context limitations of analyzing isolated code snippets, a factor that affected all tested LLMs. The ablation studies conducted primarily with DeepSeek were crucial in identifying key TRIZ-derived prompt components, such as detailed framing and justification requirements. Subsequent testing of select variations across all three LLMs indicated that certain prompt modifications had consistent effects: simplifying mitigation checks (as in Var 5) generally offered stable, balanced performance relative to the baseline TRIZ prompt V9 across different LLMs. Adjusting indicator specificity by removing detailed structures (as in Var 4) reliably increased Recall and often F1-scores for DeepSeek and Llama, albeit with a corresponding significant rise in FPR; for Mistral, this variation did not surpass baseline F1-scores. These findings emphasize that the "best" specific prompt configuration is often LLM and dataset dependent, necessitating tailored approaches rather than a one-size-fits-all solution.

Furthermore, the conceptual application of TRIZ provided a valuable framework for analyzing vulnerabilities across the entire code lifecycle (transmission, storage, execution) and generated novel ideas for systemic defenses. This confirms the versatility of TRIZ as a tool for cybersecurity problem-solving that extends beyond prompt engineering for static code analysis.

While LLMs guided by TRIZ-enhanced prompts show considerable promise as aids in the complex task of security analysis—particularly with LLMs like DeepSeek where clear advantages in detection rates over simpler prompting methods were observed—they currently complement rather than replace traditional static analysis tools, dynamic testing, and expert human review. The significant performance variability observed across different LLMs, the persistent context limitations, and the high error rates encountered with some models (notably Llama and Mistral) underscore the ongoing need for careful LLM selection, rigorous validation, and thoughtful integration of LLM-based tools into broader security workflows.

6 Conclusion

This thesis investigated the application of TRIZ principles to the multifaceted challenge of identifying security vulnerabilities in software code. The research explored two primary avenues: the engineering of structured prompts for LLMs to enhance static code analysis, and the conceptual application of TRIZ system thinking to analyze the broader code security lifecycle, including its transmission, storage, and execution. A central aim was to determine if TRIZ could provide a robust, structured methodology to improve automated vulnerability detection via general-purpose LLMs and to offer novel perspectives on mitigating software security risks.

The comprehensive empirical findings revealed a nuanced picture. While the structured TRIZ framework demonstrated potential for enhancing LLM-driven vulnerability detection, its effectiveness relative to standard CoT and DFA baselines varied considerably across the tested LLMs (DeepSeek, Llama, and Mistral) and datasets (VulnPatchPairs and CVEFixes). With the DeepSeek LLM, TRIZ-based prompts generally led to significantly improved F1-scores and Recall compared to the baselines. However, for Llama and Mistral, the CoT and DFA baselines often achieved comparable or superior F1-scores and Recall, indicating that the benefits of the specific TRIZ prompts evaluated are LLM-dependent. The research also highlighted persistent challenges in optimizing performance metrics due to the inherent Recall and Precision trade-off, context limitations of single-function analysis, and significant variations in LLM capabilities, error rates, and sensitivities to prompt structures.

Furthermore, while the conceptual application of TRIZ yielded promising systemic defense ideas, translating these into practically implemented and validated security solutions remains an area for future work.

6.1 Contributions

Comprehensive Empirical Evaluation and Validation: The study provides extensive empirical evidence on the effectiveness and challenges of using TRIZ-based prompts for vulnerability detection. This multi-faceted evaluation includes: (1) Iterative prompt development and refinement, primarily using the DeepSeek LLM on the VulnPatchPairs dataset. (2) A systematic ablation study with DeepSeek to identify the impact of specific TRIZ-derived prompt components. (3) Cross-dataset validation of prompts on both VulnPatchPairs and CVEFixes datasets to assess generalizability. (4) Crucially, cross-LLM

validation of key TRIZ prompts (V9, Var 4, Var 5) and baselines (CoT, DFA) using three different LLMs (DeepSeek, Llama, Mistral) on both datasets.

This comprehensive testing revealed that: (1) With the DeepSeek LLM, TRIZ-based prompts consistently achieved significantly improved F1-scores and Recall compared to standard CoT and DFA baselines across both datasets. (2) With Llama and Mistral LLMs, the relative performance was more varied. While TRIZ prompts sometimes offered advantages over CoT, the DFA and CoT baselines frequently achieved comparable or superior F1-scores and Recall, particularly with Mistral. These comparisons were also attended by considerations of high False Positive Rates for some high-performing baselines (e.g., Llama with DFA) and significant operational error rates for Llama and Mistral across various prompts.

This extensive validation, therefore, contributes by not only demonstrating the potential of the TRIZ methodology with certain LLMs but also by highlighting the critical LLM-specific nature of prompt effectiveness and reliability in the cybersecurity domain.

6.2 Future Work

6.2.1 Implications for Practitioners

Employ Structured LLM Queries: Practitioners using general-purpose LLMs for code review can benefit significantly from structuring their prompts systematically. Drawing inspiration from TRIZ Function Analysis and Resource Analysis – by asking the LLM specific questions about inputs, outputs, resource handling, data flow, and the presence or absence of expected security checks – is demonstrably more effective than using generic "find vulnerabilities" prompts. This structured approach was shown to improve detection capabilities across multiple LLMs.

Strategic Prompt and LLM Selection Based on Need and Risk Tolerance: The study clearly shows that different prompt structures and LLMs yield varying balances of Recall, Precision, and FPR.

For scenarios requiring high Recall where subsequent human review can filter noise (e.g., deep-dive security assessments), prompts like Variation 4 (which removes specific indicator structures) consistently provided higher F1-scores and Recall across DeepSeek, Llama, and Mistral, albeit with substantially higher FPRs.

For quicker checks, pre-commit hooks, or situations where minimizing false alarms is critical, practitioners should opt for prompts and LLM combinations that offer better Precision and lower FPR. While a universally "high-precision" TRIZ prompt across all LLMs wasn't identified as clearly as with DeepSeek (where Var 2 was promising), variations like Var 5 (simplified mitigation) offered a more balanced performance that was relatively stable across the tested LLMs.

Practitioners must consider the specific LLM's characteristics. For example, while Llama showed high recall with some prompts, its high error rate and FPR in this study make it less reliable for practical, consistent use without significant result validation. DeepSeek offered more stable, albeit sometimes more conservative, performance.

Critically Acknowledge LLM Limitations and Variability: It is crucial for practitioners to remain aware of the inherent limitations of current general-purpose LLMs when analyzing isolated code snippets.

Context Window: The single-function analysis limits the LLM's ability to assess inter-procedural vulnerabilities or the true impact of external mitigations. Findings, especially for complex flaws, should be treated as indicators requiring further manual investigation or analysis with tools providing broader program context.

LLM Inconsistency and Reliability: Performance (accuracy, F1-score, FPR) and reliability (error rates in generating responses) vary significantly between different LLMs (as seen between DeepSeek, Llama, and Mistral). A prompt effective with one LLM may perform differently with another. The high error rates observed with some LLMs in this study are a serious practical concern. Practitioners should therefore pilot and select LLM models carefully for their specific environment and use case.

Relying solely on current general-purpose LLM prompts for definitive vulnerability assessment is not advisable.

Leverage TRIZ-Inspired Prompts for Enhanced Initial Scanning: Despite the limitations, well-structured TRIZ-inspired prompts (such as Var 5 or a carefully chosen V9 variant depending on the selected LLM and tolerance for FPR) generally offer a more effective way to leverage general-purpose LLMs for an initial phase of vulnerability scanning compared to basic CoT or DFA prompts, particularly for improving detection rates (Recall and F1-Score). However, the choice of LLM and the need for careful review of the LLM’s outputs, given the potential for errors and false positives/negatives, cannot be overstated.

6.2.2 Implications for Researchers

Enhancing Prompts for Broader Contextual Analysis: A primary direction is to develop and evaluate TRIZ-based prompts designed for analyzing interactions between multiple functions or components. This could involve incorporating RAG techniques to dynamically provide necessary context (e.g., caller/callee functions, related data structures, project-specific security policies) to the LLM, potentially overcoming some of the limitations of single-function analysis observed across all tested LLMs.

Refining and Optimizing TRIZ Prompt Components for Different LLMs: Further research is needed to investigate the optimal structure for vulnerability indicators and mitigation assessment logic within TRIZ prompts.

The performance of Variation 4 (ablating specific indicator structures) suggests that while it boosts Recall and F1-score across LLMs, its high FPR needs mitigation, perhaps through post-processing or refined secondary checks.

Variation 5 (simplified boolean mitigation) showed promise as a stable performer across LLMs and could be a basis for further refinement.

The varying impact of components like explicit Contradiction Analysis (e.g., Var 2 with DeepSeek) across datasets, and the untested performance of several ablation variants (Var 1, Var 2, Var 3, Var 6) on Llama and Mistral, warrants further LLM-specific investigation and tuning of prompt components.

Exploring Multi-Turn TRIZ-based Dialogues: Investigating the use of TRIZ principles like Dynamicity and Feedback in multi-turn conversational prompting frameworks is a promising area. Such an approach would allow the LLM's intermediate findings to dynamically guide subsequent analytical steps, potentially leading to more adaptive and deeper analysis than single-shot prompting.

Comprehensive Cross-LLM and Cross-Dataset Validation and Benchmarking: While this study initiated cross-LLM validation for selected prompts, more extensive research is needed.

Validate the full suite of TRIZ prompt variations (including all V9 ablations) across a wider range of LLMs (both proprietary and leading open-source models) and diverse vulnerability datasets (e.g., Juliet, OWASP Benchmark, other CWE-specific datasets) covering multiple programming languages.

Systematically investigate and characterize the reasons behind performance differences and error rate variations observed across LLMs like DeepSeek, Llama, and Mistral with specific prompts.

Establish robust benchmarks for TRIZ-inspired prompting techniques in cybersecurity.

Understanding and Mitigating LLM Errors and Unreliability: The high error rates encountered with some LLMs (Llama and Mistral in Tables 5 and 6) are a critical area for research. Future work should focus on: (1) Identifying the causes of these null/error responses (e.g., prompt complexity, specific code constructs, API limitations, model-specific failure modes). (2) Developing strategies to improve prompt robustness or LLM handling to minimize such failures.

Deepening the Understanding of Dataset Impact on LLM Performance: The observed variability in LLM and prompt performance between the VulnPatchPairs and CVEFixes datasets needs further exploration. Research could focus on identifying dataset characteristics (e.g., code complexity, vulnerability type distribution, code style, function length) that most significantly influence LLM analytical capabilities and how these interact with specific LLM architectures.

Investigating the Feasibility of Conceptual TRIZ-inspired Defenses: The systemic defense concepts proposed (e.g., "Analog Filter," layered execution environments) require feasibility studies. This could involve theoretical modeling, simulation, or proof-of-concept implementations to assess their potential and practicality in real-world systems.

Applying TRIZ for Exploit Path Analysis and Generation: Future research could explore the application of TRIZ principles to model potential exploit paths or to analyze the systemic contradictions that make particular exploits possible, potentially aiding in proactive defense design or penetration testing.

Fine-tuning LLMs with TRIZ-Structured Data: Explore fine-tuning LLMs using datasets where the vulnerabilities and secure coding patterns are explicitly annotated or explained using the structured analytical framework of TRIZ (e.g., Function Analysis, Resource Analysis). This could potentially imbue models with a more systematic "understanding" of vulnerability mechanisms.

6.3 Concluding Remarks

This research successfully demonstrated that the systematic problem-solving methodology of TRIZ can be effectively adapted and applied to the modern challenge of enhancing LLM-driven code vulnerability detection. The empirical results established that incorporating structured analytical steps derived from TRIZ concepts into LLM prompts can yield substantial improvements in vulnerability detection effectiveness (notably F1-Score and Recall) compared to standard CoT and DFA baselines, particularly when applied with certain LLMs. This advantage was clearly observed with the DeepSeek LLM across multiple datasets (VulnPatchPairs and CVEFixes). However, with Llama and Mistral, the performance of TRIZ prompts relative to the baselines was more varied, with baselines often proving competitive or superior in F1-score and Recall. These LLM-specific behaviors, alongside observed nuances in reliability and error rates, underscore the complexity of generalizing prompt effectiveness across different models.

While context limitations inherent in static, single-function analysis continue to pose hurdles to achieving perfect accuracy with any of the tested approaches, the TRIZ framework provides a valuable and structured method for guiding LLM reasoning and for systematically exploring the critical Recall/Precision trade-off. The ablation study was instrumental in pinpointing key elements of the TRIZ-based prompts that contribute to performance with DeepSeek, such as detailed analytical framing and justification generation. It also identified promising prompt variations whose impacts (like adjusting indicator specificity or simplifying mitigation checks) showed certain consistencies but also LLM and dataset dependencies when tested more broadly.

Furthermore, the conceptual application of TRIZ generated novel perspectives on systemic defenses across the code lifecycle, confirming the methodology's broader relev-

ance to cybersecurity beyond prompt engineering. This work lays a foundation for future research aimed at further integrating TRIZ's structured innovation techniques with the rapidly evolving capabilities of Large Language Models. The goal remains to create more effective, reliable, and context-aware tools for building and maintaining secure software systems, a journey that underscores both the potential of such interdisciplinary approaches and the ongoing critical need for rigorous, model-specific validation and refinement in the face of diverse technological landscapes.

References

[1] I. M. Ilevbare, D. Probert, and R. Phaal, "A review of TRIZ, and its benefits and challenges in practice," Technovation, vol. 33, no. 2–3, pp. 30–37, Feb.–Mar. 2013. doi: 10.1016/j.technovation.2012.11.003.

[1] L. Shulyak and S. Rodman, 40 principles: TRIZ Keys to Technical Innovation. Technical Innovation Center, Inc., 2002.

[2] *D. Kluender, "TRIZ for software architecture," Procedia Engineering, vol. 9, pp. 708–713, Jan. 2011, doi: 10.1016/j.proeng.2011.03.159.*

[3] D. Mann, "TRIZ For Software?" 2004. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d8c51f320d658289f02a02f5a5c7045ad9e7524d (accessed Apr. 15, 2025).

[4] K. C. Rea, "Using TRIZ in Computer Science - Concurrency" Metodolog.ru TRIZ Journal. [Online]. Available: https://www.metodolog.ru/triz-journal/archives/1999/08/d/index.htm (accessed Apr. 15, 2025).

[5] S. Jiang, W. Li, Y. Qian, Y. Zhang, and J. Luo, "AutoTRIZ: Automating Engineering Innovation with TRIZ and Large Language Models," arXiv.org, Mar. 13, 2024. https://arxiv.org/abs/2403.13002 (accessed Apr. 15, 2025).

[6] L. Chen, Y. Song, S. Ding, L. Sun, P. Childs, and H. Zuo, "TRIZ-GPT: An LLM-augmented method for problem-solving," arXiv.org, Aug. 12, 2024. https://arxiv.org/abs/2408.05897 (accessed Apr. 15, 2025).

[7] J. White et al., "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT," arXiv.org, Feb. 21, 2023. https://arxiv.org/abs/2302.11382 (accessed Apr. 15, 2025).

[8] *K. Tamberg and H. Bahsi, "Harnessing large language models for software vulnerability Detection: A Comprehensive Benchmarking study," IEEE Access, p. 1, Jan. 2025, doi: 10.1109/access.2025.3541146.*

[9] A. A. Mahyari, "Harnessing the power of LLMs in source code vulnerability detection," arXiv.org, Aug. 07, 2024. https://arxiv.org/abs/2408.03489 (accessed Apr. 15, 2025).

[10] I. Ceka, F. Qiao, A. Dey, A. Valecha, G. Kaiser, and B. Ray, "Can LLM prompting serve as a proxy for static analysis in vulnerability detection," arXiv.org, Dec. 16, 2024. https://arxiv.org/abs/2412.12039 (accessed Apr. 15, 2025).

[11] *C. D. Xuan, D. B. Quang, and V. D. Quang, "Large language models based vulnerability detection: How does it enhance performance?," International Journal of Information Security, vol. 24, no. 1, Jan. 2025, doi: 10.1007/s10207-025-00983-8.*

[12] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, and H. Cai, "Chain-of-Thought prompting of large language models for discovering and fixing software vulnerabilities," arXiv.org, Feb. 27, 2024. https://arxiv.org/abs/2402.17230 (accessed Apr. 15, 2025).

[13] A. Z. H. Yang, H. Tian, H. Ye, R. Martins, and C. L. Goues, "Security Vulnerability Detection with Multitask Self-Instructed Fine-Tuning of Large Language Models," arXiv.org, Jun. 09, 2024. https://arxiv.org/abs/2406.05892 (accessed Apr. 15, 2025).

[14] *Phadnis, Nikhil & Torkkeli, Marko. (2024). Evaluating the Effectiveness of Generative AI in TRIZ: A Comparative Case Study. 10.1007/978-3-031-75919-2_11.*

[15] "Application of TRIZ framework for resolving security issues in IOT," IEEE Conference Publication | IEEE Xplore, Apr. 01, 2016. https://ieeexplore.ieee.org/document/7813921

[16] *Qi, Ming and Chang-Yi Zou. "A Study of Anti-phishing Strategies Based on TRIZ." 2009 International Conference on Networks Security, Wireless Communications and Trusted Computing 2 (2009): 536-538.*

[17] U. Mishra, "Protecting anti-virus programs from viral attacks," arXiv.org, Jul. 24, 2013. https://arxiv.org/abs/1307.6354 (accessed Apr. 15, 2025).

[18] D. Kluender. (2011). TRIZ for software architecture. Procedia Engineering. 9. 708-713. 10.1016/j.proeng.2011.03.159.

[19] M. Rubin and S. Sysoev. 2017. Applications of TRIZ methods in SW development and design. In Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17). Association for Computing Machinery, New York, NY, USA, Article 18, 1–6. https://doi.org/10.1145/3166094.3166112

[20] "Exploration on prompting LLM with Code-Specific information for vulnerability detection," IEEE Conference Publication | IEEE Xplore, Jul. 07, 2024. https://ieeexplore.ieee.org/document/10664399 (accessed Apr. 15, 2025).

[21] X. Zhou, S. Cao, X. Sun, and D. Lo, “Large Language Model for Vulnerability Detection and Repair: Literature review and the Road Ahead,” arXiv.org, Apr. 03, 2024. https://arxiv.org/abs/2404.02525 (accessed Apr. 15, 2025).

[22] J. Zhang et al., “When LLMs Meet Cybersecurity: A Systematic Literature Review,” arXiv.org, May 06, 2024. https://arxiv.org/abs/2405.03644 (accessed Apr. 15, 2025).

[23] “Papers with Code - Large Language Models and Code Security: A Systematic Literature Review,” Dec. 19, 2024. https://paperswithcode.com/paper/large-language-models-and-code-security-a (accessed Apr. 15, 2025).

[24] “Papers with Code - Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead,” Apr. 03, 2024. https://cs.paperswithcode.com/paper/large-language-model-for-vulnerability-1 (accessed Apr. 15, 2025).

[25] M. Bhatt et al., “Purple llama CyberSecEval: a secure coding benchmark for language models,” arXiv.org, Dec. 07, 2023. https://arxiv.org/abs/2312.04724 (accessed Apr. 15, 2025).

[26] X. Du et al., “VUL-RAG: Enhancing LLM-based vulnerability detection via Knowledge-level RAG,” arXiv.org, Jun. 17, 2024. https://arxiv.org/abs/2406.11147 (accessed Apr. 15, 2025).

[27] Tmylla, “GitHub - tmylla/Awesome-LLM4Cybersecurity: An overview of LLMs for cybersecurity.,” GitHub. https://github.com/tmylla/Awesome-LLM4Cybersecurity (accessed Apr. 15, 2025).

[28] 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.org, Aug. 24, 2024. https://arxiv.org/abs/2408.13597 (accessed Apr. 15, 2025).

[29] N. Tihanyi, M. A. Ferrag, R. Jain, T. Bisztray, and M. Debbah, “CyberMetric: A Benchmark Dataset based on Retrieval-Augmented Generation for Evaluating LLMs in Cybersecurity Knowledge,” arXiv.org, Feb. 12, 2024. https://arxiv.org/abs/2402.07688 (accessed Apr. 15, 2025).

[30] J. He and M. Vechev, “Large language models for code: security hardening and adversarial testing,” Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 1865–1879, Nov. 2023, doi: 10.1145/3576915.3623175.

[31] B. Dolan-Gavitt et al., "LAVA: Large-scale Automated Vulnerability Addition," in NEU SecLab [Online]. Available: https://seclab.nu/static/publications/sp2016lava.pdf (accessed Apr. 15, 2025).

[32] S. Roy, A. Mukherjee, and S. K. Ghosh, "Explainable AI in Vulnerability Detection Using Large Language Models," in AI Techniques for Reliability Prediction and Estimation. Singapore: Springer Nature, 2024, pp. 249-269. doi: 10.1007/978-981-97-5791-6_15.

[33] C. Wang et al., “REEF: a framework for collecting Real-World vulnerabilities and fixes,” arXiv.org, Sep. 15, 2023. https://arxiv.org/abs/2309.08115 (accessed Apr. 15, 2025).

[34] Ase-Reef, “GitHub - ASE-REEF/REEF-DatA,” GitHub. https://github.com/ASE-REEF/REEF-data/tree/main (accessed Apr. 15, 2025).

[35] M. D. Preda, N. Marastoni, and F. Paci, "Next Generation Vulnerability Detection with LLMs," ERCIM News, no. 139, pp. 10-11, Oct. 2024. [Online]. Available: https://ercim-news.ercim.eu/en139/special/next-generation-vulnerability-detection-with-llms (accessed Apr. 15, 2025).

[36] X. Jiang et al., “Investigating large Language models for code vulnerability Detection: An Experimental study,” arXiv.org, Dec. 24, 2024. https://arxiv.org/abs/2412.18260 (accessed Apr. 15, 2025).

[37] S. Sultana, S. Afreen, and N. U. Eisty, “Code Vulnerability Detection: A comparative analysis of emerging large language models,” arXiv.org, Sep. 16, 2024. https://arxiv.org/abs/2409.10490 (accessed Apr. 15, 2025).

[38] N. Risse and M. Böhme, “Uncovering the limits of machine learning for automatic vulnerability detection,” arXiv.org, Jun. 28, 2023. https://arxiv.org/abs/2306.17193 (accessed Apr. 15, 2025).

[39] “NEON-GPT: A Large Language Model-Powered Pipeline for Ontology Learning,” Analytic Computing, Institute for Artificial Intelligence, University of Stuttgart, Germany, 2021, [Online]. Available: https://github.com/andreamust/NEON-GPT (accessed Apr. 15, 2025).

[40] D. Pereira, “What is a Data Diode & How Do Data Diodes Work?,” Owl Cyber Defense, Mar. 19, 2025. https://owlcyberdefense.com/blog/what-is-data-diode-technology-how-does-it-work/ (accessed Apr. 15, 2025).

[41] A. S. Gillis, “air gap (air gapping),” WhatIs, Sep. 13, 2022. https://www.techtarget.com/whatis/definition/air-gapping (accessed Apr. 15, 2025).

[42] G. Klein et al., "seL4: Formal Verification of an OS Kernel," Commun. ACM, vol. 53, no. 6, pp. 107–115, Jun. 2010. doi: 10.1145/1743546.1743574.

[43] G. Bhandari, A. Naseer, and L. Moonen, “CVEfixes Dataset: Automatically Collected Vulnerabilities and Their Fixes from Open-Source Software,” *Zenodo*

(CERN European Organization for Nuclear Research). Jul. 18, 2021. doi: 10.5281/zenodo.4476564.

[44] secureIT-Project, "GitHub - secureIT-project/CVEfixes: CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," *GitHub*. https://github.com/secureIT-project/CVEfixes (accessed May 5, 2025).

[45] T. Li *et al.*, "From crowdsourced data to High-Quality benchmarks: Arena-Hard and BenchBuilder Pipeline," *arXiv.org*, Jun. 17, 2024. https://arxiv.org/abs/2406.11939 (accessed May 5, 2025).

[46] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, "A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions," *Electronics*, vol. 12, no. 6, p. 1333, Mar. 2023, doi: 10.3390/electronics12061333.

[47] "Systematic Literature Review on Security Risks and its Practices in Secure Software Development," *IEEE Journals & Magazine | IEEE Xplore*, 2022. https://ieeexplore.ieee.org/abstract/document/9669954 (accessed May 12, 2025).

[48] F. Jimmy, "Cyber security Vulnerabilities and Remediation Through Cloud Security Tools," *Deleted Journal*, vol. 2, no. 1, pp. 129–171, Apr. 2024, doi: 10.60087/jaigs.v2i1.102.

[49] "Massivizing Computer Systems: a vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems," *IEEE Conference Publication | IEEE Xplore*, Jul. 01, 2018. https://ieeexplore.ieee.org/abstract/document/8416385 (accessed May 12, 2025).

[50] "Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools - NORMA@NCI Library." https://norma.ncirl.ie/5956/ (accessed May 12, 2025).

[51] S. Nandi, "EVALUATING THE EFFECTIVENESS OF SECURITY TESTING TOOLS IN AUTOMATED TESTING," 2024. [Online]. Available: https://trepo.tuni.fi/bitstream/handle/10024/162230/NandiSangeeta.pdf?sequence=2 (accessed May 12, 2025).

[52] "'False negative - that one is going to kill you': Understanding Industry Perspectives of Static Analysis based Security Testing," *IEEE Conference Publication | IEEE Xplore*, May 19, 2024. https://ieeexplore.ieee.org/abstract/document/10646636 (accessed May 12, 2025).

[53] M. A. Khair, "Security-Centric Software Development: Integrating Secure Coding Practices into the Software Development Lifecycle," Feb. 02, 2018. https://hal.science/hal-04565385/ (accessed May 12, 2025).

[54] J. Terninko, A. Zusman, and B. Zlotin, *Systematic innovation : an introduction to TRIZ ; (theory of inventive problem solving)*. 2017. [Online]. Available: https://dx.doi.org/10.4324/9781482279160 (accessed May 12, 2025).

[55] V. Geroimenko, "Key techniques for writing effective prompts," in *SpringerBriefs in computer science*, 2025, pp. 37–83. doi: 10.1007/978-3-031-86206-9_3.

[56] L. Fiorineschi, F. S. Frillici, and F. Rotini, "Enhancing functional decomposition and morphology with TRIZ: Literature review," *Computers in Industry*, vol. 94, pp. 1–15, Sep. 2017, doi: 10.1016/j.compind.2017.09.004.

[57] G. S. Al′tshuller, L. A. Shulyak, and S. Rodman, *40 Principles: TRIZ Keys to Technical Innovation*. 1998. [Online]. Available: https://ci.nii.ac.jp/ncid/BB04190248 (accessed May 12, 2025).

[58] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, A. Cheshkov, and P. Zadorozhny, "Finetuning large language models for vulnerability detection," *arX-iv.org*, Jan. 30, 2024. https://arxiv.org/abs/2401.17010 (accessed May 12, 2025).

[59] Admin, "Understanding the Classification Report in Machine Learning: A Comprehensive guide," *Data Science/ML/BI Consulting Company*, Aug. 12, 2024. https://datamiracle.com/2024/08/12/machine-learning-classification-reporit/ (accessed May 12, 2025).

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Anton Matskevitš

1 Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis Application of TRIZ to Find Vulnerabilities in Code, Methods of its Transmission, Storage and Execution, supervised by Hayretdin Bahşi

1.1 to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;

1.2 to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.

2 I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.

3 I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

16.05.2025

[1] The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – CoT Prompt

As a security researcher, you are tasked with reporting security vulnerabilities. You are currently looking at the following code:
‘‘‘

{code}
‘‘‘

You are expected to work step-by-step:

1. Identify Weaknesses: Identify which vulnerabilities could be present in the following code.
2. Review User Input Handling: Look for any input sources that are not properly validated or sanitized before being used in unsafe manner. If variable that is passed into unsafe function is not directly influenced by external user input, the vulnerability is not currently present and must not be reported.
3. Analyze Data Flow: Trace the flow of untrusted data to the system command. Ensure that there are no points where user-controlled input can directly influence the command execution.
4. Check for Mitigations: Examine if there are any mitigations in place to prevent command injection, such as input validation, sanitization, or using safer alternatives to executing system commands.
5. Evaluate Conditional Branching: If there’s dead code (that can never be reached), then this part must not be evaluated.
6. Assess Error Handling: Evaluate how errors, if any, are handled.
7. Identify Code Leaking Secrets: Check whether the code contains secrets that should not be public knowledge.
8. Provide verdict (one line for every potential discovered weakness). Keep in mind you must not report vulnerabilities that cannot be currently abused by malicious actors. False positive results must be kept to minimum. The verdict must be in the format: `vulnerable: <YES or NO>`

Appendix 3 – V8 TRIZ prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments.

TASK:
Analyze the provided code snippet(s) and any accompanying context. Your goal is to accurately classify whether an exploitable security vulnerability exists. Strive for a strong balance between detecting vulnerabilities (Recall) and minimizing false alarms (Precision/FPR), aiming for an improved F1-Score compared to previous attempts. Apply the TRIZ-based analytical process rigorously below to make your determination.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially for each potential finding before making the final YES/NO decision):

Initial Identification & Contextual Understanding:

Identify potential vulnerability candidates within the code (e.g., OWASP Top 10 patterns).

Analyze the candidate snippet and its surrounding context (function, class, related files, framework info) to understand purpose, data flow, inputs, outputs, and local operating environment.

TRIZ Problem Framing:

(a) Identify Contradiction(s): State the core technical/physical contradiction(s). Assess the apparent effectiveness of the code's attempt to resolve this contradiction. Are there signs of incomplete resolution or introduced weaknesses?

(b) Function Analysis: Describe the intended useful function(s). Identify potential harmful functions or negative side effects. Note insufficient useful functions (e.g., missing checks). Briefly assess the potential impact and likelihood of harm if these functions are misused or insufficient.

(c) Resource Analysis: List key resources involved. Evaluate the controls. Assess the realistic controllability/accessibility of resources required for a potential exploit in a typical operational context.

Vulnerability Hypothesis (TRIZ Lens):

Based on the TRIZ framing (especially poorly resolved contradictions, impactful harmful/insufficient functions, or realistically misusable resources), formulate a specific hypothesis explaining how this could lead to a security vulnerability.

Consider relevant TRIZ Inventive Principles to refine the potential exploit path or identify overlooked weaknesses.

Contextual & Systemic Evaluation (Mitigation & Exploitability Check):

Evaluate the hypothesis critically:

Data/Control Flow: Is the path plausibly reachable by potentially untrusted data flow?

Mitigation Assessment: Identify any mitigating factors (sanitization, framework protection, config). Critically evaluate the quality, applicability, and likely effectiveness of these mitigations against the specific hypothesized vulnerability. Is the mitigation robust and directly relevant?

Exploitability Realism: Based on resource controllability (from 2c) and mitigation assessment, how realistic is the exploit scenario? Does it require highly improbable conditions?

Binary Classification Decision:

Based on the comprehensive analysis (Steps 2-4), make the binary classification:

vulnerable: YES if:

(A strong TRIZ signal exists (e.g., clearly unresolved/poorly resolved contradiction, high-impact harmful/insufficient function, easily misused resource) AND identified mitigations are assessed as likely ineffective, inapplicable, weak, or absent (Step 4)).

OR (A plausible TRIZ signal exists AND the exploit path appears realistic with no strong, effective mitigation identified).

vulnerable: NO if:

No significant TRIZ weakness signal is identified.

OR (A potential TRIZ weakness is identified BUT there is strong evidence of effective, applicable mitigation OR the required exploit conditions/resource control are deemed highly unrealistic in a typical context).

Justification:

Provide a clear and concise rationale explaining the YES or NO decision.

Reference the core TRIZ weakness identified (contradiction, function, resource).

Crucially, explain the assessment of mitigation effectiveness and exploitability realism (Step 4) and how it informed the final decision according to the logic in Step 5.

OUTPUT FORMAT:

Output only the final classification and its justification:

Classification: vulnerable: YES or vulnerable: NO

Justification: Detailed reasoning based on the TRIZ analysis, focusing on the assessment of the identified weakness, mitigation effectiveness, and exploitability realism.

FINAL INSTRUCTION:

Apply the TRIZ analysis rigorously. Your final output must be strictly binary ("vulnerable: YES" or "vulnerable: NO"). Focus on achieving a better balance (F1-Score) by accurately identifying vulnerabilities where mitigations are weak or absent and exploitability is realistic, while filtering out cases with strong mitigations or highly unrealistic exploit paths. Provide the detailed justification explaining your assessment.

Appendix 4 – V9 TRIZ prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by achieving the best possible balance between detecting true vulnerabilities (Recall) and minimizing false alarms (Precision). Apply the TRIZ-based analytical process rigorously below, focusing on identifying high-confidence vulnerability indicators verifiable within the snippet.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and any provided context to understand the function's purpose, inputs, outputs, and apparent operating environment.

TRIZ Problem Framing & Weakness Identification:
(a) Function Analysis: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences like executing commands, writing arbitrary data) or insufficient useful functions (missing necessary security actions like validation, checks, cleanup - violates Principle 11: Beforehand Cushioning).
(b) Resource Analysis: List key resources manipulated (inputs, memory, files, state variables). Evaluate how they are controlled. Focus on the flow of potentially untrusted resources (especially function arguments, data read within the function).
(c) Contradiction Analysis: Identify any core contradictions the function attempts to resolve (e.g., process input quickly vs. process input safely). Note signs of poor resolution.

Identify High-Confidence Vulnerability Indicators (TRIZ Lens):
Based on Step 2, determine if any of the following high-confidence indicators are clearly present within the function's logic:
Indicator A (Taint Flow to Sink): Clear path for a potentially untrusted input/resource (Resource) to reach a known dangerous sink (e.g., system call, DB query, unescaped web output, file path operation - Harmful Function) without verifiable evidence of standard, effective sanitization/validation/encoding applied to that specific resource before the sink (Insufficient Useful Function, Lack of Principle 9/10).

Indicator B (Resource Mishandling): Clear logical flaw in resource management visible within the function (e.g., pointer arithmetic suggesting buffer overflow, clear potential for double-free/use-after-free based on resource state tracking within the function, leak of sensitive data Resource to logs/output).

Indicator C (Auth/Access Control Flaw): Clear absence of necessary authorization/access control checks within the function before performing an operation identified as security-sensitive based on its name or actions. (Insufficient Useful Function).

Indicator D (Dangerous Function Usage): Direct use of known insecure functions/APIs (e.g., strcpy, sprintf, insecure random number generator for security context) with inputs whose size or content is not demonstrably controlled within the function.

Contextual Evaluation (Mitigation Check for Indicators):

If one or more High-Confidence Indicators (A, B, C, D) were identified in Step 3:

Re-examine the code_snippet specifically for concrete evidence of standard, applicable, and correctly implemented mitigating controls that directly and effectively neutralize the identified indicator(s) within the function's scope. (E.g., Does the function itself apply htmlspecialchars right before echoing user input? Does it use snprintf instead of sprintf with size limits?)

Assess Mitigation Confidence: Strong Internal Mitigation (clear, standard, effective control found within the function for the specific indicator), Weak/No Internal Mitigation (no clear/standard/effective control found within the function for the indicator).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Indicators (A, B, C, or D from Step 3) were clearly identified AND

Step 4 assessed the mitigation within the function as Weak/No Internal Mitigation.

Classify "vulnerable: NO" if:

No High-Confidence Indicators were clearly identified in Step 3.

OR A High-Confidence Indicator was identified, BUT Step 4 found Strong Internal Mitigation effectively neutralizing it within the function's scope.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which High-Confidence Indicator(s) (A, B, C, D) were triggered, explain the supporting TRIZ analysis (taint flow, resource issue, missing checks), and state why internal mitigation was assessed as Weak/No.

If "NO": State that no High-Confidence Indicators were clearly identified OR specify the indicator found and explain the Strong Internal Mitigation identified within the function that neutralizes it.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO

FINAL INSTRUCTION:

Apply the TRIZ analysis rigorously, focusing on identifying the specified High-Confidence Indicators based on evidence within the provided code snippet. Base your final decision on the presence of these indicators coupled with the assessment of verifiable mitigations within the function. Aim for the optimal F1-Score. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 5 – V10 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by maintaining high Recall while reducing the False Positive Rate compared to the previous attempt. Apply the TRIZ-based analytical process rigorously below, focusing on high-confidence indicators and verifiable mitigations (standard or effective non-standard) within the snippet.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and context to understand purpose, inputs, outputs, data flow, and operating environment.

TRIZ Problem Framing & Weakness Identification:
(a) Function Analysis: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences) or insufficient useful functions (missing necessary security checks, validation, cleanup - violates Principle 11: Beforehand Cushioning). Assess potential impact.
(b) Resource Analysis: List key resources (inputs, memory, state). Evaluate controls. Focus on potentially tainted resources and their flow towards sensitive operations.
(c) Contradiction Analysis: Identify relevant security contradictions (e.g., input flexibility vs. safety). Note signs of poor resolution.

Identify High-Confidence Vulnerability Indicators (TRIZ Lens - Refined):
Based on Step 2, determine if any of the following indicators are clearly present and appear non-trivial within the function's logic and context:
Indicator A (Taint Flow): Clear path for a potentially untrusted input/resource (Resource) to reach a known dangerous sink (Harmful Function) AND lack of verifiable, standard sanitization/validation/encoding applied before the sink (Insufficient Useful Function), AND any simple non-standard checks present appear obviously insufficient (e.g., flawed blacklist).
Indicator B (Resource Mishandling): Clear logical flaw demonstrable within the function leading to likely unsafe resource state (e.g., concrete off-by-one in loop interacting with buffer, definite conditional path to double-free/use-after-free, clear leak of sensitive data Resource). Avoid purely speculative issues.

Indicator C (Auth/Access Control Flaw): Clear absence of expected authorization/access control checks within the function before a security-sensitive operation, where context doesn't strongly imply external checks. (Insufficient Useful Function).

Indicator D (Dangerous Function Usage): Direct use of known insecure functions/APIs where the dangerous aspect is influenced by function inputs, and internal controls (like size limits for sprintf) are missing or clearly insufficient.

Contextual Evaluation (Mitigation Check - Refined):

If one or more High-Confidence Indicators (A, B, C, D) were identified in Step 3:

Search the code_snippet specifically for concrete evidence of mitigating controls that directly and effectively neutralize the identified indicator(s) within the function's scope.

Consider both:

Standard Mitigations: Are they present, correctly applied, and relevant?

Non-Standard Mitigations: Is there custom logic (validation, state checks, combined partial checks) that clearly and completely addresses the specific indicator's risk based on the function's internal logic?

Assess Mitigation Confidence: Strong Mitigation (clear evidence of effective standard OR clear/complete non-standard mitigation found), Weak/No Mitigation (no clear/effective mitigation found, or only partial/unclear/bypassable non-standard mitigation found).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Indicators (A, B, C, or D from Step 3) were clearly identified AND

Step 4 assessed the mitigation within the function as Weak/No Mitigation.

Classify "vulnerable: NO" if:

No High-Confidence Indicators were clearly identified in Step 3.

OR A High-Confidence Indicator was identified, BUT Step 4 found Strong Mitigation (either standard or demonstrably effective/complete non-standard) neutralizing it within the function's scope.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which High-Confidence Indicator(s) were triggered, explain the supporting TRIZ analysis, and state why internal mitigation was assessed as Weak/No (lacking standard or clear/complete non-standard controls).

If "NO": State that no High-Confidence Indicators were clearly identified OR specify the indicator found and explain the Strong Mitigation (standard or non-standard) identified within the function and why it is considered effective and applicable in neutralizing the indicator.

OUTPUT FORMAT:

Output only the final classification: vulnerable: YES or vulnerable: NO.

FINAL INSTRUCTION:

Apply the TRIZ analysis rigorously, focusing on identifying the specified High-Confidence Indicators based on clear evidence within the code. Base your final decision on the presence of these indicators coupled with a careful assessment of verifiable mitigations (standard or effective non-standard) within the function. Aim to reduce False Positives from the previous run while retaining as much Recall as possible, improving the F1-Score. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 6 – V11 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to significantly improve Recall compared to the previous attempt, while still leveraging TRIZ to maintain reasonable Precision, thereby improving the F1-Score. Apply the TRIZ-based analytical process rigorously below.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and context to understand purpose, inputs, outputs, data flow, and operating environment.

TRIZ Problem Framing & Weakness Identification:
(a) Function Analysis: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences) or insufficient useful functions (missing necessary security checks, validation, cleanup - violates Principle 11: Beforehand Cushioning). Assess potential impact. Prioritize identifying these functional issues.

(b) Resource Analysis: List key resources (inputs, memory, state). Evaluate controls. Focus on potentially tainted resources and their flow towards sensitive operations (Resource misuse). Prioritize identifying taint flow issues.
(c) Contradiction Analysis: Identify relevant security contradictions (e.g., input flexibility vs. safety). Note signs of poor resolution.

Identify Plausible Vulnerability Indicators (TRIZ Lens):

Based on Step 2 (prioritizing Function and Resource analysis), determine if any plausible indicators of weakness are present within the function's logic:
Indicator A (Taint Flow): A path exists for a potentially untrusted input/resource (Resource) to reach a known dangerous sink (Harmful Function) without clear, standard sanitization/validation applied before the sink (Insufficient Useful Function, Lack of Principle 9/10).

Indicator B (Resource Mishandling): Logical flow suggests potential unsafe resource state (e.g., boundary condition errors in loops handling buffers, potential path to double-free/use-after-free, leak of sensitive data Resource).

Indicator C (Auth/Access Control Flaw): Absence of apparent authorization/access control checks within the function before a potentially sensitive operation (Insufficient Useful Function).

Indicator D (Dangerous Function Usage): Use of known insecure functions/APIs where inputs are not clearly and safely constrained within the function.

Contextual Evaluation (Mitigation Check - Stricter Assessment):

If one or more Plausible Indicators (A, B, C, D) were identified in Step 3:

Search the code_snippet specifically for concrete evidence of mitigating controls that directly, effectively, and completely neutralize the identified indicator(s) within the function's scope.

Consider both:

Standard Mitigations: Are they present, correctly applied, and fully relevant?

Non-Standard Mitigations: Is there custom logic? Assess critically: Does it unambiguously and completely address the specific risk identified? Partial or potentially bypassable custom logic does not count as strong mitigation.

Assess Mitigation Confidence: Strong Mitigation (clear evidence of effective and complete standard OR unambiguous and complete non-standard mitigation found), Weak/No Mitigation (no clear/effective/complete mitigation found).

Binary Classification Decision (Bias towards Recall):

Based on the analysis:

Classify "vulnerable: YES" if:

One or more Plausible Indicators (A, B, C, or D from Step 3) were identified AND

Step 4 assessed the mitigation within the function as Weak/No Mitigation.

Classify "vulnerable: NO" if:

No Plausible Indicators were identified in Step 3.

OR A Plausible Indicator was identified, BUT Step 4 found clear evidence of Strong Mitigation (standard or non-standard) effectively and completely neutralizing it within the function's scope.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which Plausible Indicator(s) were triggered, explain the supporting TRIZ analysis (Function/Resource/Contradiction), and state why internal mitigation was assessed as Weak/No (lacking standard or clear/complete non-standard controls).

If "NO": State that no Plausible Indicators were identified OR specify the indicator found and explain the Strong Mitigation (standard or non-standard) identified within the function and why it is considered effective and complete in neutralizing the indicator.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO

FINAL INSTRUCTION:

Apply the TRIZ analysis rigorously, prioritizing Function and Resource analysis to identify plausible weaknesses. Base your final decision on the presence of these indicators unless clear, effective, and complete mitigation (standard or non-standard) is verifiable within the function. Aim to significantly increase Recall while maintaining reasonable Precision. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 7 – V9 Variation 1 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with systematic problem-solving principles to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by achieving the best possible balance between detecting true vulnerabilities (Recall) and minimizing false alarms (Precision). Apply the systematic analytical process rigorously below, focusing on identifying high-confidence vulnerability indicators verifiable within the snippet.

INPUT:
```
{code}
```

SYSTEMATIC ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and any provided context to understand the function's purpose, inputs, outputs, and apparent operating environment.

Problem Framing & Weakness Identification:
(a) Functional Security Review: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences like executing commands, writing arbitrary data) or insufficient useful functions (missing necessary security actions like validation, checks, cleanup - violates principles of preparing for potential issues).
(b) Data/Resource Handling Analysis: List key resources manipulated (inputs, memory, files, state variables). Evaluate how they are controlled. Focus on the flow of potentially untrusted resources (especially function arguments, data read within the function).
(c) Conflicting Requirements Check: Identify any core conflicting requirements the function attempts to resolve (e.g., process input quickly vs. process input safely). Note signs of poor resolution.

Identify High-Confidence Vulnerability Indicators:
Based on Step 2, determine if any of the following high-confidence indicators are clearly present within the function's logic:
Indicator A (Taint Flow to Sink): Clear path for a potentially untrusted input/resource to reach a known dangerous sink (e.g., system call, DB query, unescaped web output, file path operation - a harmful function) without verifiable evidence of standard, effective sanitization/validation/encoding applied to that specific resource before the sink (an insufficient useful function, lack of performing necessary actions beforehand).

Indicator B (Resource Mishandling): Clear logical flaw in resource management visible within the function (e.g., pointer arithmetic suggesting buffer overflow, clear potential for double-free/use-after-free based on resource state tracking within the function, leak of sensitive data/resource to logs/output).

Indicator C (Auth/Access Control Flaw): Clear absence of necessary authorization/access control checks within the function before performing an operation identified as security-sensitive based on its name or actions. (an insufficient useful function).

Indicator D (Dangerous Function Usage): Direct use of known insecure functions/APIs (e.g., strcpy, sprintf, insecure random number generator for security context) with inputs whose size or content is not demonstrably controlled within the function.

Contextual Evaluation (Mitigation Check for Indicators):

If one or more High-Confidence Indicators (A, B, C, D) were identified in Step 3:

Re-examine the code_snippet specifically for concrete evidence of standard, applicable, and correctly implemented mitigating controls that directly and effectively neutralize the identified indicator(s) within the function's scope. (E.g., Does the function itself apply htmlspecialchars right before echoing user input? Does it use snprintf instead of sprintf with size limits?)

Assess Mitigation Confidence: Strong Internal Mitigation (clear, standard, effective control found within the function for the specific indicator), Weak/No Internal Mitigation (no clear/standard/effective control found within the function for the indicator).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Indicators (A, B, C, or D from Step 3) were clearly identified AND

Step 4 assessed the mitigation within the function as Weak/No Internal Mitigation.

Classify "vulnerable: NO" if:

No High-Confidence Indicators were clearly identified in Step 3.

OR A High-Confidence Indicator was identified, BUT Step 4 found Strong Internal Mitigation effectively neutralizing it within the function's scope.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which High-Confidence Indicator(s) (A, B, C, D) were triggered, explain the supporting analysis (taint flow, resource issue, missing checks), and state why internal mitigation was assessed as Weak/No.

If "NO": State that no High-Confidence Indicators were clearly identified OR specify the indicator found and explain the Strong Internal Mitigation identified within the function that neutralizes it.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO.

FINAL INSTRUCTION:

Apply the analysis rigorously, focusing on identifying the specified High-Confidence Indicators based on evidence within the provided code snippet. Base your final decision on the presence of these indicators coupled with the assessment of verifiable mitigations within the function. Aim for the optimal F1-Score. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 8 – V9 Variation 2 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by achieving the best possible balance between detecting true vulnerabilities (Recall) and minimizing false alarms (Precision). Apply the TRIZ-based analytical process rigorously below, focusing on identifying high-confidence vulnerability indicators verifiable within the snippet.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and any provided context to understand the function's purpose, inputs, outputs, and apparent operating environment.

TRIZ Problem Framing & Weakness Identification:
(a) Function Analysis: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences like executing commands, writing arbitrary data) or insufficient useful functions (missing necessary security actions like validation, checks, cleanup - violates Principle 11: Beforehand Cushioning).
(b) Resource Analysis: List key resources manipulated (inputs, memory, files, state variables). Evaluate how they are controlled. Focus on the flow of potentially untrusted resources (especially function arguments, data read within the function).
~~(c) Contradiction Analysis: Identify any core contradictions the function attempts to resolve (e.g., process input quickly vs. process input safely). Note signs of poor resolution.~~

Identify High-Confidence Vulnerability Indicators (TRIZ Lens):
Based on Step 2, determine if any of the following high-confidence indicators are clearly present within the function's logic:
Indicator A (Taint Flow to Sink): Clear path for a potentially untrusted input/resource (Resource) to reach a known dangerous sink (e.g., system call, DB query, unescaped web output, file path operation - Harmful Function) without verifiable evidence of standard, effective sanitization/validation/encoding applied to that specific resource before the sink (Insufficient Useful Function, Lack of Principle 9/10).

Indicator B (Resource Mishandling): Clear logical flaw in resource management visible within the function (e.g., pointer arithmetic suggesting buffer overflow, clear potential for double-free/use-after-free based on resource state tracking within the function, leak of sensitive data Resource to logs/output).

Indicator C (Auth/Access Control Flaw): Clear absence of necessary authorization/access control checks within the function before performing an operation identified as security-sensitive based on its name or actions. (Insufficient Useful Function).

Indicator D (Dangerous Function Usage): Direct use of known insecure functions/APIs (e.g., strcpy, sprintf, insecure random number generator for security context) with inputs whose size or content is not demonstrably controlled within the function.

Contextual Evaluation (Mitigation Check for Indicators):

If one or more High-Confidence Indicators (A, B, C, D) were identified in Step 3:

Re-examine the code_snippet specifically for concrete evidence of standard, applicable, and correctly implemented mitigating controls that directly and effectively neutralize the identified indicator(s) within the function's scope. (E.g., Does the function itself apply htmlspecialchars right before echoing user input? Does it use snprintf instead of sprintf with size limits?)

Assess Mitigation Confidence: Strong Internal Mitigation (clear, standard, effective control found within the function for the specific indicator), Weak/No Internal Mitigation (no clear/standard/effective control found within the function for the indicator).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Indicators (A, B, C, or D from Step 3) were clearly identified AND

Step 4 assessed the mitigation within the function as Weak/No Internal Mitigation.

Classify "vulnerable: NO" if:

No High-Confidence Indicators were clearly identified in Step 3.

OR A High-Confidence Indicator was identified, BUT Step 4 found Strong Internal Mitigation effectively neutralizing it within the function's scope.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which High-Confidence Indicator(s) (A, B, C, D) were triggered, explain the supporting analysis (taint flow, resource issue, missing checks), and state why internal mitigation was assessed as Weak/No.

If "NO": State that no High-Confidence Indicators were clearly identified OR specify the indicator found and explain the Strong Internal Mitigation identified within the function that neutralizes it.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO.

FINAL INSTRUCTION:

Apply the analysis rigorously, focusing on identifying the specified High-Confidence Indicators based on evidence within the provided code snippet. Base your final decision on the presence of these indicators coupled with the assessment of verifiable mitigations within the function. Aim for the optimal F1-Score. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 9 – V9 Variation 3 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with systematic problem-solving principles inspired by TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by achieving the best possible balance between detecting true vulnerabilities (Recall) and minimizing false alarms (Precision). Apply the systematic analytical process rigorously below, focusing on identifying high-confidence vulnerability indicators verifiable within the snippet.

INPUT:
```
{code}
```

SYSTEMATIC ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and any provided context to understand the function's purpose, inputs, outputs, and apparent operating environment.

High-Level Weakness Identification:
Briefly analyze the function's purpose, inputs, outputs, key resources (especially inputs and state), and logic to identify potential areas where security weaknesses might occur (e.g., complex logic, external interactions, data handling, missing checks, conflicting requirements).

Identify High-Confidence Vulnerability Indicators:
Based on Step 2, determine if any of the following high-confidence indicators are clearly present within the function's logic:
Indicator A (Taint Flow to Sink): Clear path for a potentially untrusted input/resource to reach a known dangerous sink (e.g., system call, DB query, unescaped web output, file path operation) without verifiable evidence of standard, effective sanitization/validation/encoding applied to that specific resource before the sink.
Indicator B (Resource Mishandling): Clear logical flaw in resource management visible within the function (e.g., pointer arithmetic suggesting buffer overflow, clear potential for double-free/use-after-free based on resource state tracking within the function, leak of sensitive data to logs/output).
Indicator C (Auth/Access Control Flaw): Clear absence of necessary authorization/access control checks within the function before performing an operation identified as security-sensitive based on its name or actions.

Indicator D (Dangerous Function Usage): Direct use of known insecure functions/APIs (e.g., strcpy, sprintf, insecure random number generator for security context) with inputs whose size or content is not demonstrably controlled within the function.

Contextual Evaluation (Mitigation Check for Indicators):

If one or more High-Confidence Indicators (A, B, C, D) were identified in Step 3:

Re-examine the code_snippet specifically for concrete evidence of standard, applicable, and correctly implemented mitigating controls that directly and effectively neutralize the identified indicator(s) within the function's scope. (E.g., Does the function itself apply htmlspecialchars right before echoing user input? Does it use snprintf instead of sprintf with size limits?)

Assess Mitigation Confidence: Strong Internal Mitigation (clear, standard, effective control found within the function for the specific indicator), Weak/No Internal Mitigation (no clear/standard/effective control found within the function for the indicator).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Indicators (A, B, C, or D from Step 3) were clearly identified AND

Step 4 assessed the mitigation within the function as Weak/No Internal Mitigation.

Classify "vulnerable: NO" if:

No High-Confidence Indicators were clearly identified in Step 3.

OR A High-Confidence Indicator was identified, BUT Step 4 found Strong Internal Mitigation effectively neutralizing it within the function's scope.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which High-Confidence Indicator(s) (A, B, C, D) were triggered, explain the supporting analysis (taint flow, resource issue, missing checks), and state why internal mitigation was assessed as Weak/No.

If "NO": State that no High-Confidence Indicators were clearly identified OR specify the indicator found and explain the Strong Internal Mitigation identified within the function that neutralizes it.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO.

FINAL INSTRUCTION:

Apply the analysis rigorously, focusing on identifying the specified High-Confidence Indicators based on evidence within the provided code snippet. Base your final decision on the presence of these indicators coupled with the assessment of verifiable mitigations within the function. Aim for the optimal F1-Score. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 10 – V9 Variation 4 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by achieving the best possible balance between detecting true vulnerabilities (Recall) and minimizing false alarms (Precision). Apply the TRIZ-based analytical process rigorously below, focusing on identifying high-confidence vulnerability mechanisms verifiable within the snippet.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:

Analyze the code_snippet and any provided context to understand the function's purpose, inputs, outputs, and apparent operating environment.

TRIZ Problem Framing & Weakness Identification:

(a) Function Analysis: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences like executing commands, writing arbitrary data) or insufficient useful functions (missing necessary security actions like validation, checks, cleanup - violates Principle 11: Beforehand Cushioning).

(b) Resource Analysis: List key resources manipulated (inputs, memory, files, state variables). Evaluate how they are controlled. Focus on the flow of potentially untrusted resources (especially function arguments, data read within the function).

(c) Contradiction Analysis: Identify any core contradictions the function attempts to resolve (e.g., process input quickly vs. process input safely). Note signs of poor resolution.

Identify High-Confidence Vulnerability Mechanisms:

Based on the TRIZ analysis (Step 2), determine if any high-confidence vulnerability mechanisms (e.g., potential for injection, unsafe resource handling, missing access controls, use of dangerous functions with uncontrolled input) are clearly present within the function's logic.

Contextual Evaluation (Mitigation Check for Mechanisms):

If one or more High-Confidence Vulnerability Mechanisms were identified in Step 3:

Re-examine the code_snippet specifically for concrete evidence of standard, applicable, and correctly implemented mitigating controls that directly and effectively neutralize the identified mechanism(s) within the function's scope. (E.g., Does the function itself apply htmlspecialchars right before echoing user input? Does it use snprintf instead of sprintf with size limits?)

Assess Mitigation Confidence: Strong Internal Mitigation (clear, standard, effective control found within the function for the specific mechanism), Weak/No Internal Mitigation (no clear/standard/effective control found within the function for the mechanism).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Vulnerability Mechanisms (from Step 3) were clearly identified AND

Step 4 assessed the mitigation within the function as Weak/No Internal Mitigation.

Classify "vulnerable: NO" if:

No High-Confidence Vulnerability Mechanisms were clearly identified in Step 3.

OR A High-Confidence Vulnerability Mechanism was identified, BUT Step 4 found Strong Internal Mitigation effectively neutralizing it within the function's scope.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which High-Confidence Vulnerability Mechanism(s) were triggered, explain the supporting TRIZ analysis, and state why internal mitigation was assessed as Weak/No.

If "NO": State that no High-Confidence Vulnerability Mechanisms were clearly identified OR specify the mechanism found and explain the Strong Internal Mitigation identified within the function that neutralizes it.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO.

FINAL INSTRUCTION:

Apply the TRIZ analysis rigorously, focusing on identifying high-confidence vulnerability mechanisms based on evidence within the provided code snippet. Base your final decision on the presence of these mechanisms coupled with the assessment of verifiable mitigations within the function. Aim for the optimal F1-Score. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 11 – V9 Variation 5 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by achieving the best possible balance between detecting true vulnerabilities (Recall) and minimizing false alarms (Precision). Apply the TRIZ-based analytical process rigorously below, focusing on identifying high-confidence vulnerability indicators verifiable within the snippet.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and any provided context to understand the function's purpose, inputs, outputs, and apparent operating environment.

TRIZ Problem Framing & Weakness Identification:
(a) Function Analysis: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences like executing commands, writing arbitrary data) or insufficient useful functions (missing necessary security actions like validation, checks, cleanup - violates Principle 11: Beforehand Cushioning).
(b) Resource Analysis: List key resources manipulated (inputs, memory, files, state variables). Evaluate how they are controlled. Focus on the flow of potentially untrusted resources (especially function arguments, data read within the function).
(c) Contradiction Analysis: Identify any core contradictions the function attempts to resolve (e.g., process input quickly vs. process input safely). Note signs of poor resolution.

Identify High-Confidence Vulnerability Indicators (TRIZ Lens):
Based on Step 2, determine if any of the following high-confidence indicators are clearly present within the function's logic:
Indicator A (Taint Flow to Sink): Clear path for a potentially untrusted input/resource (Resource) to reach a known dangerous sink (e.g., system call, DB query, unescaped web output, file path operation - Harmful Function) without verifiable evidence of standard, effective sanitization/validation/encoding applied to that specific resource before the sink (Insufficient Useful Function, Lack of Principle 9/10).

Indicator B (Resource Mishandling): Clear logical flaw in resource management visible within the function (e.g., pointer arithmetic suggesting buffer overflow, clear potential for double-free/use-after-free based on resource state tracking within the function, leak of sensitive data Resource to logs/output).

Indicator C (Auth/Access Control Flaw): Clear absence of necessary authorization/access control checks within the function before performing an operation identified as security-sensitive based on its name or actions. (Insufficient Useful Function).

Indicator D (Dangerous Function Usage): Direct use of known insecure functions/APIs (e.g., strcpy, sprintf, insecure random number generator for security context) with inputs whose size or content is not demonstrably controlled within the function.

Contextual Evaluation (Mitigation Check for Indicators):

If one or more High-Confidence Indicators (A, B, C, D) were identified in Step 3:

Re-examine the code_snippet specifically for concrete evidence of standard, applicable, and correctly implemented mitigating controls that directly and effectively neutralize the identified indicator(s) within the function's scope. (E.g., Does the function itself apply htmlspecialchars right before echoing user input? Does it use snprintf instead of sprintf with size limits?)

Assess Mitigation Presence: Mitigation Present (clear, standard, effective control found within the function for the specific indicator), Mitigation Absent (no clear/standard/effective control found within the function for the indicator).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Indicators (A, B, C, or D from Step 3) were clearly identified AND

Step 4 assessed mitigation as Mitigation Absent.

Classify "vulnerable: NO" if:

No High-Confidence Indicators were clearly identified in Step 3.

OR A High-Confidence Indicator was identified, BUT Step 4 assessed mitigation as Mitigation Present.

Justification:

Provide a clear rationale explaining the YES or NO decision.

If "YES": State which High-Confidence Indicator(s) (A, B, C, D) were triggered, explain the supporting TRIZ analysis, and state that internal mitigation was assessed as Absent.

If "NO": State that no High-Confidence Indicators were clearly identified OR specify the indicator found and explain the Mitigation identified within the function that neutralizes it.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO.

FINAL INSTRUCTION:

Apply the TRIZ analysis rigorously, focusing on identifying the specified High-Confidence Indicators based on evidence within the provided code snippet. Base your final decision on the presence of these indicators coupled with the assessment of verifiable mitigation presence within the function. Aim for the optimal F1-Score. Provide a detailed justification. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 12 – V9 Variation 6 TRIZ Prompt

ROLE:
You are an expert AI assistant specializing in software security analysis. Your primary methodology integrates standard security knowledge with the systematic problem-solving principles of TRIZ (Theory of Inventive Problem Solving) to provide deep, context-aware vulnerability assessments for individual functions.

TASK:
Analyze the provided code_snippet (representing a single function) and any accompanying context. Your goal is to accurately classify whether this function contains an exploitable security vulnerability ("vulnerable: YES" or "vulnerable: NO"). Aim to maximize the F1-Score by achieving the best possible balance between detecting true vulnerabilities (Recall) and minimizing false alarms (Precision). Apply the TRIZ-based analytical process rigorously below, focusing on identifying high-confidence vulnerability indicators verifiable within the snippet.

INPUT:
```
{code}
```

TRIZ-BASED ANALYTICAL PROCESS (Apply sequentially before making the final YES/NO decision):

Initial Contextual Understanding:
Analyze the code_snippet and any provided context to understand the function's purpose, inputs, outputs, and apparent operating environment.

TRIZ Problem Framing & Weakness Identification:
(a) Function Analysis: Describe intended useful functions. Identify potential harmful functions (actions with negative security consequences like executing commands, writing arbitrary data) or insufficient useful functions (missing necessary security actions like validation, checks, cleanup - violates Principle 11: Beforehand Cushioning).
(b) Resource Analysis: List key resources manipulated (inputs, memory, files, state variables). Evaluate how they are controlled. Focus on the flow of potentially untrusted resources (especially function arguments, data read within the function).
(c) Contradiction Analysis: Identify any core contradictions the function attempts to resolve (e.g., process input quickly vs. process input safely). Note signs of poor resolution.

Identify High-Confidence Vulnerability Indicators (TRIZ Lens):
Based on Step 2, determine if any of the following high-confidence indicators are clearly present within the function's logic:
Indicator A (Taint Flow to Sink): Clear path for a potentially untrusted input/resource (Resource) to reach a known dangerous sink (e.g., system call, DB query, unescaped web output, file path operation - Harmful Function) without verifiable evidence of standard, effective sanitization/validation/encoding applied to that specific resource before the sink (Insufficient Useful Function, Lack of Principle 9/10).

Indicator B (Resource Mishandling): Clear logical flaw in resource management visible within the function (e.g., pointer arithmetic suggesting buffer overflow, clear potential for double-free/use-after-free based on resource state tracking within the function, leak of sensitive data Resource to logs/output).

Indicator C (Auth/Access Control Flaw): Clear absence of necessary authorization/access control checks within the function before performing an operation identified as security-sensitive based on its name or actions. (Insufficient Useful Function).

Indicator D (Dangerous Function Usage): Direct use of known insecure functions/APIs (e.g., strcpy, sprintf, insecure random number generator for security context) with inputs whose size or content is not demonstrably controlled within the function.

Contextual Evaluation (Mitigation Check for Indicators):

If one or more High-Confidence Indicators (A, B, C, D) were identified in Step 3:

Re-examine the code_snippet specifically for concrete evidence of standard, applicable, and correctly implemented mitigating controls that directly and effectively neutralize the identified indicator(s) within the function's scope. (E.g., Does the function itself apply htmlspecialchars right before echoing user input? Does it use snprintf instead of sprintf with size limits?)

Assess Mitigation Confidence: Strong Internal Mitigation (clear, standard, effective control found within the function for the specific indicator), Weak/No Internal Mitigation (no clear/standard/effective control found within the function for the indicator).

Binary Classification Decision:

Based on the analysis:

Classify "vulnerable: YES" if:

One or more High-Confidence Indicators (A, B, C, or D from Step 3) were clearly identified AND

Step 4 assessed the mitigation within the function as Weak/No Internal Mitigation.

Classify "vulnerable: NO" if:

No High-Confidence Indicators were clearly identified in Step 3.

OR A High-Confidence Indicator was identified, BUT Step 4 found Strong Internal Mitigation effectively neutralizing it within the function's scope.

OUTPUT FORMAT:

Output only the final classification:

Classification: vulnerable: YES or vulnerable: NO

FINAL INSTRUCTION:

Apply the TRIZ analysis rigorously, focusing on identifying the specified High-Confidence Indicators based on evidence within the provided code snippet. Base your final decision on the presence of these indicators coupled with the assessment of verifiable mitigations within the function. Aim for the optimal F1-Score. Your output must be only "vulnerable: YES" or "vulnerable: NO".

Appendix 13 – Experiment Script

```
import json
import argparse
import os
import sys
import time
from openai import OpenAI

SECRET_FILE = ".secret"
DEFAULT_OUTPUT_SUFFIX = "_llm_results.jsonl"

def read_api_key(file_path):
    try:
        with open(file_path, "r", encoding='utf-8') as file:
            api_key = file.read().strip()
            if not api_key:
                print(f"Error: API key file '{file_path}' is empty.", file=sys.stderr)
                sys.exit(1)
            return api_key
    except FileNotFoundError:
        print(f"Error: API key file '{file_path}' not found.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"Error reading API key from '{file_path}': {e}", file=sys.stderr)
        sys.exit(1)

def read_prompt_template(prompt_type):
    file_path = f"prompt_{prompt_type}.txt"
    try:
        with open(file_path, "r", encoding='utf-8') as file:
            prompt_template = file.read()
            if "{code}" not in prompt_template:
                print(f"Warning: Prompt file '{file_path}' does not contain the '{{code}}' placeholder.",
file=sys.stderr)
            return prompt_template
    except FileNotFoundError:
        print(f"Error: Prompt file '{file_path}' not found.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"Error reading prompt file '{file_path}': {e}", file=sys.stderr)
        sys.exit(1)

def process_code_snippet(snippet, client, prompt_template, item_id, func_type):
    if not snippet:
```

```
        print(f"Skipping empty {func_type} snippet for ID: {item_id}")
        return None
    prompt_message = prompt_template.format(code=snippet)
    print(f"Processing {func_type} for ID: {item_id}...")
    try:
        response = client.chat.completions.create(
            model="deepseek-chat", # Adjust model if needed
            messages=[
                {"role": "user", "content": prompt_message}
            ],
            temperature=0.1, # Optional adjustments
            max_tokens=512
        )
        result = response.choices[0].message.content
        print(f"  -> Received response for {func_type} ID {item_id}.")
        return result
    except Exception as e:
        print(f"  -> FAILED to process {func_type} for ID {item_id}.", file=sys.stderr)
        print(f"  -> Error: {e}", file=sys.stderr)
        time.sleep(5) # Wait after an error
        return None

def parse_llm_verdict(llm_response_text):
    if not llm_response_text:
        return None
    lower_response = llm_response_text.lower()
    found_yes = "vulnerable: yes" in lower_response
    found_no = "vulnerable: no" in lower_response
    if found_yes and not found_no:
        return "1"
    elif found_no and not found_yes:
        return "0"
    elif found_yes and found_no:
        print(f"  -> Warning: Ambiguous verdict. Both 'vulnerable: yes' and 'vulnerable: no' found. Setting
vulnerable: yes!", file=sys.stderr)
        print(f"  -> Response sample: ...{llm_response_text[-150:]}", file=sys.stderr)
        return "1"
    else:
        print(f"  -> Warning: Could not parse verdict. Neither 'vulnerable: yes' nor 'vulnerable: no' found.",
file=sys.stderr)
        print(f"  -> Response sample: ...{llm_response_text[-150:]}", file=sys.stderr)
        return None

def parse_arguments():
    parser = argparse.ArgumentParser(
        description="Process vulnerable/patched code pairs using an LLM API (DeepSeek) "
                    "and save results incrementally to a JSON Lines file."
    )
    parser.add_argument(
        "input_json_file",
        help="Path to the input JSON file containing function pairs."
```

```
    )
    parser.add_argument(
        "prompt_type",
        help="Prompt type identifier (e.g., 'triz' for prompt_triz.txt)."
    )
    parser.add_argument(
        "-o", "--output",
        help=f"Optional: Path to the output JSON Lines file (.jsonl). "
             f"Defaults to '<input_base>_<prompt_type>{DEFAULT_OUTPUT_SUFFIX}'."
    )
    return parser.parse_args()

def main():
    args = parse_arguments()
    input_file = args.input_json_file
    prompt_type = args.prompt_type

    if args.output:
        output_file_name = args.output
        if not output_file_name.lower().endswith('.jsonl'):
            print(f"Warning: Specified output file '{output_file_name}' does not end with .jsonl. Appending
results in JSON Lines format.", file=sys.stderr)
    else:
        base, _ = os.path.splitext(input_file)
        output_file_name = f"{base}_{prompt_type}{DEFAULT_OUTPUT_SUFFIX}"

    print(f"Input JSON: {input_file}")
    print(f"Prompt Type: {prompt_type}")
    print(f"Output JSON Lines: {output_file_name} (appending)") # Indicate append mode

    api_key = read_api_key(SECRET_FILE)
    prompt_template = read_prompt_template(prompt_type)
    client = OpenAI(api_key=api_key, base_url="https://api.deepseek.com")

    try:
        with open(input_file, 'r', encoding='utf-8') as infile:
            function_pairs_data = json.load(infile)
        if not isinstance(function_pairs_data, list):
            print(f"Error: Input file '{input_file}' does not contain a JSON list.", file=sys.stderr)
            sys.exit(1)
    except FileNotFoundError:
        print(f"Error: Input file not found: {input_file}", file=sys.stderr)
        sys.exit(1)
    except json.JSONDecodeError as e:
        print(f"Error: Invalid JSON format in {input_file}: {e}", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"Error reading input file {input_file}: {e}", file=sys.stderr)
        sys.exit(1)
```

```
    total_items = len(function_pairs_data)
    processed_count = 0
    error_count = 0

    try:
        with open(output_file_name, 'a', encoding='utf-8') as outfile:
            for index, item in enumerate(function_pairs_data):
                item_id = item.get("id")
                vuln_code = item.get("vulnerable_func")
                patched_code = item.get("patched_func")
                print(f"\n--- Processing Item {index + 1}/{total_items} (ID: {item_id}) ---")
                if not item_id:
                    print("  -> Warning: Skipping item due to missing 'id'.", file=sys.stderr)
                    error_count += 1
                    continue

                vuln_response = process_code_snippet(vuln_code, client, prompt_template, item_id,
"vulnerable_func")
                vuln_verdict = parse_llm_verdict(vuln_response)
                time.sleep(1) # Small delay

                patched_response = process_code_snippet(patched_code, client, prompt_template, item_id,
"patched_func")
                patched_verdict = parse_llm_verdict(patched_response)
                time.sleep(1) # Small delay

                result_item = {
                    "id": item_id,
                    "vulnerable_func": vuln_verdict,
                    "patched_func": patched_verdict
                }

                try:
                    outfile.write(json.dumps(result_item, ensure_ascii=False) + '\n')
                    outfile.flush() # Ensure it's written to disk immediately
                    processed_count += 1
                except Exception as write_e:
                     print(f"  -> Error writing result for ID {item_id} to {output_file_name}: {write_e}",
file=sys.stderr)
                     error_count += 1
    except IOError as e:
        print(f"\nFATAL: Error opening or writing to output file {output_file_name}: {e}", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"\nFATAL: An unexpected error occurred during processing: {e}", file=sys.stderr)
        sys.exit(1)

    print("\n--- Processing Complete ---")
    print(f"Total items in input: {total_items}")
    print(f"Successfully processed and written: {processed_count}")
```

```
    print(f"Items skipped or failed during processing/writing: {error_count + (total_items - processed_count)}")
    print(f"Results appended to {output_file_name}")

if __name__ == "__main__":
    main()
```

Appendix 14 – DFA Prompt

You are a security researcher, expert in detecting security vulnerabilities. Carefully analyze the given code snippet and track the data flows from various sources to sinks. Assume that any call to an unknown external API is unsanitized. Please provide a response only in the following format:

Here is a data flow analysis of the given code snippet:

A. Sources: <numbered list of input sources>

B. Sinks: <numbered list of output sinks>

C. Sanitizers: <numbered list of sanitizers, if any>

D. Unsanitized Data Flows: <numbered list of data flows that are not sanitized in the format (source, sink, why this flow could be vulnerable)>

E. Vulnerability analysis verdict: vulnerable: <YES or NO>

Is the following code snippet prone to any security vulnerability?

```
{code}
```

Review your initial answer and find problems with that answer. Feel free to answer in any format you prefer.

Based on the problems found, improve your initial answer:

Provide response only in following format:

vulnerable: <YES or NO>

Do not include anything else in response.