CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

LLMs for SDVs: Automated Software Vulnerability Detection and Repair

Master's Thesis in Computer science and engineering

Wenkang Gong, Jieman Yan

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's Thesis 2026

LLMs for SDVs: Automated Software Vulnerability Detection and Repair

Wenkang Gong, Jieman Yan

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

LLMs for SDVs: Automated Software Vulnerability Detection and Repair
Wenkang Gong, Jieman Yan

Supervisor: Simin Sun, Miroslaw Staron, Department of Computer Science and Engineering
Examiner: Francisco Gomes de Oliveira Neto, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Wenkang Gong, Jieman Yan
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Software-Defined Vehicles (SDVs) increasingly rely on large-scale C/C++ software stacks to implement safety-critical functionalities. While these languages provide the deterministic performance and hardware control required in automotive systems, they are also susceptible to memory-safety vulnerabilities such as buffer overflows, out-of-bounds accesses, NULL pointer dereferences, and resource management errors. Existing vulnerability analysis approaches remain essential in industrial practice but face limitations in scalability, coverage, and manual remediation effort when applied to modern automotive-scale software systems. Recent advances in Large Language Models (LLMs) have motivated increasing research interest in automated vulnerability detection and repair.

This thesis presents an experimental study of a two-stage detection and repair pipeline for function-level C/C++ memory-safety vulnerability detection and repair in software relevant to SDVs. For the detection stage, the study evaluates how classification strategy, pre-trained code model selection, and inference-time threshold selection affect detection performance for four vulnerability categories, Common Weakness Enumeration (CWE)-787, CWE-476, CWE-399, and CWE-125. Detection experiments compare CodeBERT, GraphCodeBERT, and UniXcoder across specialised binary classifiers and a shared multiclass classifier. For the repair stage, the study evaluates how detection-augmented prompting using vulnerability guidance affect LLM-based automated vulnerability repair performance. Repair experiments evaluate three prompting strategies with increasing levels of vulnerability guidance.

Experiments on the BigVul and PrimeVul datasets show that the specialised binary classifiers outperform the multiclass classifier for all model-CWE combinations, with per-CWE F1-score improvements ranging from +0.13 to +0.35. The results also show that no evaluated pre-trained code model is strongest across all four CWE types. Thresholds selected on validation F1 make the detector more permissive, increasing the rate at which the ground-truth CWE reaches the repair stage by 11.6 to 21.4 percentage points; UniXcoder achieves the highest detection rate of 85.9%. For vulnerability repair, detection-augmented prompting improves vulnerability repair performance, increasing the vulnerability pattern removal rate from 28.22% under the unguided baseline to 48.43% under the detailed guided prompting strategy, while maintaining high code quality.

The results indicate that specialised binary classifiers are the strongest evaluated architecture, while model selection and threshold selection still affect how these classifiers perform within the pipeline. Moreover, incorporating detection results into repair prompts proves an effective strategy for improving vulnerability repair quality, though the improvement is bounded by upstream detection accuracy.

Keywords: Software-Defined Vehicles, Vulnerability Detection, Automated Program Repair, Large Language Models, Pre-trained Code Models

Acknowledgements

We would like to express our sincere gratitude to our supervisors, Simin Sun and Miroslaw Staron, for their continuous guidance, thoughtful discussions, and valuable suggestions throughout this thesis project. Their feedback and encouragement greatly helped us refine both the research direction and the overall quality of this work.

We would also like to thank our examiner, Francisco Gomes de Oliveira Neto, for his rigorous and insightful comments on the thesis drafts. His critical observations and constructive suggestions helped us identify weaknesses in the work and improve the clarity and quality of the thesis.

In addition, we would like to thank the Department of Computer Science and Engineering at Chalmers University of Technology and the University of Gothenburg for providing an inspiring academic environment and the resources necessary for conducting this research.

Finally, we would like to express our gratitude to our families and friends for their patience, encouragement, and support throughout our studies and during the completion of this thesis.

Wenkang Gong, Jieman Yan, Gothenburg, June 2026

Contents

List of Figures

List of Tables

1

Introduction

This chapter establishes the motivation and scope of the study. It opens with the background of Software-Defined Vehicles and the security challenges posed by their C/C++ software foundations, followed by a motivating example that illustrates the class of vulnerability the study targets. The chapter then states the gap left by existing approaches, presents the research questions and contributions, and concludes with the delimitations that define the study's boundaries.

1.1 Background

The automotive industry is undergoing a fundamental shift towards SDVs, in which vehicle functionalities, behaviours, and core features are primarily controlled by software rather than by dedicated mechanical or electronic hardware [8]. In traditional vehicle architectures, software is distributed across dozens of separate Electronic Control Units (ECUs), each dedicated to a specific function such as engine management or braking. By contrast, the SDV paradigm is moving towards consolidated compute platforms in which multiple functions share centralised hardware and new capabilities can be delivered, updated, or reconfigured through software independently of the physical vehicle [8]. A key enabler of this transition is the over-the-air update mechanism, which allows manufacturers to deploy features and security patches remotely throughout the vehicle's operational life [50].

Across both traditional distributed ECUs architectures and emerging centralised platforms, the total volume of vehicle software is substantial and continues to grow. According to Sheehan et al. [50], modern vehicles routinely execute approximately 100 million lines of code. Dibaei et al. [11] further report that the software in a Ford vehicle had already reached 150 million lines of code by 2016, while De Vincenzi et al. [8] note that fully autonomous systems will require up to a billion lines of code to ensure the reliable and precise performance needed for high-level automation. Due to the strict requirements for deterministic latency, real-time processing, and direct hardware manipulation, the foundational software layers of SDVs, including Real-Time Operating Systems (RTOS), hardware abstraction layers, and middleware, are predominantly implemented in C and C++ [8, 30]. Industry-wide standards further reinforce this language choice. Specifically, the Automotive Open System Architecture (AUTOSAR) partnership, which defines a standardized software ar-

chitecture for vehicle ECUs, mandates the use of C and C++ coding guidelines for safety-critical development [1].

However, the reliance on C and C++ in these foundational layers is accompanied by a high prevalence of memory-safety vulnerabilities [8]. In SDVs, such vulnerabilities can affect more than software quality because low-level components increasingly sit behind externally reachable interfaces. Modern SDVs expose connectivity channels including cellular networks, Bluetooth, Wi-Fi, and vehicle-to-everything protocols [11]. A memory-safety flaw in software reachable through these interfaces can therefore become an entry point into lower-level vehicle systems. Miller and Valasek [40] demonstrated this risk by showing that an attacker could exploit a car's connectivity firmware to access the internal Controller Area Network (CAN) bus. Since the CAN bus lacks built-in authentication [15], the attacker can then seize control of safety-critical systems, including braking and steering [11, 30].

To study this class of risk in a controlled way, the thesis uses MITRE's CWE framework, a standard taxonomy for software weakness types [42]. To ensure alignment with SDV-relevant memory safety concerns, available labeled C/C++ vulnerability data, and the practical constraints of function-level detection and repair, we selected our target categories. Accordingly, this study focuses on out-of-bounds writes (CWE-787), out-of-bounds reads (CWE-125), NULL pointer dereferences (CWE-476), and resource management errors (CWE-399). Together, these categories define the memory-safety vulnerability scope for the function-level detection-repair pipeline developed in this thesis.

A Motivating Example To illustrate the memory vulnerabilities, we consider the following example that processes a network packet received over a vehicle's Bluetooth interface.

Listing 1.1: A simplified packet-processing function containing a CWE-787 out-of-bounds write.

```
void process_packet(char *data, int len) {
    char buf[64];
    memcpy(buf, data, len);
    handle_command(buf);
}
```

The function allocates a fixed-size stack buffer of 64 bytes and copies the incoming data into it using the caller-supplied length. If an attacker sends a packet where `len` exceeds 64, the `memcpy` call writes past the end of `buf`, overwriting adjacent memory on the stack. This is a classic CWE-787 out-of-bounds write vulnerability. And as `buf` resides on the stack alongside the function's saved return address, writing beyond its boundary allows an attacker to overwrite that address. When the function returns, the processor jumps to the attacker-supplied address, redirecting execution to attacker-controlled code.

In an SDV context, this vulnerability is not merely a software quality issue. Compo-

nents such as gateways sit at the boundary between external communication interfaces and the internal CAN bus, which connects safety-critical actuators including braking and steering systems. Because the CAN bus lacks built-in authentication, any node on the bus can send commands to any other node without verification as mentioned before. Therefore, an attacker who achieves code execution on such a component through a vulnerability like this example can issue arbitrary CAN messages, including commands that directly affect vehicle motion.

1.2 Problem Statement

Mitigating the security challenges outlined above requires effective approaches for both detecting and repairing vulnerabilities. Existing approaches face practical constraints when applied to the complex C/C++ software that supports automotive and other safety-critical systems.

Conventional static and dynamic analysis tools remain indispensable, as safety standards such as ISO 26262 and MISRA C require them as part of the verification baseline [1]. Their practical use involves trade-offs. Static Application Security Testing (SAST) tools often prioritize recall to reduce missed vulnerabilities, but this can produce many false positives that require manual triage. Lipp et al. [35] also found that C/C++ static analysers can miss known real-world Common Vulnerabilities and Exposures (CVE) cases. Dynamic Application Security Testing (DAST) tools provide runtime evidence, but they require extensive test harnesses and often struggle to cover complex embedded code paths [8]. These limitations motivate the exploration of deep learning-based approaches capable of capturing nuanced semantic patterns. Pre-trained code models are one such approach because they can learn vulnerability-relevant representations from vast codebases [18].

Both conventional static analysers and neural vulnerability classifiers function primarily as diagnostic instruments: they identify potential issues but do not synthesize secure patches by themselves. The burden of producing a secure patch for every detected issue still falls on human developers. Manual code review and repair become a bottleneck when developers face the volume of security alerts typical in modern vehicle software. LLMs offer a possible way to reduce this repair burden. Xia and Zhang [53] showed that LLMs can produce syntactically valid and contextually plausible patches for Automated Program Repair (APR), outperforming classical repair tools across several benchmarks [52].

Applying these general-purpose generative models to safety-critical SDV code introduces an additional category of risk. Without structured vulnerability localisation, an unguided LLM must simultaneously infer the nature of the security flaw and determine the appropriate fix from the raw code alone. This lack of explicit guidance increases the risk of generating a candidate patch that appears plausible but fails to resolve the underlying vulnerability or introduces secondary defects. This failure mode is commonly referred to as a *pseudo-fix* (a patch that appears syntactically reasonable but either fails to address the underlying defect or introduces

a secondary one) [56]. Such unconstrained generation raises concerns about the reliability of unguided LLM-based repair in safety-critical contexts.

This thesis investigates this gap through a cohesive pipeline that connects specialised detection with classifier-guided generative repair. Instead of relying on LLMs to infer both the defect and the solution without external guidance, we explore using the outputs of specialised classifiers to inject discriminative signals into the repair prompt. The study evaluates whether providing the predicted CWE type and relevant information as structured cues helps steer the model toward repair patterns more consistent with the identified vulnerability category. Designing and evaluating this two-stage approach through controlled experiments on labelled benchmark datasets forms the basis of this study.

1.3 Research Questions and Contributions

The study is structured around two main aspects of the detection and repair pipeline. The first aspect concerns the design of the detection module. Accurately classifying specific vulnerability types is necessary to provide reliable guidance to the downstream repair model, yet detection performance can depend on several design choices: whether one shared classifier or separate per-CWE classifiers are used, which pretrained code model is selected, and which inference-time threshold is used when deciding whether each binary classifier reports its target CWE label. Conventional deep learning approaches often use a single multiclass classifier that predicts one label from a shared set of vulnerability classes [54, 61, 19]. This study compares that design with a modular approach in which an independent binary classifier is fine-tuned for each target CWE. The second aspect involves evaluating whether the guided repair module provides measurable improvement. We therefore evaluate whether providing the LLM with structured detection context measurably improves repair quality compared to an unguided baseline. We frame our experimental investigation around the following two research questions.

RQ1: To what extent do classification strategy, model selection, and inference-time threshold selection affect the detection performance of selected memory-safety vulnerabilities in C/C++ functions?

We examine this question to establish the detection configuration used by the pipeline. To start with, the classification-strategy comparison investigates whether four specialised binary classifiers, each dedicated to one target CWE, perform differently from a single shared multiclass classifier. This analysis focuses on four memory-safety categories including CWE-787, CWE-476, CWE-399, and CWE-125. We then repeat this model-selection experiment using CodeBERT, GraphCodeBERT, and UniXcoder to see how the architecture holds up across different models. Finally, the threshold comparison evaluates two ways of deciding when a binary classifier should report its target label. We compare a simple fixed threshold of 0.5 against custom thresholds selected on the validation set to maximize the F1-score. Since these thresholds directly control label reporting based on the positive-class scores,

we evaluate the final prediction quality through precision, recall, F1-score, and the overall pipeline detection rate. Together, these comparisons inform the selection of the detection configuration used as the upstream component of the repair experiments in RQ2.

RQ2: To what extent does detection-augmented prompting improve the performance of LLM-based automated vulnerability repair?

This research question evaluates whether injecting structured vulnerability guidance into the prompt of an LLM yields higher-quality patches compared to an unguided baseline. We evaluate this through comparative experiments across three prompting strategies with progressively increasing levels of vulnerability guidance as detailed in Chapter 5.

The contributions of this study are as follows:

1. **A modular detection and repair pipeline** for function-level C/C++ memory safety vulnerabilities, in which specialised binary classifiers feed structured CWE-type cues into an LLM repair module.

2. **An experimental evaluation of detection-module design choices** across four CWE types, comparing specialised binary classifiers with a shared multi-class classifier, replicating the comparison across three pre-trained code models (CodeBERT, GraphCodeBERT, UniXcoder), and evaluating how inference-time threshold strategies affect pipeline detection rate.

3. **A controlled evaluation of three prompting strategies** (Unguided, Type-Guided, Detail-Guided) for LLM-based vulnerability repair, measuring repair performance across three dimensions: basic code quality, similarity with ground truth, and vulnerability repair quality.

4. **A novel Vulnerability Pattern Removal Rate metric** for evaluating vulnerability repair quality, which assesses whether generated patches apply the expected vulnerability-specific repair patterns, complementing similarity metrics that are insufficient for directly measuring vulnerability repair effectiveness.

1.4 Delimitations

To ensure the feasibility and focus of the research, certain boundaries have been established:

- *Language and domain:* The study focuses on C/C++ code at the function level, targeting the types of memory safety vulnerabilities common in automotive foundational software. Since production automotive codebases are proprietary and unavailable for academic research, the evaluation is conducted on the BigVul [16] and PrimeVul datasets, which comprise vulnerability records

from open-source system-level C/C++ projects (e.g., Linux Kernel, QEMU, Wireshark, FFmpeg). These projects share the low-level memory management patterns, pointer-intensive logic, and vulnerability profiles characteristic of automotive foundational layers, making them a widely used proxy in vulnerability detection research. The results therefore directly characterise the pipeline's effectiveness on system-level C/C++ code; transferability to production automotive software remains an open question for future validation.

- *Vulnerability types:* The study targets four specific, high-impact vulnerability types (CWE-787, CWE-476, CWE-399, and CWE-125) based on their observed prevalence in the BigVul dataset, their relevance to SDV memory-safety threats, and their structural suitability for function-level detection. The rationale for each type is detailed in Section 2.3.

- *Data granularity:* Models are trained and evaluated on function-level vulnerabilities. Whole-system or multi-file architectural flaws are outside the scope of this study.

- *Generative model:* The repair evaluation uses DeepSeek-V3.2, accessed via API with temperature 0 for deterministic output. The study evaluates the detection-augmented prompting with different levels of vulnerability guidance rather than comparing multiple LLMs. Extending to other generative models is identified as future work.

- *Detection models:* The scope of the study is delimited to the evaluation of three selected pre-trained code models acting as classifiers. Generative LLM-based detection is not evaluated.

- *Context scope:* The detection models operate solely on isolated function-level source code without interprocedural, cross-file, or runtime context. Vulnerabilities whose detection depends on external call relationships, global state, macro expansion, or broader program semantics are therefore outside the scope of this study.

2

Background

This chapter provides the foundational technical context necessary for the subsequent evaluation. It reviews the core concepts underlying both vulnerability detection and automated repair. Section 2.1 explains the memory-safety mechanism examined in this thesis. Section 2.2 explains the mechanisms of conventional vulnerability detection tools. Section 2.3 defines the four CWE categories targeted in this evaluation and explains the rationale for their selection. Section 2.4 details the pre-trained code models used for classification. Section 2.5 discusses LLM-based code generation alongside the prompt engineering concepts that form the basis of guided repair.

2.1 Memory-Safety Mechanisms in SDV Software

SDV architectures shift many vehicle functions from separate ECUs toward more consolidated computing platforms. This shift matters for memory safety because functions once hosted on physically separate controllers may now share processors, operating-system services, and middleware layers. Their separation therefore depends not only on hardware placement, but also on software mechanisms such as process boundaries, memory protection, interface checks, and resource management. A memory-safety defect can weaken these local boundaries by allowing a program to access memory outside the object, buffer, or resource lifetime intended by the developer. Depending on where the invalid access occurs, the observable effect may be a crash, information disclosure, corruption of adjacent state, or an unintended control-flow change.

The mechanism is closely connected to the responsibilities left to C and C++ program logic. These languages give developers direct control over allocation, pointer use, buffer indexing, and resource release, while many invalid operations are not checked automatically at runtime [39]. In a function-level detection and repair pipeline, such responsibilities appear as local code patterns, for example an unchecked copy length, a pointer dereferenced before validation, or a resource that is not released along all relevant paths. The target vulnerability categories in this thesis represent four recurring ways in which these responsibilities can fail. These patterns involve writing beyond a buffer boundary, reading beyond a valid memory region, dereferencing a null pointer, and mismanaging allocated resources. Section 2.3 maps

these defect patterns to their corresponding CWE categories and explains the selection rationale.

2.2 Mechanisms of Conventional Detection Tools

SAST tools examine program source code without running it, searching for patterns associated with known vulnerability types [35]. One common approach tracks where unvalidated data enters the program, such as through a network connection or user input, and follows it through each function and variable assignment to determine whether it can reach a sensitive operation, such as writing into a fixed-size memory buffer, without being checked first [22].

A second approach reasons systematically about every possible value a program variable could hold at every point in the code, without executing the program at all. Because this reasoning must account for all conceivable inputs, the analysis tends to flag code paths that look potentially unsafe even if many of those paths would never be reached during normal execution. This cautious strategy is what causes static analysis tools to generate a high number of false alarms alongside genuine findings [26]. At the same time, static approximations remain bounded by the rules, abstractions, and program context available to the analyser.

DAST tools take the opposite approach by running the program and observing what it actually does [8]. One widely used technique, fuzzing, automatically generates large numbers of varied inputs and feeds them to the program, recording any crashes or unexpected behaviour that result [60].

Symbolic execution takes a more systematic approach by reasoning about what specific inputs would be required to reach a particular part of the code and then constructing those inputs deliberately. This can reveal vulnerabilities that purely random testing would miss [2]. The approach, however, runs into a fundamental scaling problem. In any program with multiple decision points, the number of distinct execution paths grows exponentially as the program becomes more complex, making it impractical to test even a small fraction of them [8].

Conventional tools provide valuable but fundamentally partial evidence. In automotive development specifically, dynamic testing is further complicated by the need to reproduce the behaviour of physical hardware components in a test environment. Both families of tools are therefore best understood as instruments offering incomplete coverage rather than comprehensive guarantees, a limitation that continues to motivate the exploration of alternative validation approaches.

2.3 The Targeted CWEs

Discussing and comparing software vulnerabilities requires a shared vocabulary. The CWE, maintained by MITRE Corporation, provides a community-developed dictionary of software and hardware weakness types [42]. Each type is characterised by

its root cause at the source level rather than by any specific exploit instance. This source-level grounding makes CWE a natural basis for training and evaluating vulnerability detection models.

Within C/C++ system-level software, certain CWE categories appear with high frequency and carry severe consequences in safety-critical deployments. This study applies three criteria to define the target vulnerability set.

The first criterion is safety relevance within SDV foundational layers. Vulnerabilities in these categories can affect the availability and execution integrity of vehicle control systems. Buffer boundary violations can corrupt adjacent memory, including control data such as return addresses or function pointers, which creates a route to arbitrary code execution when the vulnerable code is reachable through an external vehicle interface. Unhandled null pointers can crash real-time services and reduce system availability. Detecting and repairing these flaws is therefore important for reducing cyber-physical security risk in automotive environments.

The second criterion is observed prevalence in the BigVul dataset [16]. Deep learning approaches require a substantial volume of positive examples to learn meaningful features. Weakness types that appear frequently in real-world C/C++ repositories provide a more reliable basis for classifier development and experimental evaluation.

The final criterion follows from the function-level scope of the pipeline. The detection and repair modules operate on isolated functions. Candidate vulnerability types therefore need to be identifiable primarily from local code patterns within a single routine. Categories suitable for this pipeline typically manifest as localized errors.

Based on these criteria, this study targets the following four vulnerability types [48].

- **CWE-787 (Out-of-Bounds Write)** This weakness occurs when a program writes data to a memory location beyond the allocated buffer boundary. In automotive RTOS and middleware, unbounded write operations on network-received payloads are a common pattern. They give attackers a potential path to arbitrary code execution by overwriting adjacent memory, return addresses, or function pointers.

- **CWE-476 (NULL Pointer Dereference)** This weakness occurs when a program dereferences a pointer that is expected to be valid but is NULL, typically resulting in a crash or exploitable memory access. In SDV middleware handling dynamic resource allocation or optional subsystem interfaces, missing NULL checks are a frequent source of reliability and availability failures.

- **CWE-399 (Resource Management Errors)** This category covers the incorrect management of system resources, including memory leaks, improper allocation, and mishandled file or socket handles. In long-running automotive systems, resource exhaustion caused by accumulated leaks can silently degrade or disable safety-critical services over time.

- **CWE-125 (Out-of-Bounds Read)** This weakness occurs when a program reads data from a memory location beyond the allocated buffer boundary. In automotive RTOS and middleware, out-of-bounds reads commonly arise during the parsing of variable-length fields in network frames or diagnostic payloads. An attacker can craft a malformed message to cause the software to read past the end of a buffer. Although this weakness does not directly overwrite memory, it exposes sensitive internal data such as cryptographic keys or pointer values. This leaked information can then be leveraged to bypass exploit mitigations in subsequent attack stages.

Of the four targeted categories, CWE-787 and CWE-125 share a common parent node in the CWE hierarchy. Both are subtypes of CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer, and both involve boundary violations on a memory buffer. They differ mainly in the direction of the invalid access. CWE-787 concerns out-of-bounds writes, while CWE-125 concerns out-of-bounds reads. CWE-476 and CWE-399, by contrast, occupy independent positions in the hierarchy and carry structurally distinct code signatures.

2.4 Pre-trained Code Models for Classification

Deep learning approaches for code analysis increasingly rely on the pre-train and fine-tune paradigm, established by Devlin et al. [10] with BERT (Bidirectional Encoder Representations from Transformers). In this paradigm, a model is first trained on a large and general corpus in a self-supervised manner using objectives such as masked language modelling. This allows the model to acquire broad general-purpose representations. These representations are then adapted to specific downstream tasks through fine-tuning on a comparatively small labelled dataset. This transfer learning approach is particularly valuable in the vulnerability detection domain, where labelled data is inherently scarce relative to the volume of code available in public repositories [10].

The Transformer architecture proposed by Vaswani et al. [51] underlies all three models compared in this study. Its core innovation is the self-attention mechanism, which computes relevance scores between token pairs across an input sequence. This structural property aligns with the inherent challenges of source code analysis. Memory-safety flaws in C/C++ often depend on execution context that is scattered across multiple lines of code. For example, a buffer might be allocated early in a function but accessed unsafely much later within a nested loop. While traditional sequential models generally struggle to retain representational context over such extended distances, self-attention helps mitigate this limitation. By evaluating token relationships globally within its context window, the model can more easily associate a potentially unsafe pointer operation with a distant variable declaration. This capability may help Transformer-based architectures capture some of the long-range contextual dependencies relevant to memory-safety defects.

This study compares three pre-trained code models that form a natural progression

in terms of pre-training information.

CodeBERT [18] was proposed by Feng et al. and is pre-trained on a bimodal corpus pairing natural language documentation with source code across six programming languages, including C. CodeBERT employs two complementary pre-training objectives. The first is masked language modelling over both natural language and code sequences. The second is a replaced-token detection task in which the encoder is trained to distinguish genuine code tokens from plausible substitutions generated by an auxiliary model. These objectives produce representations that capture both intra-code structural dependencies and the semantic correspondence between code and its documentation. CodeBERT processes source code as a flat token sequence without explicit structural annotations.

GraphCodeBERT [25] extends CodeBERT by incorporating explicit data-flow graph edges derived from static analysis directly into the attention mechanism. Where CodeBERT operates solely on the token sequence, GraphCodeBERT augments the input with edges that encode variable definition-use relationships. This allows the model to reason about which expression defines a variable and which expression subsequently consumes it [25]. This structural enrichment is relevant to vulnerability types such as buffer boundary violations, where the unsafe use of a variable may depend on an allocation occurring much earlier in the code.

UniXcoder [24] further extends GraphCodeBERT by incorporating Abstract Syntax Tree structure in addition to data-flow information. UniXcoder is pre-trained with a unified cross-modal pre-training objective that spans code understanding, code generation, and code-to-text tasks. The addition of syntax-level structural awareness on top of data-flow encoding incorporates the largest variety of structural information among the three evaluated models.

The three models thus form a cumulative series from CodeBERT (text only) to GraphCodeBERT (text + data flow) and UniXcoder (text + data flow + AST). All three are evaluated under both classifier architectures in this study to confirm that the architectural findings hold across pre-training strategies, and to identify the most effective detection backend for the repair pipeline, as evaluated in Chapter 6.

2.5 LLM-based Code Generation and Prompt Engineering

Recent advancements in deep learning have expanded the methodologies available for software maintenance. Xia and Zhang [53] demonstrated in their research that applying large pre-trained language models to automated program repair can yield competitive bug-fixing capabilities compared to traditional techniques. LLMs have emerged as a prominent approach in automated code generation and repair. An LLM is a neural network, typically based on the Transformer architecture, that has been pre-trained on a very large corpus of text and code to perform next-token prediction. Given a sequence of input tokens, the model assigns a probability dis-

tribution over the vocabulary for the next token. A complete output is generated by sampling tokens autoregressively, meaning one at a time, with each new token conditioned on all previously generated tokens [52]. This generation process is governed by several parameters, of which the temperature is particularly important for reproducibility. A temperature of zero selects the most probable token at each step, generally producing deterministic output. Higher temperatures introduce greater diversity and randomness.

Building upon this autoregressive mechanism, LLMs pre-trained on large code repositories acquire broad knowledge of programming patterns, idioms, and conventions. This pre-training enables them to produce syntactically valid and often semantically plausible code from a natural language or structured instruction prompt without task-specific fine-tuning [52, 57]. When applied to automated program repair, the typical workflow provides the LLM with the vulnerable function and a natural language instruction describing the repair task. The model then generates a candidate patched function as output.

Although pre-training equips these models with general coding capabilities, the actual quality of the generated patch heavily depends on how the repair task is presented. Prompt engineering refers to the practice of structuring the input provided to an LLM to obtain more accurate and reliable outputs without modifying the internal parameters of the model [52]. The simplest approach is zero-shot prompting, in which the model receives only a task instruction and the relevant input without any worked examples. Instruction prompting structures the input as an explicit and detailed task specification. This often involves assigning the model a role, specifying constraints on the output format, and decomposing the task into clearly delineated steps. This approach is widely adopted in applied LLM-based repair systems [52].

While standard instruction prompting establishes the basic task format, repairing specific security vulnerabilities often requires more precise guidance. Beyond general paradigms, *structured context injection* incorporates task-specific metadata directly into the prompt. In the vulnerability repair domain, this includes information such as the detected vulnerability type or descriptions of common fix patterns for a given weakness class. Providing the model with explicit structured knowledge may reduce ambiguity about the nature of the defect and encourage the model toward repair strategies more consistent with the identified weakness. [52, 57].

Our experimental study applies this principle to CWE-specific vulnerability repair under the term *detection-augmented prompting*. In this approach, the classifier output is injected into the LLM prompt. This context comprises the predicted CWE type and optionally includes a set of associated repair-strategy hints. This auxiliary information is intended to guide the generation process toward fix patterns appropriate for the identified weakness class. By systematically varying the level of injected context, our evaluation aims to quantify the actual impact of this structured guidance on automated repair performance. The three concrete prompting strategies designed for this experimental comparison are detailed in Chapter 5.

3

Related Work

This chapter contextualizes the experimental study by reviewing prior research in two complementary domains. The first domain encompasses deep learning techniques for vulnerability detection. The second domain involves the application of large language models to automated program repair. Reviewing the empirical findings in these areas provides the background for the specific variables and architectures explored in this study.

3.1 Deep Learning for Vulnerability Detection

Early applications of deep learning to code security explored various code representations. Zhou et al. [59] evaluated a Graph neural networks (GNN)-based detection approach on a dataset of function-level vulnerabilities extracted from several large open-source C projects. Their Devign framework constructs a composite graph that combines multiple structural edges (including Abstract Syntax Tree (AST) edges, control-flow edges, and data-dependency edges) into a single joint representation and processes it with a Gated Graph neural networks (GGNN). Devign reported accuracy improvements over prior sequence-based approaches. However, the evaluation was primarily conducted in a general binary setting of vulnerable versus non-vulnerable code without differentiating between specific CWE types.

Subsequent empirical studies observed variations in detection performance across different deep learning approaches and datasets. Chakraborty et al. [7] investigated several existing detection models and found that performance could decrease when models were assessed in different prediction scenarios. Their analysis suggested that sequence-based models might learn dataset-specific artifacts rather than generalized vulnerability features. They proposed a vulnerability detection framework named REVEAL, combining graph-based feature extraction with targeted representation learning and resampling. Their results indicated that handling data imbalance and selecting appropriate feature representations strongly influence detection reliability.

While models like REVEAL improved feature extraction, supervised approaches remain constrained by the limited availability of high-quality labelled vulnerability data. To mitigate this reliance on labelled data, recent work has increasingly applied the pre-train and fine-tune paradigm to code-level classification [18]. Empirical

studies have shown that pre-trained code models can be adapted to vulnerability detection with comparatively modest labelled data [44]. On function-level vulnerability records from BigVul [16], CodeBERT-based classifiers have also been reported to achieve detection performance comparable to conventional static analysis tools [57].

Subsequent research explored whether adding structural information can improve code representations. GraphCodeBERT incorporates data-flow information [25], while UniXcoder combines multiple code modalities, including syntax-level structure [24]. These structure-aware representations are designed to encode program relations that are absent from a plain token sequence, but they rely on external extraction steps that can be brittle for incomplete commit snippets [57]. The resulting trade-off motivates treating CodeBERT, GraphCodeBERT, and UniXcoder as an experimental variable rather than assuming that a richer representation is always preferable.

Beyond model selection, the structure of the classification architecture represents another relevant design dimension. As noted earlier, many early frameworks evaluated models without differentiating between specific CWE types. Recent empirical investigations observe that specialised single-CWE models can achieve high sensitivity to the distinct structural signatures of individual weakness classes [6]. However, the existing literature does not systematically establish whether a modular architecture of independent binary classifiers exhibits different performance trade-offs compared to a unified multi-class approach across structurally similar vulnerabilities. Because different vulnerability types can exhibit distinct code patterns and varying class-imbalance ratios, exploring this structural configuration remains an open empirical question.

Despite their detection capabilities, the discriminative classifiers discussed above function purely as analytical instruments. Their output is a prediction about the presence, location, or type of a vulnerability rather than the corrective code needed to resolve it. Bridging the gap between identifying a defect and implementing a secure patch therefore requires a complementary generative capability. This functional limitation motivates the exploration of automated program repair techniques.

3.2 LLM-Based Automated Program Repair

The generative capabilities of large language models have prompted research into their application for automated program repair. Xia et al. [53] evaluated nine pre-trained language models across five benchmarks and found that direct application without task-specific fine-tuning could produce viable patches for general software bugs. Their study also observed a scaling effect in which larger models generally achieved better repair performance.

When applying these models specifically to security vulnerabilities, researchers have observed different challenges. Sajadi et al. [46] conducted a study of LLM and agentic repair workflows on the SWE-bench benchmark, which contains real-world

GitHub issues drawn from popular open-source Python repositories. They reported instances where generated patches introduced new security vulnerabilities that were not present in the original code. Their analysis also noted that generative repair sometimes produced defect patterns uncharacteristic of human developers [46]. These findings motivate closer examination of how the generation process can be constrained and guided.

Prompt formulation is another factor in repair performance. Wang et al. [52] noted in a comprehensive survey that the structure of the repair prompt can have a measurable effect on patch quality. The survey also highlighted that many LLM-based repair systems operate with basic instruction prompting and generate patches in a single forward pass. These observations leave room to examine whether additional structured context can make vulnerability repair more targeted.

One way to add such context is to use the output of an upstream discriminative model. Fatima et al. [17] investigated this idea in flaky test repair, where a classifier predicted the fix category and the prediction was incorporated into the LLM prompt. Their evaluation reported higher patch correctness than an unguided baseline [17]. Systematic reviews in the vulnerability repair domain [57] similarly observe that auxiliary information can affect repair outcomes, although the effect may depend on the implementation and vulnerability type.

3.3 Research Gap and the Present Study

The reviewed literature indicates that deep learning models can support vulnerability classification and that LLMs can generate plausible program repairs. Studies in adjacent domains further suggest that classifier-guided prompting can influence generative outcomes.

Existing work provides comparatively limited evaluation of the specific combination examined in this thesis, where modular per-CWE detection is followed by structured vulnerability guidance in the repair prompt. The present study focuses on this combination through a controlled experimental setup. It evaluates the detection characteristics of specialised per-CWE classifiers and investigates how different levels of vulnerability guidance affect repair performance.

4

The Detection and Repair Pipeline

This chapter describes the detection and repair pipeline proposed in this study. Section 4.1 provides a high-level overview of the pipeline architecture. Section 4.2 describes the Specialised Detection Module and the design choices behind the modular binary architecture. Section 4.3 describes the Guided Repair Module and the concept of detection-augmented prompting. The experimental conditions under which the pipeline is evaluated are defined in Chapter 5.

4.1 Pipeline Overview

The proposed pipeline is a two-stage detection and repair pipeline. It takes function-level C/C++ source code as input and produces a repaired patch as output using vulnerability guidance. Figure 4.1 illustrates the overall pipeline architecture.

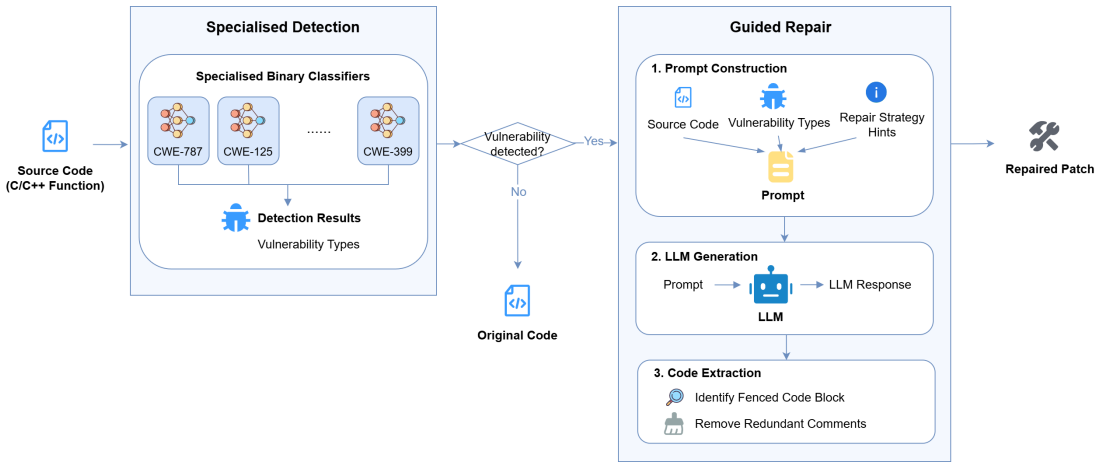

Figure 4.1: Architecture diagram of the detection and repair pipeline. The input C/C++ function is first analysed by a suite of fine-tuned binary classifiers to detect specific vulnerability types. If no vulnerability is detected, the original code is returned. Otherwise, the detected vulnerability types and corresponding repair strategy hints are injected into the repair prompt, which guides the repair module to generate a targeted patch.

The data flow proceeds as follows. A C/C++ function is passed to the Specialised Detection Module, which runs each binary classifier independently and compares

the positive-class probability for its target CWE against the corresponding decision threshold to produce a list of detected vulnerability types. If no vulnerability type is reported, the pipeline produces no repair candidate and returns the original code as its output. Otherwise, the detected vulnerability types and corresponding repair strategy hints are incorporated into the Guided Repair Module's prompt as structured guidance. The LLM generates a repaired patch, which is post-processed to extract the code block and returned as the final output.

4.2 Specialised Detection Module

The Specialised Detection Module is responsible for identifying the selected CWE-related vulnerabilities within a given C/C++ function, outputting a list of detected CWE types.

4.2.1 Modular Architecture

To evaluate its feasibility, the proposed pipeline is initially configured with a modular binary detection architecture. Under this design, one binary classifier is fine-tuned per target CWE type, and all classifiers run independently at inference. This choice is motivated by the two considerations described below.

The first consideration concerns per-type configuration. Each binary classifier can be optimised independently for its own operating conditions, including the decision threshold, loss weighting, and early-stopping criterion, without being constrained by the requirements of other CWE types. This matters because the target CWE categories differ in class balance and data characteristics, so a single shared configuration may not be equally suitable for all of them. The modular design is also extensible in that adding support for a new CWE type requires only training one additional binary classifier, with no changes to the existing components.

The second consideration concerns class separation among structurally related CWE types. CWE-787 (Out-of-Bounds Write) and CWE-125 (Out-of-Bounds Read) are both subtypes of CWE-119, the broader category of improper memory buffer restrictions. They involve similar code constructs, such as array indexing operations, pointer arithmetic, and length checks, differing mainly in whether the out-of-bounds access is a read or a write. A multiclass model trained on both must assign probability across these related labels within a single output layer. The binary design is evaluated as a way to reduce this form of inter-class competition because each classifier only decides whether its own target type is present, rather than making structurally similar labels compete in a shared output layer.

The comparison with the multiclass architecture is defined in Chapter 5 and reported in Chapter 6, where the architectural choice is evaluated rather than assumed.

4.2.2 Pre-trained Code Model Selection

The classification task in this study presents three main requirements. It requires processing source code as a sequence of tokens, outputting a probability score suitable for binary decisions, and maintaining manageable inference costs to facilitate testing across multiple vulnerability types and random seeds. Pre-trained code models based on bidirectional Transformer representations generally align well with these requirements.

The choice of these models over generative models is primarily based on their output mechanisms. While generative models are typically designed to predict subsequent tokens step by step, this classification task generally requires an overall assessment of the given code snippet. Bidirectional Transformer-based code models process the entire input concurrently and generate a contextualized representation. By adding a classification layer on top of the model's pooled representation (a mathematical summary of the sequence), a binary classifier can produce a two-class output for its target CWE. The threshold-based decision mechanism then uses the positive-class probability to decide whether the corresponding CWE label should be reported. Furthermore, the bidirectional attention mechanism allows the model to process the sequence without a strict left-to-right constraint. This characteristic helps the model capture dispersed contextual clues [23]. For instance, it can often associate a variable allocated early in a function with an unsafe operation that appears much later.

Another consideration is how these models handle structurally incomplete code. The functions in the training and evaluation datasets are extracted from real-world Git commits and frequently lack surrounding context, such as header files or forward declarations. Traditional static analysis tools that rely on strict parsing or ASTs can sometimes encounter difficulties or require substantial preprocessing when analyzing uncompilable code fragments. In contrast, the evaluated pre-trained code models process source code primarily as a sequence of text tokens, which helps reduce the dependency on successful syntactic parsing [18, 25, 24].

To identify a suitable model for this task, three candidate pre-trained code models are evaluated: CodeBERT, GraphCodeBERT, and UniXcoder, which are described in Section 2.4. Evaluating multiple models serves dual purposes. For the downstream pipeline, it supports model selection. From a methodological perspective, it helps control for potential confounding factors introduced by model selection. Observing the experimental outcomes across three distinct models reduces the risk that the findings are merely artifacts of a single model's specific pre-training strategy. As discussed in Chapter 6, UniXcoder is ultimately selected as the primary model for the downstream repair pipeline. This selection is informed by a combination of experimental observations and architectural considerations, including pipeline detection behaviour, model stability, and input-length support.

4.3 Guided Repair Module

The Guided Repair Module uses an LLM to perform automated vulnerability repair. It receives the original vulnerable source code and the detection output from the upstream detection module, constructs an augmented prompt, and returns a candidate repaired patch.

4.3.1 Detection-Augmented Prompting

Detection-augmented prompting refers to the practice of incorporating the vulnerability guidance, including vulnerability type information from the detection output of the upstream classifier and pre-defined repair strategy hints associated with each vulnerability type, into the LLM's prompt as structured cues. The rationale is that a general-purpose LLM, without explicit vulnerability type guidance, must simultaneously infer both the nature of the vulnerability and an appropriate repair strategy from the code alone. Providing the vulnerability type narrows the generative search space by telling the model *what kind* of vulnerability to repair; providing repair hints further constrains it to *how* to repair it. As noted in Section 3.2, FlakyFix [17] reported an analogous improvement in the flaky-test repair domain. The three concrete prompting strategies evaluated in this study (Strategy A: Unguided, Strategy B: Type-Guided, Strategy C: Detail-Guided) are defined in Section 5.4.2 as experimental conditions.

4.3.2 Code Extraction and Post-processing

Since LLM outputs frequently include natural language explanations alongside the generated code, a robust extraction procedure is applied to each response. The procedure searches for fenced code blocks using multiple pattern variants (including language tags such as `c`, `cpp`, and untagged fences) and extracts the content within the first matching block. Leading comments such as "`// fixed code`" are removed to obtain clean function-level code. If no fenced block is found, the entire response text is used as a fallback. Each LLM call is configured with up to three retry attempts with a linearly increasing delay between attempts to handle transient API failures.

5

Experimental Setup

This chapter describes all experimental conditions needed to reproduce and interpret the results in Chapter 6. Section 5.1 maps each research question to its corresponding experiments. Section 5.2 describes the datasets and preprocessing protocol. Section 5.3 defines the detection experiments and detection metrics. Section 5.4 defines the repair experiments and repair metrics.

5.1 Overview of Experiments

The experimental study is organised around the two research questions:

- **RQ1** → *Detection experiments* (Section 5.3): classification strategy (binary per-CWE classifiers vs. a multiclass architecture); model selection across three pre-trained code models (CodeBERT, GraphCodeBERT, UniXcoder); inference-time threshold selection.
- **RQ2** → *Repair experiments* (Section 5.4): three prompting strategies (Unguided, Type-Guided, Detail-Guided); stratification analysis based on detection accuracy (Exact Match, Noisy Match, Wrong Type).

5.2 Datasets

5.2.1 Data Sources and Selection Rationale

Dataset selection is guided by four criteria applied to candidate sources:

- **C1 (Real-world industrial relevance)**: The dataset must consist of vulnerabilities drawn from real-world open-source or industrial code rather than synthetic examples, to ensure structural complexity representative of production C/C++ systems.
- **C2 (Function-level granularity)**: Vulnerability labels must be assignable at the function level, matching the input granularity of the classifier models and the context scope of the repair module.

- **C3 (Specific CWE categorisation)**: The dataset must provide CWE-type labels rather than binary vulnerable/non-vulnerable labels, as CWE-typed annotations are required for training the specialised binary classifiers and evaluating their per-type detection performance.
- **C4 (Ground-truth fixes)**: Paired pre-fix and post-fix function code is required to serve as the reference output for repair quality evaluation.

C4 applies selectively to the test source. Training the classifier models requires labelled examples of vulnerable and non-vulnerable functions but does not depend on the availability of corresponding developer fixes. A dataset satisfying C1–C3 without satisfying C4 can therefore contribute to the training and validation pool, though not to the repair evaluation.

The candidate datasets considered are C/C++ vulnerability datasets that have been applied in the deep-learning-based vulnerability detection literature and derive from real-world open-source system software. Table 5.1 summarises how each candidate meets the four criteria.

Table 5.1: Comparison of investigated vulnerability datasets against selection criteria.

Dataset	C1: Real-world Complexity	C2: Function Level	C3: Specific CWE Labels	C4: Ground-truth Fixes
SARD [3]	×	✓	✓	×
Devign [59]	✓	✓	×	×
Reveal [7]	✓	✓	×	×
BigVul [16]	✓	✓	✓	✓
PrimeVul [12]	✓	✓	✓	×

Devign and Reveal provide only binary (vulnerable/non-vulnerable) labels and therefore do not satisfy C3. SARD consists of synthetic test cases rather than real-world vulnerabilities. Among the candidate datasets considered, **BigVul** [16] is the only one satisfying all four criteria. It is built from C/C++ project commits, provides function-level granularity, carries CWE labels validated against published CVE entries, and supplies the corresponding expert-written post-fix functions (`func_after`). BigVul is therefore used exclusively as the test source.

PrimeVul [12] satisfies C1–C3 and is combined with BigVul to form the training and validation pool, available as a pre-merged resource [37]. PrimeVul was developed with particular attention to label quality and was designed to reduce sources of annotation noise present in earlier C/C++ vulnerability datasets as reported by Ding et al. One of the filtering criteria introduced in PrimeVul is the *PrimeVulOneFunc* rule, which retains a vulnerable function only when its fix commit modifies that single function and excludes commits whose repair spans multiple functions. The merged pool provides a larger collection of labelled functions than BigVul alone, which is beneficial for CWE types where the number of positive training examples would otherwise be limited.

5.2.2 Sample Definition and Statistics

For the experiments, **one sample corresponds to one C/C++ function.** Each function is labelled with the CWE type of the commit in which it appears. A commit fixes a specific CVE, which maps to a CWE type, and all functions modified in that commit inherit that label.

The Merged pool contains 342,604 records before filtering. BigVul contains 217,007 records. Decontamination uses two filtering steps. First, records whose commit identifier appears in the BigVul test split are removed, preventing any test-set commit from appearing in training or validation data. Second, MD5 function-level deduplication removes records whose raw function text is identical to one already retained. The resulting pool of 281,966 records is used for training and validation. Training and validation samples are drawn from this decontaminated pool and test samples are drawn from BigVul.

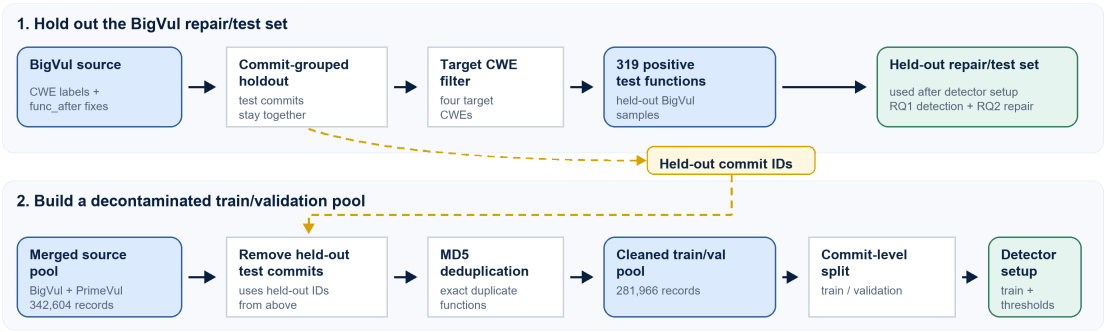

Figure 5.1: Dataset construction and split protocol for the detection experiments. BigVul test commits are held out first, and their commit identifiers are removed from the merged BigVul–PrimeVul pool before MD5 deduplication and commit-level train/validation splitting.

Table 5.2 reports the positive vulnerable function counts per split together with the size characteristics of the test-set functions. These counts do not include the negative samples used for classifier training and evaluation, which are constructed separately as described below. LOC counts non-blank lines. Token count approximates the Byte Pair Encoding (BPE) tokenisation used by all three pre-trained code models. The Test commits column reports the number of distinct BigVul commits contributing to the test split for each CWE, since the split protocol assigns whole commits rather than individual functions across partitions. The truncation column reports the proportion of test functions exceeding the 512-token input limit used by the detection classifiers.

The function size distribution varies across different CWE types. CWE-399 functions are relatively concise (median 16 LOC), which aligns with resource-release omissions typically occurring in simple, focused routines. In contrast, CWE-787 and CWE-125 functions exhibit greater length (median 50 and 64 LOC respectively), as boundary violations frequently arise within complex buffer-manipulation logic. Over one third of all test functions exceed the 512-token model input limit

Table 5.2: Positive function counts per CWE per split and size characteristics of the test-set functions.

CWE	Train	Val	Test	Test commits	LOC median	Token median	>512 tokens
CWE-787	387	132	73	24	50	368	46%
CWE-476	293	106	52	26	35	267	21%
CWE-399	423	168	83	21	16	121	11%
CWE-125	545	218	111	47	64	550	52%
Overall	1,648	624	319	118	43	329	36%

and are therefore presented to the classifier with their tail truncated. This truncation rate is most pronounced for CWE-125 (52%), a potential source of missed detections that is discussed further in Chapter 7.

For the detection experiments, each positive sample consists of the C/C++ function and its ground-truth CWE type, while the corresponding negative samples are added according to the split protocol described below. For the repair experiments, the 319 positive BigVul test functions serve as the repair inputs. Each repair sample includes the vulnerable function, the ground-truth CWE type, the expert-written ground truth fix function (`func_after`) used as the repair reference for evaluation, and a `detected_cwes` field containing the vulnerability types detected by the fine-tuned UniXcoder classifier from the detection experiments. This classifier is chosen as it achieved the highest detection rate among the evaluated models as described in Section 6.1.

5.2.3 Preprocessing and Split Protocol

Code normalisation. Raw C/C++ function text is preprocessed before use. Both multi-line and single-line comments are stripped using regular expressions, and blank lines are removed. This reduces lexical variation introduced by comments and empty-line formatting across contributors and repositories.

Decontamination. The two-step filtering procedure described in Section 5.2.2 is applied before any training or validation data are drawn from the Merged pool. The resulting cleaned pool contains 281,966 records.

Commit-level split grouping. Functions sharing a commit also share code style and vulnerability context. Assigning them to different splits would place near-duplicate samples across training and test partitions, causing the model to encounter closely related code in both. All splits are therefore constructed with commit-level grouping: every function from a given commit is assigned to the same partition. The Test commits counts in Table 5.2 describe these commit groups in the final test split, not additional test cases. The test set is drawn from BigVul by commit-grouped sampling, targeting approximately 10% of the merged pool count per CWE. The validation set is drawn from the cleaned merged pool by the same method, targeting approximately 20% of the merged pool count per CWE. The training set consists of the remaining functions in the cleaned merged pool after test and validation commits

are excluded.

Negative sample composition. Each binary classifier is trained and evaluated on datasets with a 1:2 positive-to-negative ratio. The negative pool consists of two fixed components: 70% are drawn from functions not associated with any of the four target CWE types (easy negatives), and 30% are drawn from functions associated with the other three target CWE types (hard negatives). Including hard negatives exposes each classifier to vulnerable code from related categories, encouraging it to learn representations specific to its target CWE rather than ones that merely separate vulnerable from non-vulnerable code in general. For the multiclass classifier, the non-vulnerable class is formed by sampling at a 1:1 ratio relative to the combined positive count across all four CWE types; no separate hard negative construction is applied, as all four CWE types are present as distinct labels within the same classification task.

Balanced dataset configuration. Positive sample counts vary across CWE types in the original dataset configuration. To verify that the detection conclusions are not primarily driven by differences in class size, an additional balanced dataset configuration is constructed through downsampling. For each split independently (training, validation, and test), all CWE categories are downsampled to match the positive sample count of the smallest category in that split, which is consistently CWE-476. Consequently, all CWE categories contain 293 positive samples in the training split, 106 in the validation split, and 52 in the test split. Negative samples are reduced proportionally to preserve the 1:2 positive-to-negative ratio used throughout the detection experiments. Results under both the original and balanced dataset configurations are reported in Chapter 6.

5.3 Detection Experiments

The detection experiments evaluate the three aspects of RQ1, namely classification strategy, model selection, and inference-time threshold selection. This section first describes the shared training setup, then the main training runs that compare classifier types across the three pre-trained code models, then the threshold-selection comparison, and finally two robustness checks.

5.3.1 Classifier Training Setup

Each classifier is a pre-trained code model with a binary or multiclass classification head, fine-tuned on the training split described in Section 5.2. The training configuration is shared across all combinations of model, classifier configuration, random seed, and dataset configuration. All three pre-trained code models are configured with a maximum input length of 512 tokens. This limit coincides with the architectural maximum of CodeBERT and GraphCodeBERT. UniXcoder supports longer sequences by design but is capped at 512 tokens here to maintain identical input conditions across all three models. The complete set of hyperparameters is provided in Appendix A. All reported metrics are presented as mean ± standard deviation

over the four random seeds per configuration.

5.3.2 Training Runs

The main detection comparison is conducted on the original dataset configuration and varies two factors, namely the type of classifier and the choice of pre-trained code model.

For the classification-strategy comparison, two types of classifier are trained for each model. The specialised binary configuration trains four classifiers per model, one per target CWE, with each classifier producing a probability for its target CWE over a vulnerable and a non-vulnerable class. The decision rule then compares this probability against a threshold to decide whether the target CWE label should be reported. This follows the modular design described in Section 4.2. The multiclass configuration trains one classifier per model with five output classes, namely the four target CWEs and a non-vulnerable class. It is trained under the same setup, differing only in that the probability is distributed over five classes rather than two. This configuration represents the monolithic architecture against which the modular design is compared.

For the model-selection comparison, the same classifier comparison is replicated across three pre-trained code models described in Section 2.4, namely CodeBERT, GraphCodeBERT, and UniXcoder. Replicating the comparison informs the selection of a classifier for the downstream repair module and examines whether the classification-strategy finding generalises across pre-training strategies rather than being limited to a specific model.

For each pre-trained code model, the comparison therefore contains five classifier configurations, namely four specialised binary classifiers and one multiclass classifier. Across the three models, this gives $3 \times 5 = 15$ classifier configurations. Each configuration is repeated with four independent random seeds, producing 60 training runs for the main comparison. The class-size robustness check described below repeats the same set of runs on the balanced dataset configuration, adding 60 more runs. The detection experiments comprise 120 independent training runs in total.

5.3.3 Threshold Selection

The classification threshold is the decision boundary applied, after training, to the probability a binary classifier assigns to its target CWE. If this probability is at or above the threshold, the detector reports the target CWE label; otherwise the label is not reported. The threshold does not change the trained model parameters; it only sets the operating point at which the probability is turned into a reported label, before outputs are forwarded to the repair module. Precision, recall, and F1-score then measure the quality of the reported labels. A higher threshold reports fewer labels and usually reduces false positives, while a lower threshold reports more labels and usually reduces missed detections. Two threshold strategies are compared. The fixed 0.5 threshold strategy applies the same decision boundary to

all models and CWEs and serves as the default reference. The validation-optimal threshold strategy selects, for each combination of CWE and model, the threshold that maximises validation F1, and applies this threshold unchanged to the test set. Validation F1 is used as the selection criterion to remain consistent with the headline classifier-level metric defined in Section 5.3.5; the downstream effect of this choice on pipeline-level detection rate is evaluated separately in Section 6.1.2.

5.3.4 Robustness Checks

Two robustness checks are reported alongside the main detection comparison.

The first check examines class-size sensitivity. The main detection comparison is repeated on the balanced dataset configuration, in which all three splits are down-sampled to $n = 293$, 106, and 52 positive samples per CWE in the training, validation, and test splits respectively (see Section 5.2). Comparing results across the two dataset configurations examines whether the classification-strategy ranking is influenced by class-size differences across CWE types.

The second check examines random-initialisation sensitivity. The worst single-seed binary F1-score for each CWE is compared against the corresponding multiclass mean to examine whether the classification-strategy ranking is sensitive to random initialisation.

5.3.5 Evaluation Metrics

Precision, **Recall**, and **F1-score** are the primary detection evaluation metrics. F1-score is used as the headline metric because it accounts for both precision and recall under the class-imbalanced test distribution. It is the primary criterion for comparing the binary and multiclass architectures within the classification-strategy part of RQ1.

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}, \quad \text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

All detection metrics are reported as mean ± standard deviation across four independent training runs.

Pipeline detection rate measures the proportion of the 319 BigVul test functions for which the detection module successfully identifies the ground-truth CWE type. Let y_i denote the ground-truth CWE type for function i, and let D_i denote the set of CWE types output by the detection module. The metric is defined as

$$\text{Pipeline detection rate} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[y_i \in D_i].$$

At inference, all four binary classifiers process the same input function independently. Any CWE whose probability is at or above the corresponding threshold is appended to the output list (Section 4.2). A function is counted as detected if the ground-truth CWE type appears anywhere in this list, regardless of whether other CWE types are also flagged simultaneously. Because the metric asks only whether the correct CWE is forwarded to the repair module, it does not penalise co-occurring false positives on other categories. It is therefore recall-focused at the pipeline level, rather than balancing precision and recall the way a per-CWE binary F1-score does. The metric is reported separately for the fixed 0.5 threshold and the validation-optimal threshold to quantify the practical effect of per-CWE threshold tuning.

5.4 Repair Experiments

5.4.1 Experimental Settings

The generative model for the repair module is DeepSeek-V3.2 [9], a state-of-the-art LLM developed by DeepSeek and accessed via an OpenAI-compatible API endpoint. DeepSeek-V3.2 is a Mixture-of-Experts (MoE) language model and has demonstrated strong performance in code tasks and reasoning tasks [9], making it well-suited for vulnerability repair. An alternative approach would be to use locally deployed open-source models such as Code Llama, which have full control over model weights without API dependency. However, these models require significant GPU resources for inference, which exceeded the computational budget available for this study.

All LLM calls are made with a temperature of 0.0 to reduce output variability and support reproducibility. The maximum token length is set to 8192 tokens to support full function-length responses. Up to three retries are performed for failed API calls, with increasing delay between attempts. Concurrent API calls are executed using a thread pool of up to 20 workers to reduce total experiment runtime.

5.4.2 Prompting Strategy Design

To evaluate the effect of detection-augmented prompting to answer RQ2, this study designs three prompting strategies with progressively increasing levels of vulnerability guidance. Specifically, Strategies B and C adopt the detection-augmented prompting, which utilizes vulnerability type cues to guide the generative repair process. All three strategies share a common instruction frame that specifies the task as minimal vulnerability repair while preserving the original function signature and logic, and requires the output to be a single fenced code block containing the complete repaired function.

Strategy A (Unguided): The prompt provides only the original vulnerable code and a general instruction to fix the vulnerability in the code. No information about the vulnerability type or repair strategy is included. This strategy serves as the

baseline representing conventional unguided LLM repair, where the model must independently identify both the vulnerabilities and appropriate repair strategies. The complete prompt template for Strategy A is provided in Appendix B.

Strategy B (Type-Guided): In addition to the original vulnerable code, the prompt explicitly provides the detected vulnerability type information, in the form of CWE identifiers with text description (for example, "CWE-787: Out-of-Bounds Write"). This strategy represents the core detection-augmented prompting approach of the proposed pipeline, where the detection output from the detection module is injected as vulnerability cues to narrow the generative search space.

The prompt instructs the LLM to (1) first verify whether the detected vulnerability types are present, (2) fix the identified vulnerabilities with minimal changes if present, and (3) if none of the suggested types match but other vulnerabilities exist, fix the actual issues. This three-step verification-and-repair instruction is intended to make the model robust to detection errors, by falling back to its own repair judgment if the detected types appear incorrect. The complete prompt template for Strategy B is provided in Appendix B.

Strategy C (Detail-Guided): This strategy extends Strategy B by appending CWE-specific repair strategy hints to the prompt. These hints list common repair patterns associated with the detected vulnerability types. For instance, for CWE-787 (Out-of-bounds Write), the hints include adding bounds checks before write operations, validating destination buffer sizes, and replacing unsafe functions such as `strcpy` with bounded alternatives such as `strncpy`. This strategy tests whether providing explicit repair knowledge, beyond mere vulnerability type information, further improves vulnerability repair performance. Strategy C also adopts the three-step verification-and-repair instruction used by Strategy B, ensuring that the only difference between the two strategies is the presence of the repair strategy hints. The complete prompt template for Strategy C is provided in Appendix B.

The three-strategy design enables a controlled comparison that isolates the contribution of each level of vulnerability guidance. Strategy A establishes the unguided baseline, Strategy B measures the effect of vulnerability type information injection alone, and Strategy C assesses the additional value of type-specific repair strategy hints.

5.4.3 Stratification Analysis Design

To further investigate how the accuracy of injected vulnerability type information affects repair performance, this study conducts a stratification analysis as a complement to the overall strategy comparison. The analysis focuses on the relationship between the accuracy of the detected vulnerability types and the effectiveness of detection-augmented prompting. The test samples are divided into three subsets based on whether the classifier's detected CWEs match the ground truth CWE type:

Exact Match: The detected vulnerability types contain exactly the ground truth CWE type. These samples represent cases where the classifier makes a fully correct and precise prediction.

Noisy Match: The detected vulnerability types include the ground truth CWE type along with additional incorrect CWE types. These samples represent cases where the classifier correctly detects the vulnerability type but also introduces false positives.

Wrong Type: The ground truth CWE type is not present in detected vulnerability types at all. These samples represent cases where the classifier fails to identify the correct vulnerability type.

For each subset, the repair performance change from Strategy A (Unguided) to Strategy C (Detail-Guided) is measured and compared. This comparison provides a more complete answer to RQ2 by examining the impact of vulnerability type accuracy on repair performance, as the effectiveness of detection-augmented prompting depends not only on whether vulnerability type information is provided, but also on whether the provided type information is correct.

5.4.4 Evaluation Metrics

To evaluate the quality of the vulnerability repair patches generated by the repair module, we use five metrics divided into three dimensions: Basic Code Quality, Similarity with Ground Truth, and Vulnerability Repair Quality, as summarised in Table 5.3 and detailed below.

Table 5.3: Overview of repair evaluation metrics.

Dimension	Metric	Focus
Basic Code Quality	Syntactic Validity Rate	Syntactic validity
	Static Analysis Improvement Rate	Static analysis problem
Similarity with Ground Truth	CodeBLEU	Structural similarity
	UniXcoder Similarity	Semantic similarity
Vulnerability Repair Quality	Vulnerability Pattern Removal Rate	Expected repair pattern

Basic Code Quality

This dimension checks whether the generated patch meets basic syntactic and code quality requirements.

Syntactic Validity Rate (SVR) measures the proportion of generated patches that pass a syntactic validity check. Each patch is analysed using Cppcheck [38], a widely used open-source static analysis tool for C/C++, with the `--check-fragments`

flag. A patch is marked as passing if no errors are reported. The final aggregate score is the proportion of passing samples across the test set. A high SVR indicates the LLM reliably produces code that is at least well-formed and syntactically valid.

Syntactic validity is a prerequisite for practical automated vulnerability repair, since patches that fail to parse cannot be integrated into the codebase. Samant highlights syntax validity as a fundamental diagnostic signal for code repair, noting that generated repairs may contain syntax errors that render them useless regardless of their semantic intent [47]. Cppcheck is chosen as the validation tool because it is specifically designed for C/C++ code analysis and is widely applied in academic vulnerability research and industrial practice [29, 43]. In addition, it supports fragment-level syntax validation without requiring a full compilation environment, making it particularly suitable for the function-level code in our dataset as described in Section 5.2.

Static Analysis Improvement Rate (SAIR) measures the proportion of patches for which the number of static analysis warnings does not increase relative to the original vulnerable code. For each sample, `cppcheck` is run with the `--enable=warning,style` flag on both the original code and the generated patch, and the respective warning counts are compared. A patch is marked as passing if the warning count satisfies $W_{patch} \leq W_{original}$. The final aggregate score is the proportion of passing samples across the test set. A high SAIR indicates the LLM avoids introducing new code quality problems when fixing the vulnerabilities.

Static analysis evaluation has been widely used in vulnerability repair and code generation studies to detect security issues and assess code quality. Dolcetti et al. note that LLM-generated code may contain vulnerabilities and code smells, and employ static analysis tool to detect potential safety vulnerabilities and evaluate the quality of the generated code [13]. This motivates the use of static analysis in our study, since patches generated by LLMs may introduce new code-quality issues or problematic coding patterns during vulnerability repair. We attempted to use other static analysis tools such as Clang Static Analyzer [36] and CodeQL [21], but found them unsuitable for function-level code snippet analysis because they depend on full compilation context or project-level build configuration. Cppcheck is therefore chosen because it can analyse function-level code snippets without requiring a complete build environment.

Similarity with Ground Truth

This dimension evaluates how closely the generated patch resembles the expert-written ground truth fix, from both structural and semantic perspectives.

CodeBLEU [45] measures the structural similarity between the generated patch and the expert-written ground truth fix. CodeBLEU extends the standard BLEU metric with three code-specific components: weighted n-gram match, syntactic AST match, and data-flow match. This study uses equal weights of 0.25 for all the four components, following the configuration used in FlakyFix [17]. A higher CodeBLEU

score indicates that the generated patch more closely resembles the expert-written repair in both lexical composition and syntactic structure.

CodeBLEU is included because it captures structural properties of the patches. Standard BLEU treats code as natural language text and ignores the structure of code. By incorporating AST match and data-flow match, CodeBLEU can reward patches that use the same structural patterns [45]. CodeBLEU has been widely adopted in automated vulnerability repair studies similar to our study. Liang et al.[32] and Zhou et al.[58] used CodeBLEU as a primary evaluation metric to assess patch quality in their vulnerability repair systems.

UniXcoder Similarity complements CodeBLEU by focusing on semantic similarity. It captures the semantic intent at the embedding level. Both the ground truth fix and the patch are passed through the UniXcoder model, and the cosine similarity between the resulting feature vectors is computed. Because it operates in a semantic space rather than at the text level, it can recognize similar semantic intent even when the patch uses different variable names, expressions, or alternative control-flow patterns.

UniXcoder Similarity is included for semantic similarity, because correct repairs may use implementation patterns different from the ground truth fix but still preserving the intended semantic and functionality. The cosine similarity over pre-trained code embeddings has been adopted as a reliable indicator of semantic and functional similarity in prior studies such as the vulnerability repair work by Islam et al. [28]. UniXcoder is chosen because it is pre-trained on multi-modal code data (source code, ASTs, and code comments) using pre-training objectives designed to produce semantically meaningful code embeddings [24].

These two metrics work together to provide complementary perspectives. CodeBLEU evaluates the structural similarity, while UniXcoder Similarity evaluates the semantic similarity, assessing whether the patch achieves the same semantic intent.

Vulnerability Repair Quality

This dimension directly assesses whether the generated patch corrects the underlying vulnerability.

Vulnerability Pattern Removal Rate (VPRR) is a custom metric designed to check if a patch applies the expected repair patterns for its ground-truth CWE type. For each sample, a set of CWE-specific rules is applied to the original code and the generated patch to check whether the patch introduces the expected repair patterns. The rules are implemented as regular expression or keyword checks. For example, a patch with ground-truth vulnerability type CWE-787 (Out-of-bounds Write) is checked for the introduction of bounds-aware constructs such as sizeof or strncpy. A patch is marked as passing if at least one applicable rule is satisfied, and the final aggregate score is the proportion of passing samples across the test set. VPRR directly evaluates whether the patch applies repair patterns for eliminating the targeted vulnerability types, making it a direct indicator of vulnerability repair

effectiveness.

The primary motivation for introducing VPRR is that similarity metrics are insufficient for evaluating vulnerability repair quality. Similarity metrics measure how closely a generated patch resembles the expert-written ground truth fix, but a patch can score low on similarity while still being a valid repair that takes a different repair approach than the ground truth fix. Conversely, a patch can score high in similarity while leaving the underlying vulnerability unfixed [20]. Han et al. demonstrated that match-based metrics do not reliably reflect improved functional correctness and security, as the scores overlap substantially for passing and failing patches [27]. Similarly, Dong et al. [14] pointed out that match-based metrics merely measure surface-level differences in code and fail to account for the functional correctness. VPRR is designed to complement similarity metrics by directly checking whether the patch applies the vulnerability-relevant repair patterns.

The rationale for applying CWE-specific pattern rules in VPRR is that vulnerability types are often associated with specific structured repair patterns. Canfora et al. showed that vulnerabilities of the same types are often resolved by applying similar code transformations[4]. In addition, Cao et al. demonstrated that repair patterns extracted from real-world vulnerability patches can be reused to guide vulnerability repair[5]. This suggests that vulnerability-specific repair patterns can encode the key repair actions typically required to eliminate specific vulnerability. Therefore, checking whether a generated patch satisfies CWE-specific repair patterns provides an effective way to assess whether the vulnerability has been properly mitigated.

The CWE-specific repair pattern rules used in VPRR are derived from the recommended repair patterns provided for each CWE type by the CWE framework. For example, the recommended repairs for CWE-787 (Out-of-bounds Write) include replacing unsafe unbounded functions such as `strcpy` with length-bounded alternatives such as `strncpy`, and adding boundary checks using constructs such as `sizeof` before buffer operations [41]. These patterns have recognizable syntactic signatures that keyword and regular expression matching can detect.

Conceptually, the rules capture common repair indicators for each vulnerability type. For CWE-787, the rules focus on the introduction of boundary validation, explicit pre-write validation, and the replacement of unsafe functions with bounded alternatives. For CWE-476, the rules detect the addition of null-pointer validation, memory-allocation checks, and defensive error-handling logic. For CWE-125, the rules focus on boundary validation, loop-bound verification, pointer validation, and array-index checking. For CWE-399, the rules identify resource-release operations, cleanup procedures, safe deallocation patterns, and error-path handling mechanisms that help prevent resource leaks. The full rule definitions are provided in Appendix C.

6

Results

This chapter presents the experimental results of the study, organised by research question. Section 6.1 presents the results for RQ1 (detection performance). Section 6.2 presents the results for RQ2 (repair performance). Each section closes with a direct answer to its research question.

6.1 RQ1: Detection Performance

This section presents the experimental results for RQ1. Unless otherwise noted, all reported results in Sections 6.1.1 to 6.1.2 use the original test set. Section 6.1.3 additionally presents results on the balanced dataset configuration as a robustness check.

6.1.1 Specialised Binary vs. Multiclass

This subsection examines the classification-strategy question within RQ1, namely whether dedicating an independent classifier to each CWE type yields higher per-type detection performance than training a single shared multiclass model. Table 6.1 first reports the mean test-set F1-scores for all 12 binary classifier configurations, establishing the performance range of the specialised binary architecture before it is compared with the multiclass architecture.

Table 6.1: Binary classifier test F1-score (mean ± std). Bold indicates the highest value per CWE.

Model	CWE-787	CWE-476	CWE-399	CWE-125
CodeBERT	0.565 ± 0.030	0.572 ± 0.053	0.635 ± 0.019	0.687 ± 0.040
GraphCodeBERT	**0.610** ± 0.024	**0.648** ± 0.050	**0.737** ± 0.071	0.690 ± 0.073
UniXcoder	0.608 ± 0.031	0.608 ± 0.031	0.689 ± 0.063	**0.743** ± 0.029

For the classification-strategy comparison, each binary classifier is compared against the corresponding per-class F1-score extracted from a five-way multiclass classifier trained under the same configuration. The comparison is presented in two complementary forms. Figure 6.1 summarises the 12 model-CWE comparisons in visual form. Table 6.2 reports the precise per-cell numerical values, including the binary

F1-score, the corresponding multiclass per-class F1-score, and their signed difference.

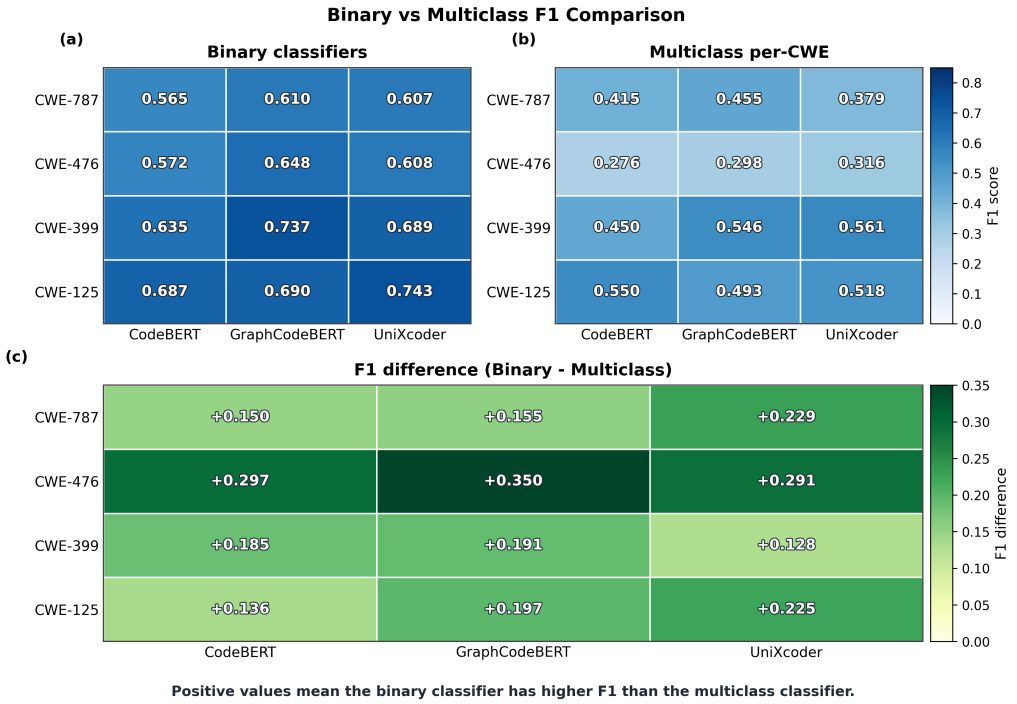

Figure 6.1: Heatmap summary of the F1-score comparison between the binary and multiclass architectures across all 12 model-CWE combinations. Panels (a) and (b) show binary and multiclass per-CWE F1-scores using the same F1 colour scale. Panel (c) reports the F1 difference (binary − multiclass), where positive values indicate higher binary-classifier performance.

Figure 6.1 summarises the main architecture comparison. Under the evaluated setting, the binary architecture achieves a higher per-CWE F1-score than the corresponding multiclass per-class F1-score in all 12 model-CWE combinations. This pattern appears for CodeBERT, GraphCodeBERT, and UniXcoder, indicating that the classification-strategy result is not confined to a single pre-trained code model. The figure does not, however, show a uniformly best model across all CWE types. For most combinations, inter-model differences remain within one standard deviation.

For context, the macro-averaged multiclass F1-scores across all five classes (the four target CWE categories plus the non-vulnerable class) are 0.505 (CodeBERT), 0.525 (GraphCodeBERT), and 0.524 (UniXcoder). These aggregate values exceed the per-CWE multiclass F1-scores in Table 6.2 because the non-vulnerable class is comparatively easier to classify.

The per-CWE numerical values in Table 6.2 show how the binary-multiclass gap varies by vulnerability type. This comparison examines whether larger gaps cluster around structurally related categories, such as CWE-787 and CWE-125, which are both boundary-violation categories under CWE-119. The observed gaps are not

Table 6.2: Per-CWE F1-score comparison between the binary architecture and the multiclass architecture (mean values across four random seeds, original test set). Each cell shows the binary F1-score followed by the corresponding multiclass per-class F1-score. Mean Δ reports the average of (binary F1 − multiclass F1) across all three pre-trained code models.

CWE	**CodeBERT** (Bin / MC)	**GraphCodeBERT** (Bin / MC)	**UniXcoder** (Bin / MC)	**Mean Δ**
CWE-787	0.565 / 0.415	0.610 / 0.455	0.608 / 0.379	+0.178
CWE-476	0.572 / 0.275	0.648 / 0.298	0.608 / 0.316	+0.313
CWE-399	0.635 / 0.450	0.737 / 0.546	0.689 / 0.561	+0.168
CWE-125	0.687 / 0.550	0.690 / 0.493	0.743 / 0.518	+0.186

concentrated in this pair. Instead, CWE-476 exhibits the largest mean Δ (+0.31), larger than the gaps for CWE-787 (+0.18) and CWE-125 (+0.19). This pattern suggests that the binary-multiclass gap is not explained only by similarity between sibling CWE categories.

Beyond the architecture comparison, the binary-only results in Table 6.1 provide two secondary observations. First, CWE-787 has the lowest mean binary F1-score across all three pre-trained code models (0.565 to 0.610). Second, CWE-125 has the highest mean binary F1-score (0.687 to 0.743), even though it also has the highest token-truncation rate among the four categories (approximately 52% of test functions exceed the 512-token limit). Chapter 7 returns to this contrast when discussing truncation and dataset composition.

Because threshold selection in Section 6.1.2 is based on validation F1, the validation and test scores are reported before the threshold results. Table 6.3 reports the mean validation-set F1-score alongside the test-set F1-score for each model and CWE, so that the validation-to-test relationship can be assessed. It also reports their signed difference $\Delta = \text{Test F1} - \text{Val F1}$.

For CWE-787, CWE-476, and CWE-125, the validation and test scores are broadly aligned. Their signed differences are relatively small in magnitude and are mostly negative, ranging from −0.127 to +0.008. This pattern indicates mild validation-to-test degradation rather than a major mismatch between the two splits.

CWE-399 differs from this pattern. All three models show positive Δ values (+0.19 to +0.27), meaning the classifiers achieve higher F1-scores on the test set than on the validation set. This behaviour is consistent across all four random seeds and all three models.

One possible explanation is that PrimeVul and BigVul construct vulnerable functions using different labelling rules. PrimeVul applies the PrimeVulOneFunc criterion proposed by Ding et al. [12]. Under this criterion, a security-related commit contributes a vulnerable function only when the fix changes a single function, which filters out commits where the repair spans multiple functions. This filter is

Table 6.3: Mean validation-set F1, mean test-set F1, and their signed difference (Δ = Test − Val) per model and CWE. Positive values indicate higher test-set performance.

CWE	Model	Val F1	Test F1	Δ
CWE-787	CodeBERT	0.648	0.565	−0.083
	GraphCodeBERT	0.737	0.610	−0.127
	UniXcoder	0.693	0.608	−0.086
CWE-476	CodeBERT	0.664	0.572	−0.092
	GraphCodeBERT	0.655	0.648	−0.006
	UniXcoder	0.658	0.608	−0.051
CWE-399	CodeBERT	0.419	0.635	+0.216
	GraphCodeBERT	0.468	0.737	+0.269
	UniXcoder	0.502	0.689	+0.187
CWE-125	CodeBERT	0.712	0.687	−0.025
	GraphCodeBERT	0.736	0.690	−0.046
	UniXcoder	0.735	0.743	+0.008

relevant for CWE-399 because resource-management fixes may involve coordinated changes across related functions. By contrast, BigVul labels all functions modified within a vulnerable commit with the corresponding CWE label. Ding et al. [12] also report label noise and overlap effects within BigVul. These dataset differences are consistent with the higher observed test-set F1-scores for CWE-399.

6.1.2 Threshold Strategy and Pipeline Detection Rate

Building on the results in Section 6.1.1, the remaining experiments use the binary classifiers exclusively and examine their behaviour as components of the detection-repair pipeline.

The classification threshold is the primary adjustable inference-time hyperparameter during deployment. Model and dataset choices are fixed at training time, whereas the threshold directly controls the precision-recall trade-off at the point where classifier outputs become inputs to the repair pipeline. A threshold set too high increases false negatives by suppressing genuine vulnerability detections, while a threshold set too low introduces additional predicted CWE labels into the repair prompt.

For the classifier-only comparison, F1-score remains the main summary metric because it balances precision and recall for each CWE. Once the classifiers are used before the repair prompt in the pipeline, however, the operational question is whether the ground-truth CWE type is forwarded to that prompt. The pipeline detection rate therefore emphasises this recall-like requirement, while the precision-recall curves show the additional false-positive trade-off introduced by lower thresholds.

Table 6.4 reports the validation-optimal thresholds alongside per-CWE test-set F1 under both strategies. Here, validation-optimal refers to the threshold that max-

imises validation-set F1 for each binary classifier. In this table, Δ denotes the F1-score change from the fixed 0.5 threshold to the validation-optimal threshold.

Table 6.4: Validation-optimal thresholds and test-set F1 under both threshold strategies per model and CWE. Δ = F1@val-opt − F1@0.5.

Model	CWE	Val-opt thr	F1@val-opt	F1@0.5	Δ
CodeBERT	CWE-787	0.30	0.658	0.547	+0.111
	CWE-476	0.40	0.615	0.578	+0.037
	CWE-399	0.15	0.663	0.671	−0.008
	CWE-125	0.30	0.740	0.683	+0.057
GraphCodeBERT	CWE-787	0.30	0.667	0.600	+0.067
	CWE-476	0.30	0.678	0.708	−0.030
	CWE-399	0.15	0.727	0.786	−0.059
	CWE-125	0.40	0.728	0.703	+0.025
UniXcoder	CWE-787	0.45	0.630	0.620	+0.010
	CWE-476	0.30	0.655	0.659	−0.004
	CWE-399	0.20	0.635	0.714	−0.079
	CWE-125	0.25	0.774	0.752	+0.022

Table 6.4 evaluates these classifier-level threshold settings using test-set F1. The F1 changes are modest for most model-CWE combinations, and CWE-399 is the only category with a consistent reduction across all three models. For this category, the validation-optimal thresholds (0.15 to 0.20) reduce test-set F1 by 0.8 to 7.9 percentage points (pp) ($\Delta = -0.008$ to -0.079). This behaviour is consistent with the validation-to-test discrepancy reported previously for CWE-399.

Broadly, all selected validation-optimal thresholds are below the fixed 0.5 threshold, which is consistent with the goal of recovering recall during validation. For CWE-399, however, these validation-selected thresholds produce lower test-set F1 than the fixed 0.5 threshold under all three models.

At the pipeline level, the relevant outcome is whether the ground-truth CWE type appears among the labels passed to the repair prompt. Table 6.5 reports this detection-rate outcome on the repair test set.

Table 6.5: Pipeline detection rate on the repair test set ($n = 319$ functions) under both threshold strategies. Δ denotes the validation-optimal detection rate minus the fixed-threshold detection rate and is reported in percentage points.

Model	Fixed thr. = 0.5	validation-optimal	Δ (pp)
CodeBERT	67.4% (215/319)	81.8% (261/319)	+14.4
GraphCodeBERT	58.9% (188/319)	80.3% (256/319)	+21.4
UniXcoder	74.3% (237/319)	**85.9%** (274/319)	+11.6

Applying these validation-optimal thresholds improves the pipeline detection rate

by 11.6 to 21.4 pp across all three pre-trained code models (Table 6.5). Under this strategy, UniXcoder achieves the highest pipeline detection rate at 85.9% (274 out of 319 functions). This table therefore reports a different property from Table 6.4. It measures whether the detector supplies the correct CWE type for the later repair stage, rather than classifier-level F1.

Figure 6.2 provides the corresponding precision-recall check. The marked operating points show that the lower validation-optimal thresholds are associated with higher recall than the fixed 0.5 threshold, while the curves show the corresponding precision differences. This pattern is consistent with the trade-off introduced above, where more predicted CWE labels can reduce missed ground-truth labels but can also add false positives. This relationship is particularly visible for GraphCodeBERT, where the pipeline detection rate increases from 58.9% to 80.3%.

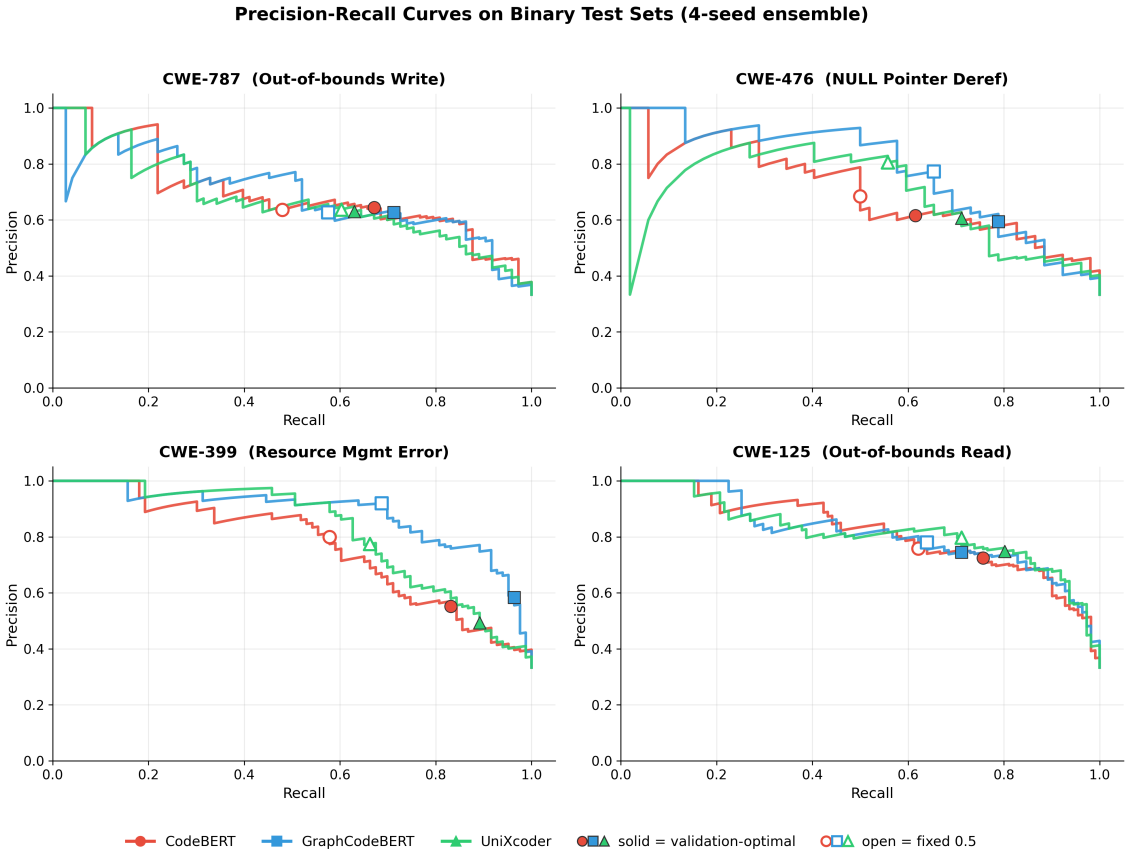

Figure 6.2: Precision-recall curves for each model on the repair test set. Colours and marker shapes distinguish the three pre-trained code models. Solid markers indicate the validation-optimal threshold, while open markers indicate the fixed 0.5 threshold.

The remaining undetected functions indicate that threshold adjustment alone does not recover all missed vulnerabilities, which represents a limitation of this approach.

6.1.3 Robustness Checks

This subsection examines whether the binary architecture's higher F1-scores reported in Section 6.1.1 hold under two alternative conditions, namely the balanced dataset configuration and independent training runs with different random seeds.

Figure 6.3 presents the binary classifier F1-scores side-by-side for the original and balanced dataset configurations. Figure 6.4 additionally shows the multiclass per-CWE F1 under the balanced configuration for direct comparison. The binary architecture rankings and the binary-versus-multiclass gap magnitudes remain broadly consistent across both configurations, suggesting that the detection conclusions are not primarily driven by class-size differences.

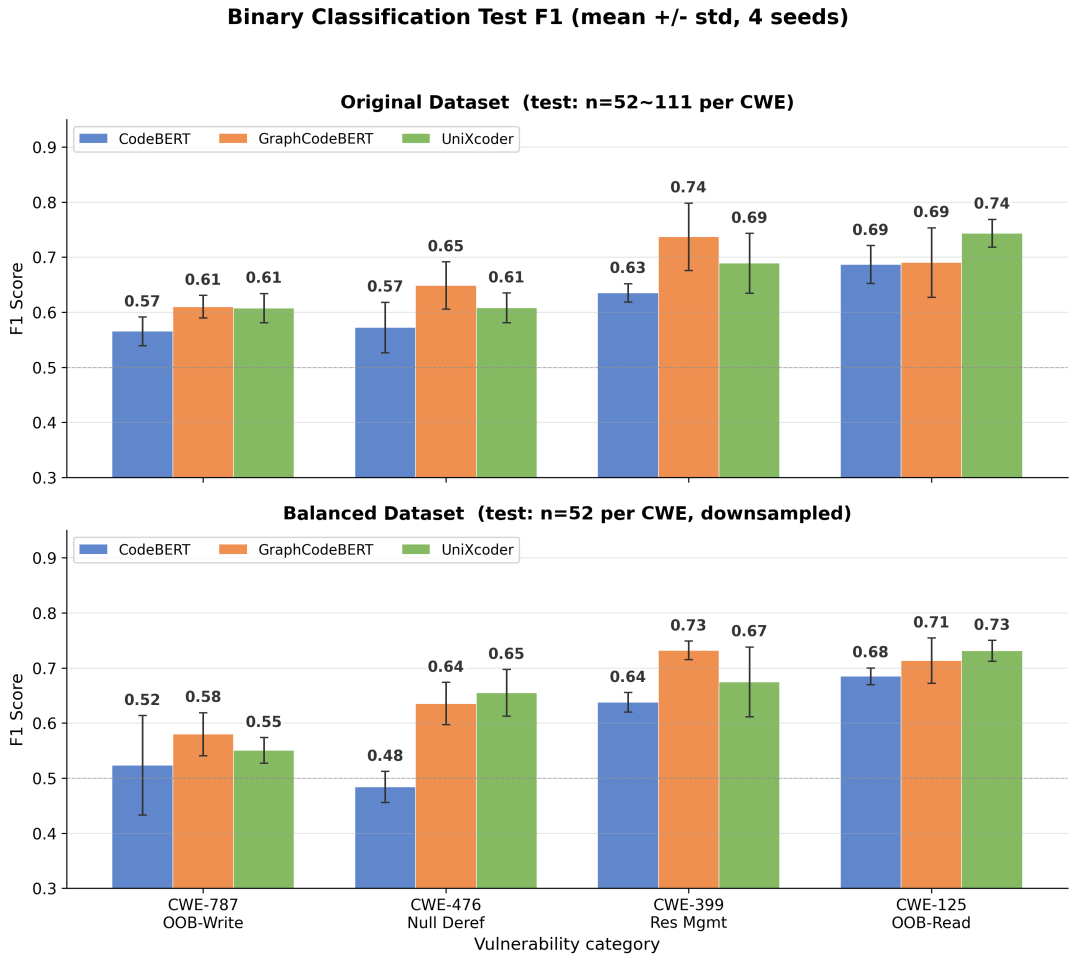

Figure 6.3: Binary classifier test F1-scores across all model-CWE combinations. The top panel shows results under the original dataset configuration and the bottom panel shows results under the balanced dataset configuration, where all CWE categories are downsampled to the same positive sample count (training: $n = 293$, validation: $n = 106$, test: $n = 52$ positive samples per CWE).

Seed-to-seed F1 variation is present across all three models and is notable for certain configurations. For example, GraphCodeBERT on CWE-399 spans 0.157 across seeds, from 0.808 with seed 100 to 0.651 with seed 512. On CWE-125, it spans 0.160,

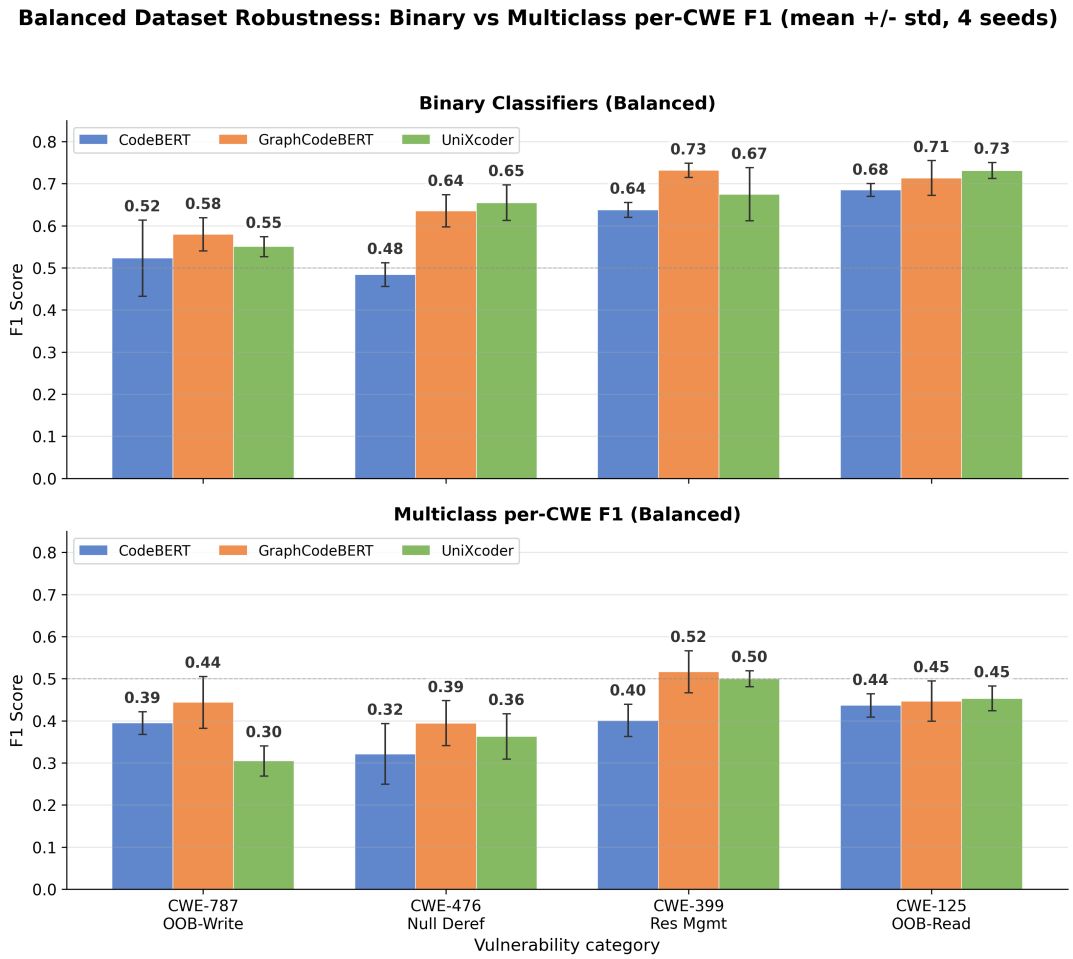

Figure 6.4: Per-CWE F1 under the balanced dataset configuration. The top panel shows binary classifier F1 and the bottom panel shows multiclass per-CWE F1, enabling direct comparison of the binary-versus-multiclass gap after class-size equalisation ($n = 52$ positive samples per CWE in the test split).

from 0.790 with seed 100 to 0.630 with seed 333. This variation may be influenced by factors such as random classification-head initialisation and the commit-grouped batch order during training. Consequently, different seeds may lead optimisation toward different regions of the loss landscape, causing early stopping to occur at different epochs.

The worst single-seed binary F1-score for each CWE also exceeds the corresponding three-model multiclass mean in every case. Specifically, the minimum binary F1-score across all seeds and models is 0.528 (CWE-787, CodeBERT, seed 2025), against a three-model multiclass mean of 0.416, a margin of +0.112. For CWE-476, the worst-case binary F1-score of 0.514 exceeds the multiclass mean of 0.297 by +0.217, the largest margin across all four types. The minimum margin across all four CWE types is +0.097 (CWE-399). The observed architecture ranking therefore appears robust across the tested random seeds. Even the single worst training run, across all models and seeds, still exceeds the average multiclass result for every targeted CWE type.

Both robustness checks support the same overall conclusion. Results on the balanced dataset configuration preserve the architecture ranking and gap magnitudes, while the worst-case single-seed binary F1-score exceeds the multiclass mean for every targeted CWE type.

> **Answer to RQ1.** The experiments indicate that classification strategy, model selection, and inference-time threshold selection all affect detection performance under the evaluated benchmark conditions. Specialised binary classifiers consistently outperformed the shared multiclass architecture, while no single pre-trained code model was uniformly strongest across the target CWE types. Thresholds selected by validation F1 changed the classifier-level F1 trade-off and increased the pipeline detection rate, although the remaining missed cases indicate that threshold tuning alone is insufficient.

6.2 RQ2: Repair Performance

This section evaluates RQ2: To what extent does detection-augmented prompting improve the performance of LLM-based automated vulnerability repair? To answer this research question, three prompting strategies with increasing levels of vulnerability guidance are compared: Strategy A (Unguided), Strategy B (Type-Guided), and Strategy C (Detail-Guided). Section 6.2.1 presents the overall repair performance results across strategies, and Section 6.2.2 further examines how detection accuracy affects repair performance.

6.2.1 Repair Performance Across Strategies

This subsection examines the repair performance of the three prompting strategies across three dimensions: basic code quality, similarity with ground truth, and vulnerability repair quality. The goal is to determine whether injecting vulnerability guidance into the prompt helps the LLM generate higher-quality repairs across these dimensions.

Table 6.6 summarises the overall repair performance of the three prompting strategies. For the similarity metrics CodeBLEU and UniXcoder Similarity, the values represent the mean and standard deviation. For other rate-based metrics SVR, SAIR, and VPRR, the values represent the proportion of samples satisfying the metric criterion.

Table 6.6: Overall repair performance across prompting strategies.

Prompt Strategy	**SVR** (%)	**SAIR** (%)	**CodeBLEU** (Mean ± Std %)	**UniXcoder Sim.** (Mean ± Std %)	**VPRR** (%)
A: Unguided	100.00	97.21	**84.34** ± 17.50	97.65 ± 4.51	28.22
B: Type-Guided	100.00	95.47	83.72 ± 17.07	**97.71** ± 4.28	45.30
C: Detail-Guided	100.00	**97.56**	83.21 ± 17.08	97.68 ± 4.28	**48.43**

Basic Code Quality

This dimension examines whether detection-augmented prompting causes any degradation in the basic quality of the generated code, such as introducing syntax errors or triggering static analysis warnings.

As shown in Table 6.6, all three strategies achieve a SVR of 100%, indicating that the LLM consistently produces syntactically valid C/C++ patches regardless of the prompting strategy used. The SAIR scores also remain high across all strategies, ranging from 95.47% to 97.56%. Although Strategy B shows a slightly lower SAIR than Strategies A and C, the difference is small. The small differences between strategies indicate that introducing vulnerability guidance generally does not degrade code syntactic or static-analysis quality.

Overall, the results for SVR and SAIR show that detection-augmented prompting preserves basic code quality. The generated patches remain syntactically valid and maintain high static-analysis quality across all prompting strategies.

Similarity with Ground Truth

This dimension evaluates whether detection-augmented prompting helps the generated patches become closer to the ground truth fixes. Two complementary similarity metrics are used. CodeBLEU measures structural similarity, and UniXcoder Similarity measures semantic similarity.

Figure 6.5 shows the CodeBLEU and UniXcoder Similarity distribution for each strategy. The CodeBLEU scores are close and slightly decrease as more vulnerability guidance is added, with mean scores ranging from 84.34% (A), 83.72% (B), to 83.21% (C). The UniXcoder similarity, on the other hand, stays almost the same across strategies: mean scores of 97.65% (A), 97.71% (B), and 97.68% (C), respectively. This divergence is further illustrated in Figure 6.6. The slight decrease in CodeBLEU under Strategy B and Strategy C suggests that more directive prompts lead the LLM to produce patches that deviate in surface-level code structure from the ground truth. However, the high and stable UniXcoder semantic similarity scores show that the patches essentially preserve similar semantic intent.

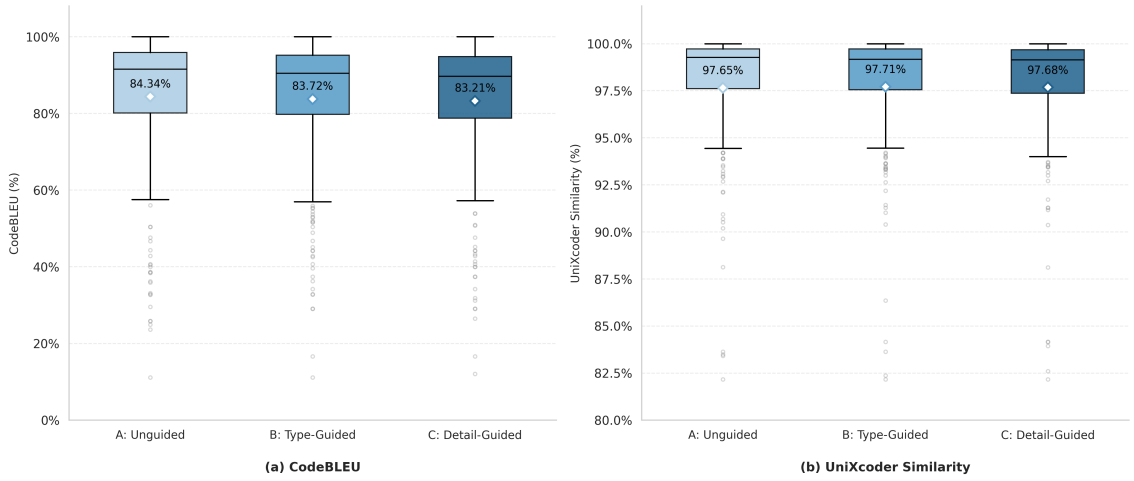

Figure 6.5: Distribution of similarity across different prompting strategies. (a) CodeBLEU distribution. (b) UniXcoder Similarity distribution. The diamond markers denote the mean values.

This difference between CodeBLEU and UniXcoder Similarity suggests that detection-augmented prompting changes the surface structure of the generated patches more than their semantic meaning. When vulnerability guidance is added, the LLM tends to generate repairs with additional validation logic, defensive checks, or control-flow changes. These modifications may reduce structural similarity with the ground truth fix, even when the repair logic remains functionally similar.

Example We use sample CWE-787_45 as the example and compare the generated

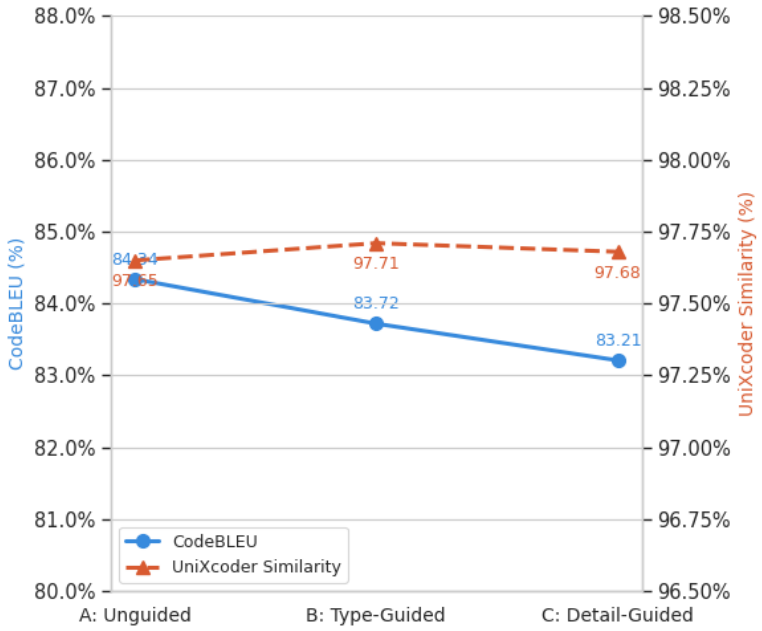

Figure 6.6: Comparison of similarity scores across different prompting strategies. The left y-axis shows the CodeBLEU, and the right y-axis represents UniXcoder Similarity. Markers indicate the mean scores for each strategy.

patches as shown in Figure 6.7. In this example, the ground truth fix removes a potential integer overflow with a minimal syntactic change, simply altering a type cast from `static_cast<size_t>` to `static_cast<unsigned>`. Under Strategy A, the LLM fails to repair the vulnerability and outputs the original code. Because the original code and the ground truth fix are nearly identical at the lexical and AST levels, Strategy A achieves a very high CodeBLEU score of 95.46%.

In contrast, Strategy B and Strategy C successfully identify the vulnerability and address the potential integer overflow through different approaches. Under Strategy B, the LLM replaces the potentially overflowing stride calculation with a call to `info.minRowBytes()`, which computes the minimum valid row byte size through a type-safe API. Under Strategy C, the targeted vulnerability guidance and repair strategy hints lead the LLM to implement an explicit defensive check to prevent overflow (`if (width > SIZE_MAX / bytesPerPixel) { return nullptr; }`). While these patches are secure and functionally correct, they introduce structural changes to the AST such as expression substitution or new variables and control-flow, causing the CodeBLEU score to drop significantly to 80.95% and 58.16% respectively. However, the UniXcoder Similarity recognizes that the semantic intent is preserved, maintaining high similarity scores of 97.54% and 94.80%.

This example illustrates why the average CodeBLEU score decreases under Strategies B and C while the UniXcoder similarity score remains stable. It suggests that the vulnerability guidance encourages the LLM to apply more extensive modifications rather than minimal edits, such as introducing additional control-flow

```
static sk_sp<SkImage> unPremulSkImageToPremul(SkImage* input) {
  SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
                                       kN32_SkColorType, kPremul_SkAlphaType);
  RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
  if (!dstPixels)
     return nullptr;
   return newSkImageFromRaster(
       info, std::move(dstPixels),
       static_cast<size_t>(input->width()) * info.bytesPerPixel());
       static_cast<unsigned>(input->width()) * info.bytesPerPixel());
 }
```

(a) Ground truth fix

```
static sk_sp<SkImage> unPremulSkImageToPremul(SkImage* input) {
  SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
                                       kN32_SkColorType, kPremul_SkAlphaType);
  RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
  if (!dstPixels)
     return nullptr;
   return newSkImageFromRaster(
       info, std::move(dstPixels),
      static_cast<size_t>(input->width()) * info.bytesPerPixel());
 }
```

(b) Patch under Strategy A

```
static sk_sp<SkImage> unPremulSkImageToPremul(SkImage* input) {
  SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
                                       kN32_SkColorType, kPremul_SkAlphaType);
  RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
  if (!dstPixels)
     return nullptr;
   return newSkImageFromRaster(
       info, std::move(dstPixels),
       static_cast<size_t>(input->width()) * info.bytesPerPixel());
       info.minRowBytes());
 }
```

(c) Patch under Strategy B

```
static sk_sp<SkImage> unPremulSkImageToPremul(SkImage* input) {
  SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
                                       kN32_SkColorType, kPremul_SkAlphaType);
  RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
  if (!dstPixels)
     return nullptr;
   return newSkImageFromRaster(
       info, std::move(dstPixels),
       static_cast<size_t>(input->width()) * info.bytesPerPixel());

   // Check for potential integer overflow in stride calculation
   size_t width = static_cast<size_t>(input->width());
   size_t bytesPerPixel = info.bytesPerPixel();
   size_t stride;
   if (width > SIZE_MAX / bytesPerPixel) {
     return nullptr; // Integer overflow would occur
   }
   stride = width * bytesPerPixel;

   return newSkImageFromRaster(info, std::move(dstPixels), stride);
 }
```

(d) Patch under Strategy C

Figure 6.7: Comparison of generated patches for sample CWE-787_45. Subfigure (a) show the ground truth fix. Subfigures (b), (c), and (d) show the generated patches under Strategy A, B, and C respectively. Highlighted areas show the code changes compared to the original vulnerable code.

branches or explicit bounds checks. In addition, this example reveals the limitation of similarity-based evaluation. A patch that is structurally similar to the ground truth fix is not necessarily a correct repair, while a structurally different patch may successfully repair the vulnerability.

Overall, the similarity results show that detection-augmented prompting does not clearly improve similarity with the ground truth fixes. Instead, it encourages the LLM to generate structurally different repairs with more extensive modifications while preserving similar semantic intent.

Vulnerability Repair Quality

This dimension evaluates whether detection-augmented prompting improves the actual vulnerability repair effectiveness of the generated patches.

As shown in Table 6.6, the VPRR improves clearly across the three prompting strategies. Strategy A achieves a VPRR of 28.22%, while Strategy B increases to 45.30%, and Strategy C further increases to 48.43%. The improvement from Strategy A to Strategy B is particularly large, with an increase of 17.08 pp. This indicates that providing the vulnerability type information helps the LLM apply appropriate repair patterns and improves the vulnerability repair quality. The further improvement from Strategy B to Strategy C (+3.13 pp) suggests that adding repair strategy hints on top of vulnerability type information provides additional smaller benefits.

Example We use sample CWE-476_90 as the example and compare the generated patches as shown in Figure 6.8. The original vulnerability is a NULL pointer dereference caused by accessing `user->passwd` without verifying whether `user` is NULL. Under Strategy A, the generated patch modifies the length parameter in the `pg_md5_encrypt()` call, which is unrelated to the actual vulnerability type CWE-476. As a result, the patch is evaluated as VPRR = False and the actual vulnerability remains unfixed.

Under Strategy B and Strategy C, the LLM is provided with the vulnerability type, NULL pointer dereference, which directly guides it to look for unguarded pointer accesses. Both strategies correctly identify `user` as the pointer that needs checking and add a NULL check for it before dereferencing it. Strategy B adds the `!user` check inline within the existing condition at the top of the function, while Strategy C inserts it as a separate validation block. Although the two patches differ slightly in code structure, they both successfully apply the expected repair pattern "added NULL check" for CWE-476 and are therefore evaluated as VPRR = True.

This example shows that without vulnerability guidance, the LLM can produce patches that look plausible but completely miss the actual vulnerability. And it illustrates how vulnerability guidance helps the LLM identify and apply the correct vulnerability-specific repair pattern, enabling more targeted vulnerability repair.

Figure 6.9 further illustrates how the effectiveness of detection-augmented prompting differs across different CWE vulnerability types. The results reveal clear differences

```
static bool check_client_passwd(PgSocket *client, const char *passwd)
{
    char md5[MD5_PASSWD_LEN + 1];
    const char *correct;
    PgUser *user = client->auth_user;

    /* disallow empty passwords */
    if (!*passwd || !*user->passwd)
        return false;

    switch (cf_auth_type) {
    case AUTH_PLAIN:
        return strcmp(user->passwd, passwd) == 0;
    case AUTH_CRYPT:
        correct = crypt(user->passwd, (char *)client->tmp_login_salt);
        return correct && strcmp(correct, passwd) == 0;
    case AUTH_MD5:
        if (strlen(passwd) != MD5_PASSWD_LEN)
            return false;
        if (!isMD5(user->passwd))
            pg_md5_encrypt(user->passwd, user->name, strlen(user->name), user->passwd);
        pg_md5_encrypt(user->passwd + 3, (char *)client->tmp_login_salt, 8, md5);
        pg_md5_encrypt(user->passwd + 3, (char *)client->tmp_login_salt, strlen((char *)client->tmp_login_salt), md5);
        return strcmp(md5, passwd) == 0;
    }
    return false;
}
```

(a) Patch under Strategy A

```
static bool check_client_passwd(PgSocket *client, const char *passwd)
{
    char md5[MD5_PASSWD_LEN + 1];
    const char *correct;
    PgUser *user = client->auth_user;

    /* disallow empty passwords */
    if (!*passwd || !*user->passwd)
    if (!user || !*passwd || !*user->passwd)
        return false;

    switch (cf_auth_type) {
    case AUTH_PLAIN:
        return strcmp(user->passwd, passwd) == 0;
    case AUTH_CRYPT:
        correct = crypt(user->passwd, (char *)client->tmp_login_salt);
        return correct && strcmp(correct, passwd) == 0;
    case AUTH_MD5:
        if (strlen(passwd) != MD5_PASSWD_LEN)
            return false;
        if (!isMD5(user->passwd))
            pg_md5_encrypt(user->passwd, user->name, strlen(user->name), user->passwd);
        pg_md5_encrypt(user->passwd + 3, (char *)client->tmp_login_salt, 4, md5);
        return strcmp(md5, passwd) == 0;
    }
    return false;
}
```

(b) Patch under Strategy B

```
static bool check_client_passwd(PgSocket *client, const char *passwd)
{
    char md5[MD5_PASSWD_LEN + 1];
    const char *correct;
    PgUser *user = client->auth_user;

    /* check for NULL user */
    if (!user)
        return false;

    /* disallow empty passwords */
    if (!*passwd || !*user->passwd)
        return false;

    switch (cf_auth_type) {
    case AUTH_PLAIN:
        return strcmp(user->passwd, passwd) == 0;
    case AUTH_CRYPT:
        correct = crypt(user->passwd, (char *)client->tmp_login_salt);
        return correct && strcmp(correct, passwd) == 0;
    case AUTH_MD5:
        if (strlen(passwd) != MD5_PASSWD_LEN)
            return false;
        if (!isMD5(user->passwd))
            pg_md5_encrypt(user->passwd, user->name, strlen(user->name), user->passwd);
        pg_md5_encrypt(user->passwd + 3, (char *)client->tmp_login_salt, 4, md5);
        return strcmp(md5, passwd) == 0;
    }
    return false;
}
```

(c) Patch under Strategy C

Figure 6.8: Comparison of generated patches for sample CWE-476_90. Subfigures (a), (b), and (c) show the generated patches under strategy A, B, and C respectively. Highlighted areas show the code changes compared to the original vulnerable code.

in VPRR improvement among the CWEs.

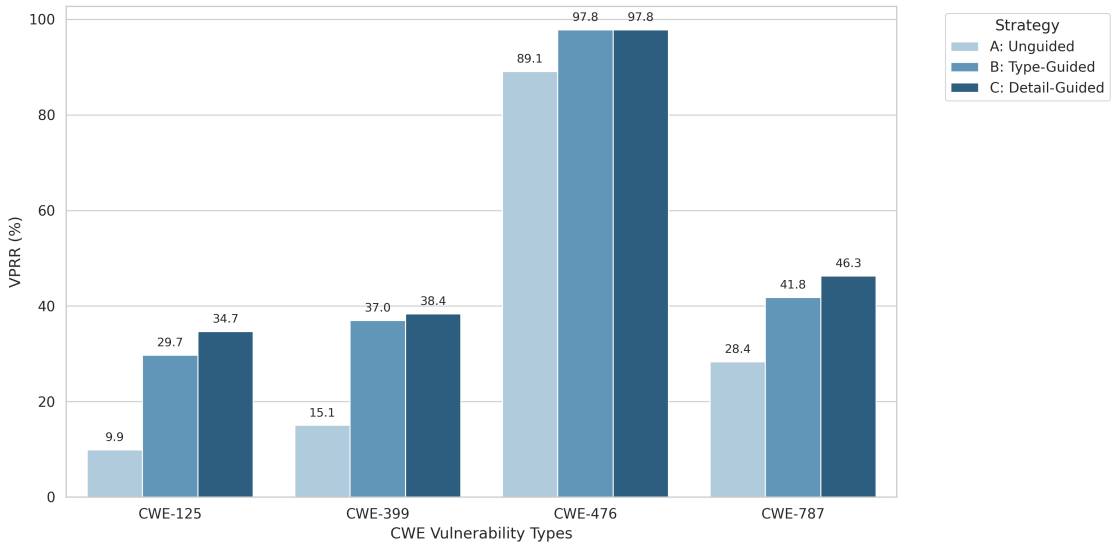

Figure 6.9: VPRR across different CWE vulnerability types. For each CWE category, the bars represent the VPRR for three prompting strategies: Unguided (A), Type-Guided (B), and Detail-Guided (C).

For CWE-476 (NULL Pointer Dereference), VPRR is already high under the unguided Strategy A (89.1%) and reaches 97.8% under Strategy B and Strategy C. This suggests that NULL pointer dereference vulnerabilities often follow standardised and recognisable repair patterns, such as adding a NULL check, which the LLM can frequently infer even without explicit vulnerability type guidance.

Example We use sample CWE-476_122 with vulnerability type CWE-476 as the example and compare the generated patches as shown in Figure 6.10. The original vulnerability is a missing NULL check on `info->attrs[NFC_ATTR_TARGET_INDEX]` before it is passed to `nla_get_u32`. Strategy A successfully adds this check by combining it with the existing attribute check at the top of the function. Strategy B and Strategy C insert the check right before the vulnerable access and return an error if the pointer is invalid. In this example, all three strategies successfully repair the vulnerability by introducing the "added NULL check" repair pattern targeted for CWE-476, achieving VPRR = True. This example explains why CWE-476 already achieves a high VPRR under unguided Strategy A. The repair pattern for NULL pointer dereference is relatively standardised such as adding a guard check on the suspicious pointer. The LLM could infer the correct fix from the code structure without explicit vulnerability type guidance.

For the other three CWE types, the unguided baseline (Strategy A) is much lower, and adding vulnerability guidance makes a clear difference. For example, CWE-399 improves from 15.1% under Strategy A to 37.0% under Strategy B and 38.4% under Strategy C. These vulnerability types typically involve more complex or varied repair patterns, with more complex contexts or less recognisable patterns. They often require understanding resource management logic, boundary constraints, or

```
static int nfc_genl_deactivate_target(struct sk_buff *skb,
                      struct genl_info *info)
{
    struct nfc_dev *dev;
    u32 device_idx, target_idx;
    int rc;

    if (!info->attrs[NFC_ATTR_DEVICE_INDEX])
    if (!info->attrs[NFC_ATTR_DEVICE_INDEX] ||
        !info->attrs[NFC_ATTR_TARGET_INDEX])
        return -EINVAL;

    device_idx = nla_get_u32(info->attrs[NFC_ATTR_DEVICE_INDEX]);

    dev = nfc_get_device(device_idx);
    if (!dev)
        return -ENODEV;

    target_idx = nla_get_u32(info->attrs[NFC_ATTR_TARGET_INDEX]);

    rc = nfc_deactivate_target(dev, target_idx, NFC_TARGET_MODE_SLEEP);

    nfc_put_device(dev);
    return rc;
}
```

(a) Patch under Strategy A

```
static int nfc_genl_deactivate_target(struct sk_buff *skb,
                      struct genl_info *info)
{
    struct nfc_dev *dev;
    u32 device_idx, target_idx;
    int rc;

    if (!info->attrs[NFC_ATTR_DEVICE_INDEX])
        return -EINVAL;

    device_idx = nla_get_u32(info->attrs[NFC_ATTR_DEVICE_INDEX]);

    dev = nfc_get_device(device_idx);
    if (!dev)
        return -ENODEV;

    if (!info->attrs[NFC_ATTR_TARGET_INDEX]) {
        nfc_put_device(dev);
        return -EINVAL;
    }

    target_idx = nla_get_u32(info->attrs[NFC_ATTR_TARGET_INDEX]);

    rc = nfc_deactivate_target(dev, target_idx, NFC_TARGET_MODE_SLEEP);

    nfc_put_device(dev);
    return rc;
}
```

(b) Patch under Strategy B

```
static int nfc_genl_deactivate_target(struct sk_buff *skb,
                      struct genl_info *info)
{
    struct nfc_dev *dev;
    u32 device_idx, target_idx;
    int rc;

    if (!info->attrs[NFC_ATTR_DEVICE_INDEX])
        return -EINVAL;

    device_idx = nla_get_u32(info->attrs[NFC_ATTR_DEVICE_INDEX]);

    dev = nfc_get_device(device_idx);
    if (!dev)
        return -ENODEV;

    if (!info->attrs[NFC_ATTR_TARGET_INDEX]) {
        nfc_put_device(dev);
        return -EINVAL;
    }

    target_idx = nla_get_u32(info->attrs[NFC_ATTR_TARGET_INDEX]);

    rc = nfc_deactivate_target(dev, target_idx, NFC_TARGET_MODE_SLEEP);

    nfc_put_device(dev);
    return rc;
}
```

(c) Patch under Strategy C

Figure 6.10: Comparison of generated patches for sample CWE-476_122. Subfigures (a), (b), and (c) show the generated patches under strategy A, B, and C respectively. Highlighted areas show the code changes compared to the original vulnerable code.

memory operation semantics. Without explicit vulnerability type information, the LLM may fail to identify the actual repair target. Therefore they benefit more from the detection-augmented prompting.

Example We use sample CWE-399_197 with vulnerability type CWE-399 as the example and compare the generated patches as shown in Figure 6.11. The original vulnerability arises because the process handles (`job_` and `worker_process_`) are terminated but never explicitly closed, leaving resources unreleased and causing resource leakage. This kind of vulnerability does not have an obvious syntactic signal with no obvious suspicious expression.

```
1  void WtsSessionProcessDelegate::Core::KillProcess(DWORD exit_code) {
2    DCHECK(main_task_runner_->BelongsToCurrentThread());
3
4    channel_.reset();
5
6    if (launch_elevated_) {
7      if (job_.IsValid()) {
         TerminateJobObject(job_, exit_code);
8        TerminateJobObject(job_.Get(), exit_code);
9      }
10   } else {
11     if (worker_process_.IsValid()) {
         TerminateProcess(worker_process_, exit_code);
12       TerminateProcess(worker_process_.Get(), exit_code);
13     }
14   }
15 }
```

(a) Patch under Strategy A

```
1  void WtsSessionProcessDelegate::Core::KillProcess(DWORD exit_code) {
2    DCHECK(main_task_runner_->BelongsToCurrentThread());
3
4    channel_.reset();
5
6    if (launch_elevated_) {
7      if (job_.IsValid()) {
8        TerminateJobObject(job_, exit_code);
9        job_.Close();
10     }
11   } else {
12     if (worker_process_.IsValid()) {
13       TerminateProcess(worker_process_, exit_code);
14       worker_process_.Close();
15     }
16   }
17 }
```

(b) Patch under Strategy B

```
1  void WtsSessionProcessDelegate::Core::KillProcess(DWORD exit_code) {
2    DCHECK(main_task_runner_->BelongsToCurrentThread());
3
4    channel_.reset();
5
6    if (launch_elevated_) {
7      if (job_.IsValid()) {
8        TerminateJobObject(job_, exit_code);
9        job_.Close();
10     }
11   } else {
12     if (worker_process_.IsValid()) {
13       TerminateProcess(worker_process_, exit_code);
14       worker_process_.Close();
15     }
16   }
17 }
```

(c) Patch under Strategy C

Figure 6.11: Comparison of generated patches for sample CWE-399_197. Subfigures (a), (b), and (c) show the generated patches under strategy A, B, and C respectively. Highlighted areas show the code changes compared to the original vulnerable code.

Under Strategy A, the generated patch replaces the handle objects with their raw `.Get()` values in the termination calls. It did not identify the actual resource leakage vulnerability and therefore fails to repair the vulnerability (VPRR = False). Under Strategy B and Strategy C, being informed that the vulnerability is a resource management error, the LLM correctly adds `job_.Close()` and `worker_process_.Close()` calls after the termination calls to release the handles. These patches successfully introduce the expected "added handle close" repair pattern for CWE-399 and are evaluated as VPRR = True.

This example shows that more complex vulnerability types such as CWE-399 benefit more from explicit vulnerability guidance because the vulnerability is not obvious and it is difficult for the unguided LLM to identify the right fix. This is a key reason why CWE-399 has a low unguided baseline VPRR of 15.1%. Once the vulnerability type is provided, the LLM can focus on the relevant aspect of the original vulnerable code and apply the correct repair pattern, which explains the large improvement from Strategy A to Strategy B (+21.9 pp) for CWE-399.

Overall, the VPRR results show that detection-augmented prompting significantly improves vulnerability repair quality. The results indicate that providing vulnerability type information helps the LLM apply appropriate repair patterns, while adding repair strategy hints on top of the type information provides a further smaller improvement. The improvement is especially clear for more complex vulnerability types, where the vulnerabilities are harder to identify and the correct repair patterns are more difficult for the LLM to infer without vulnerability guidance.

In summary, the results across the three evaluation dimensions show that detection-augmented prompting improves the repair quality of LLM-based vulnerability repair. The generated patches preserve high code quality and stable semantic similarity while clearly improving vulnerability repair quality. The results also suggest that detection-augmented prompting encourages the LLM to generate structurally different repairs with more extensive modifications.

6.2.2 Impact of Detection Vulnerability Type Accuracy

This subsection further analyses how the accuracy of the detected vulnerability type affects repair performance through a stratification analysis. This helps evaluate whether the effectiveness of detection-augmented prompting depends on the accuracy of the detection.

The test samples are divided into three subsets as defined in Section 5.4.3: Exact Match, Noisy Match, and Wrong Type. The repair performance change from Strategy A (Unguided) to Strategy C (Detail-Guided) for each subset is measured and compared. Table 6.7 presents the repair performance results across the three subsets. VPRR is excluded from this analysis because it relies on CWE-specific repair pattern rules that differ across vulnerability types, and the three subsets contain different CWE distributions. As a result, the applicable rules are inconsistent across subsets, and VPRR scores are not comparable across the stratified subsets.

For basic code quality, SVR remains at 100% across all subsets and all strategies, which means the generated patches are syntactically valid regardless of whether the injected type information is correct. Comparing Strategy A to Strategy C within each subset, the Exact Match subset shows an improvement in SAIR from 98.51% to 100.00%, while the Noisy Match subset shows a small decrease from 97.18% to 96.61%, and the Wrong Type subset shows a small increase from 95.35% to 97.67%. The differences across subsets are relatively small, and all SAIR values remain high. Overall, the code quality metrics suggest that detection-augmented prompting does

Table 6.7: Repair performance across three stratified subsets.

Subset	Prompt Strategy	SVR (%)	SAIR (%)	CodeBLEU (Mean %)	UniXcoder Sim. (Mean %)
Exact Match	A: Unguided	100.00	98.51	71.47	94.24
	C: Detail-Guided	100.00	100.00	72.77	94.68
Noisy Match	A: Unguided	100.00	97.18	89.30	98.95
	C: Detail-Guided	100.00	96.61	87.56	98.88
Wrong Type	A: Unguided	100.00	95.35	83.97	97.62
	C: Detail-Guided	100.00	97.67	81.60	97.40

not harm the syntactic or static-analysis quality of the generated patches across any of the subsets.

Beyond code quality, the similarity metrics provide a clearer picture of how detection accuracy affects repair performance. Figure 6.12 presents the similarity change ratios from Strategy A (Unguided) to Strategy C (Detail-Guided) across the three stratified subsets.

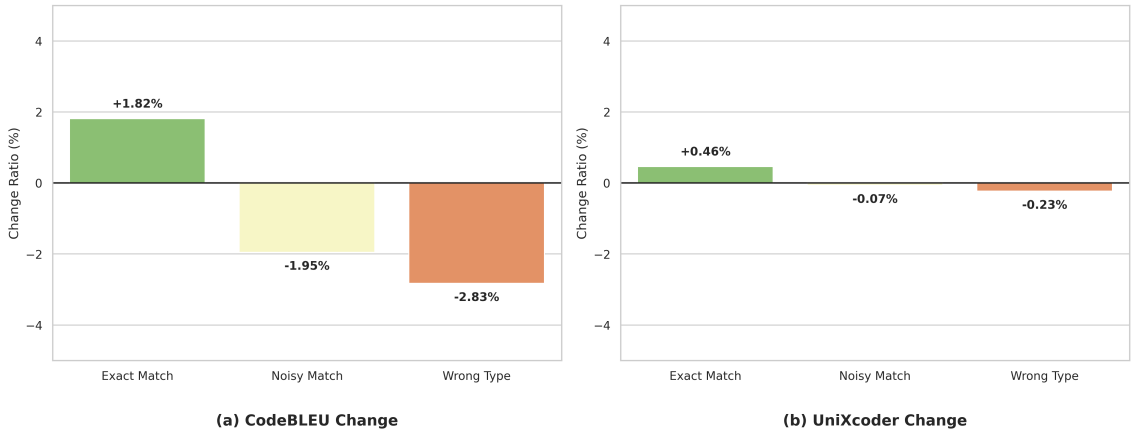

Figure 6.12: Change ratio of similarity scores from Strategy A (Unguided) to Strategy C (Detail-Guided) across three subsets stratified by the detection vulnerability type accuracy. (a) CodeBLEU change ratio. (b) UniXcoder change ratio. The change ratio is calculated as $(Sim_C - Sim_A)/Sim_A$.

For the Exact Match subset, adding detection-augmented prompting yields an improvement in CodeBLEU (+1.82%) and UniXcoder Similarity (+0.46%). This shows that when the classifiers correctly identify the vulnerability type, the detection-augmented prompting helps the LLM generate patches that are structurally and semantically closer to the ground truth fix.

For the Noisy Match subset, both metrics show a slight decrease (CodeBLEU: -1.95%, UniXcoder: -0.07%). This indicates that injecting partially correct type information has almost no negative effect compared to no guidance.

For the Wrong Type subset, similarly small negative changes are observed (CodeBLEU: -2.83%, UniXcoder: -0.23%), suggesting that incorrect type information misleads the LLM slightly.

The results suggest two observations. First, the repair performance improvement from detection-augmented prompting depends on detection accuracy. The benefit is mainly concentrated in cases where the classifier predicts the correct vulnerability type. Second, detection errors do not severely damage the repair performance. The repair remains robust to incorrect vulnerability type information, and the LLM relies on its own knowledge to generate reasonable patches when given incorrect type information.

Overall, the stratification analysis shows that correct vulnerability detection improves repair performance, while noisy or incorrect type information causes limited negative impact.

> **Answer to RQ2.** Detection-augmented prompting improves LLM-based vulnerability repair quality while preserving basic code quality. The improvement is primarily driven by vulnerability type information, and additional repair strategy hints provide smaller further benefits. The repair performance benefits most from accurate vulnerability detection, while incorrect detection causes limited degradation.

7

Discussion

The results in Chapter 6 establish what the experiments found. The aim of this chapter is to examine why. Section 7.1 examines the RQ1 detection findings by discussing the three design choices in the research question, namely classification strategy, model selection, and inference-time threshold selection. It also discusses truncation and dataset composition where they help explain the observed per-CWE patterns. Section 7.2 turns to the repair findings, examining why detection-augmented prompting improves vulnerability repair quality, the divergence between similarity and vulnerability-oriented metrics, and the impact of incorrect vulnerability guidance. Section 7.3 discusses the research and practical implications of the findings. Section 7.4 discusses the potential threats to the validity of the experimental study.

7.1 RQ1: Detection Performance

RQ1 asks how classification strategy, model selection, and inference-time threshold selection affect detection performance. The discussion below follows the same structure. Before turning to the individual design choices, it first explains why these detection results matter for software engineering in SDV contexts. It next considers why the specialised binary architecture outperforms the shared multiclass architecture. It then examines why the evaluated pre-trained code models do not form a stable ranking, and finally discusses how dataset differences may affect threshold selection and pipeline detection rate.

7.1.1 Software engineering implications for SDV vulnerability detection

The RQ1 findings can also be read in terms of software engineering risk, rather than classifier performance alone. In SDV software, C/C++ components can support communication, middleware, and control services. Where such components involve manual memory management and pointer-intensive code, they can be exposed to the kinds of memory-safety weaknesses targeted in this work. Connected vehicle interfaces can then link such software weaknesses to cyber-physical security risks [8, 11, 30]. The four targeted CWE categories therefore matter because they map to concrete failure modes discussed in Section 2.3, including buffer corruption, crashes, information disclosure, and resource exhaustion. These risks can

also carry a software engineering cost. A weakness that reaches a deployed vehicle is often costly to address. Remediation may involve large-scale recalls or coordinated software updates across a fleet, rather than a single patch, as illustrated by past automotive security incidents [40, 50]. Detecting such weaknesses earlier in development can therefore reduce this cost.

This context shapes how the RQ1 design choices should be interpreted. The detector reports which weakness type is present, and the repair stage acts on that reported type. A boundary write, an out-of-bounds read, a null-pointer dereference, and a resource-management error lead engineers toward different checks and fix patterns. From a software engineering perspective, the value of the specialised binary architecture is that it keeps these per-CWE decisions separate at the structural level. Each CWE has its own classifier, so its decision threshold and training configuration can be tuned independently, without a single shared setting having to suit all categories at once. The threshold result in this study is one use of that flexibility, where per-CWE thresholds are adjusted to control how many detected labels are forwarded to the repair stage. The modular design is also easier to extend, since support for a new CWE type can be added by training one additional classifier, leaving the existing components unchanged.

The model-selection result points to a related design property. No pre-trained code model is strongest across all CWE types, and for most categories the differences between models stay within one standard deviation, so the three models are broadly comparable rather than clearly separated. For the design of the detection component, this means the pre-trained code model is best treated as a replaceable part rather than a fixed choice. Because the models behave similarly, the selection can be guided by practical pipeline considerations, such as the resulting detection rate and the supported input length, rather than by assuming a single best model. The same modular structure that holds the per-CWE classifiers also allows the underlying model to be swapped as stronger pre-trained code models become available, without changing the rest of the pipeline.

7.1.2 Why the specialised binary architecture outperforms the multiclass architecture

One way to understand the binary architecture's consistently higher F1-scores (Table 6.2) is through the demands placed on a jointly trained multiclass model. When a single classifier must learn to detect four distinct vulnerability types simultaneously, the optimisation process must reconcile signals from classes with different structural characteristics and imbalance ratios. This explanation is consistent with earlier vulnerability-detection studies, which report that training-data composition and model representation choices can affect detection reliability on realistic datasets [7, 6]. Each CWE type also carries a different class-imbalance ratio in the training pool, with positive samples ranging from 293 for CWE-476 to 545 for CWE-125, against a shared negative pool that is substantially larger. The modular binary design avoids this shared optimisation setting by training one classifier per CWE

type, allowing each model to be tuned to the imbalance and structural properties of its target class. This places the result in contrast with common multiclass vulnerability classifiers, where one model predicts a label from a shared set of vulnerability classes [61, 19, 54]. It does not mean that binary training is generally superior, but it provides a possible explanation for why it performs better under the evaluated conditions.

This difficulty may become more pronounced for vulnerability types that appear across a wide range of syntactic contexts, as illustrated by the CWE-476 result. The binary-versus-multiclass performance gap for CWE-476 is noticeably larger than the gaps observed for the two boundary-violation sibling types, CWE-787 and CWE-125. This contrast is informative because CWE-787 and CWE-125 are related through the CWE-119 boundary-violation family, while CWE-476 is a separate weakness centred on missing pointer validation [42]. If confusion between related CWE types were the main reason for the multiclass drop, one might expect larger gaps for the two types sharing the CWE-119 parent node. The data do not strongly support this pattern. One possible interpretation is that the syntactic diversity within CWE-476 itself contributes to the gap. Null-pointer dereferences arise in a wide range of code contexts, including loop conditions, function arguments, struct-field accesses, and return assignments. As a result, positive examples for CWE-476 can differ substantially from one another, which may increase the difficulty of learning a shared multiclass model across vulnerability categories while simultaneously modelling the other vulnerability types. A dedicated binary classifier instead faces a narrower task focused on distinguishing the presence or absence of null-pointer dereference patterns within a single target class.

7.1.3 Why no single model dominates across all CWE types

The absence of a dominant model across all four CWE types (Table 6.1) is consistent with the possibility that different vulnerability types rely on different kinds of code information for reliable identification. This is in line with prior observations that vulnerability-detection results can change across datasets and experimental settings, so a model that performs well in one configuration does not necessarily remain strongest in another [7, 6]. In this study, a similar pattern appears at the per-CWE level within the same experimental setup. For most model-CWE configurations, inter-model differences fall within one standard deviation of the adjacent model's result, suggesting that the three pre-trained code models are broadly comparable on this benchmark rather than clearly separated in performance.

The three pre-trained code models differ primarily in the kinds of structural information they make available to the classifier. CodeBERT represents code primarily as a token sequence [18], GraphCodeBERT incorporates data-flow relationships, while UniXcoder additionally integrates abstract syntax tree structure [25, 24]. These architectural differences may influence how effectively each model represents different vulnerability patterns. Boundary violations such as CWE-787 and CWE-125 often involve relationships between memory operations and variables or bounds established elsewhere in the function, whereas patterns associated with CWE-476 are

often expressed within more local regions of the code. As a result, some vulnerability types may be better represented by explicit structural information, while others may remain comparatively well served by sequence-oriented representations alone.

7.1.4 Truncation effects and the unexpectedly strong CWE-125 performance

The truncation statistics in Table 5.2 present a seemingly counterintuitive pattern. CWE-125 functions are truncated at the highest rate among the four types (52% exceed the 512-token limit), yet the CWE-125 classifiers achieve the highest mean F1-scores across all three models (0.687 to 0.743, Table 6.1). CWE-787 functions are truncated at a comparable rate (46%), yet the corresponding classifiers achieve the lowest mean F1-scores (0.565 to 0.610). Because the evaluated encoder models are capped at 512 tokens in this experiment, truncation can remove code that may be useful for classification. This concern is supported by prior work on long-range context modelling for vulnerability detection, which reports that limited input length prevents Transformer-based models from capturing the extended data-flow and control-flow dependencies that some vulnerabilities depend on [33]. However, the truncation rate alone does not show whether the removed code contains vulnerability-relevant information. The contrast between CWE-125 and CWE-787 should therefore be read cautiously, since it suggests that the truncation rate alone cannot explain the per-CWE performance pattern. The next two paragraphs discuss two further differences between the types that may help explain this contrast.

The first is training set size. CWE-125 has the largest positive training set of the four types (545 samples, Table 5.2), compared with 387 for CWE-787. This is relevant because earlier detection studies discuss training-data composition as a factor that can affect whether vulnerability-related features are learned reliably [7]. A larger training set may provide broader exposure to the range of forms that out-of-bounds read vulnerabilities can take, which may partially offset the impact of truncated inputs during classification. The smaller CWE-787 training set may leave certain out-of-bounds write patterns less consistently represented, potentially increasing the classifier's sensitivity to truncated inputs.

The second difference concerns structural similarity between the two types. CWE-787 and CWE-125 are sibling categories under CWE-119 in the MITRE CWE hierarchy, and both involve memory-buffer boundary violations [42]. When training the CWE-787 binary classifier, CWE-125 samples appear as part of the negative set, meaning the classifier must distinguish between structurally similar positive and negative examples. This increased class-boundary ambiguity, independent of truncation, may contribute to the comparatively lower CWE-787 performance. A reciprocal challenge also exists for the CWE-125 classifiers, since CWE-787 samples likewise appear in their negative set. However, the larger positive training set available for CWE-125 may partially offset this difficulty by providing broader coverage of out-of-bounds read patterns during training.

Together, these observations suggest that training-set size and inter-type structural

similarity may help explain why CWE-125 remains strong despite high truncation, while CWE-787 remains weaker at a comparable truncation rate. This does not rule out truncation effects, but it indicates that the truncation rate is not sufficient as a standalone explanation. The 52% truncation rate for CWE-125 nonetheless represents a potential limitation of the current pipeline. UniXcoder supports input sequences of up to 1024 tokens by design [24] but was capped at 512 tokens here to keep conditions identical across all three models. Where vulnerability-relevant code falls beyond the truncation boundary, the classifier has no access to that information during inference. Evaluating UniXcoder at its full 1024-token capacity is therefore identified as a targeted extension in Chapter 8, with CWE-125 representing a plausible candidate for further improvement given its comparatively high truncation rate.

7.1.5 The CWE-399 validation anomaly and its implications for threshold selection

The consistently positive validation-to-test Δ values for CWE-399 (+0.19 to +0.27 across all three models, Table 6.3) present an inverted generalisation gap. Rather than indicating substantially better generalisation to the test set, this pattern is more plausibly explained by differences in dataset construction and labelling criteria between the validation and test sources. This interpretation is consistent with dataset-focused vulnerability-detection work, which reports that label quality and split construction can affect measured performance [12].

The merged validation pool includes records from PrimeVul, which applies the PrimeVulOneFunc criterion and labels a function as vulnerable only when it is the *sole* function modified by a security-related commit [12]. This stricter filtering strategy is particularly relevant for resource-management vulnerabilities (CWE-399), where fixes often involve modifications across multiple related functions. As a result, the validation split may contain fewer recurring structural patterns for this vulnerability category, making the classification task comparatively more difficult and potentially contributing to the lower observed validation F1-scores.

By contrast, the BigVul dataset used for testing adopts a broader labelling strategy in which all functions modified within a vulnerable commit inherit the corresponding CWE label. Ding et al. [12] further report that BigVul contains substantial label noise and measurable overlap effects between splits. Under these conditions, classifiers may achieve higher apparent performance on the BigVul test set than would be expected from the stricter validation distribution. This interpretation is consistent with the inverted validation-to-test gap observed for CWE-399 across all three models.

The same behaviour also affects threshold selection. Because the validation-optimal thresholds are selected by maximising validation F1, the comparatively low validation scores for CWE-399 steer the search towards more permissive operating points (0.15 to 0.20, Table 6.4) in an attempt to recover recall. When applied to the BigVul test set, these lower thresholds make the classifiers assign positive labels more often

and reduce test-set F1 relative to the fixed 0.5 threshold across all three models. At the same time, pipeline detection rate still improves (Table 6.5), because this metric counts whether the ground-truth CWE type is included in the detector output and does not penalise additional predicted labels. In this pipeline, those additional labels are not the final result. They are passed to the repair stage, so Section 6.2.2 examines repair behaviour under exact, noisy, and wrong guidance.

The case illustrates the importance of interpreting cross-dataset validation results cautiously when evaluation splits are derived from datasets with different filtering and annotation policies. For threshold selection, this case suggests that validation-optimal thresholds should be interpreted together with the validation source, especially when the validation and test splits are built under different dataset rules.

7.2 RQ2: Repair Performance

7.2.1 Software engineering implications for SDV vulnerability repair

The results suggest that vulnerability repair should be viewed as a software engineering activity rather than only a code generation task. The repair module achieved better results when it received structured vulnerability guidance from the detection module. This indicates that repair quality not only depends on underlying model capabilities but also on how related security knowledge is incorporated into the repair workflow.

The results also reflect implications for pipeline architecture design. The proposed pipeline separates the vulnerability repair task into two stages. The detection module provides vulnerability information and the repair module generates repaired patch. This separation allows each module to address a focused task. It also supports maintainability and extensibility. The modular design allows new vulnerability types to be incorporated through new detection classifiers without changes to the overall pipeline structure. This can simplify future vulnerability repair system evolution as SDV security requirements change.

The pipeline also adopts a fallback mechanism to handle detection errors. The prompt instructs the LLM to first verify whether the detected vulnerability types are present, and to use its own judgment if they do not appear correct. This robustness to detection errors is useful for a practical pipeline, because no classifier can be expected to produce correct predictions for every input. A repair module that remains stable under incorrect guidance can reduce the risk of generating misleading patches in safety-critical SDV code.

7.2.2 Why detection-augmented prompting improves vulnerability repair quality

The VPRR results in Section 6.2 show a clear improvement when vulnerability guidance is added to the prompt. Strategy A achieves a VPRR of 28.22%, while Strategy B raises this to 45.30% and Strategy C to 48.43%.

To understand why this improvement occurs, we can consider the repair task itself. Without any vulnerability guidance, the LLM must simultaneously identify what kind of vulnerability is present and decide how to repair it. This places a high demand on the LLM to correctly identify the vulnerability from the vulnerable code. The unguided LLM may fail to identify the actual vulnerability and produce a patch that modifies something unrelated to the actual vulnerability.

Providing the vulnerability type information removes the identification step from the LLM's task. This narrows the generative search space of the LLM. The LLM no longer needs to guess what kind of vulnerability is present. It can instead focus on locating the specific pattern associated with that vulnerability type and applying the appropriate repair. In addition, the large increase from Strategy A to Strategy B (+17.08 pp) compared to the smaller increase from Strategy B to Strategy C (+3.13 pp) indicates that the vulnerability type information is the primary driver for the vulnerability repair quality improvement. The vulnerability type information provides the most useful guidance, and the additional repair strategy hints provide a further refinement. This finding is consistent with related studies that use vulnerability-related context in the prompt. Khan et al. showed that providing the LLM with the exact CWE details as additional security context can significantly improve the repair success rate [31]. In addition, the role of repair strategy hints is also supported by a pattern-aware vulnerability patch generation approach PailGen, which integrates relevant fix patterns into the prompt to provide additional contextual information for the LLM [49].

The results across different vulnerability types in Figure 6.9 also supports this analysis. CWE-476 starts with a high unguided baseline of 89.1% because NULL pointer dereferences tend to follow a standardised and recognisable repair pattern. The model can often identify the unguarded pointer access without being told what to look for. The remaining three CWE types involve more complex contexts or less recognisable patterns. It makes unguided identification harder and benefit more from explicit vulnerability type information.

7.2.3 Why similarity metrics and vulnerability-oriented metrics show different trends

The results in Table 6.6 show a divergence between the similarity metrics and the vulnerability-oriented metric VPRR. CodeBLEU decreases slightly as more vulnerability guidance is added, from 84.34% under Strategy A to 83.21% under Strategy C. VPRR moves in the opposite direction, increasing by 20.21 pp from Strategy A to Strategy C. UniXcoder Similarity remains stable across all three strategies.

Similarity metrics and vulnerability-oriented metrics evaluate different aspects of vulnerability repair. CodeBLEU rewards patches that are structurally close to the ground truth fix. Ground truth fixes written by developers tend to be minimal edits, changing only what is strictly necessary to resolve the vulnerability. When the LLM receives explicit vulnerability guidance, it is directed toward specific vulnerability types and associated repair patterns. This focus encourages it to apply a broader set of recommended defensive measures. These may include explicit bounds checks, additional NULL guards, or new control-flow branches. The generated patch may differ in surface structure from the ground truth fix, which reduces the CodeBLEU score even when the patch is a valid and more complete repair. Furthermore, there is often more than one correct way to fix the vulnerabilities. Different repair implementations can successfully fix the vulnerability while using different code structures. This is pointed out by prior study by Dong et al., which notes that different codes can perform identical operations, but match-based similarity metrics primarily measure the surface differences between codes and do not consider their functional equivalence [14]. The relatively stable UniXcoder Similarity scores further suggest that the generated patches preserve similar semantic intent despite these structural differences. By contrast, VPRR focuses on whether the generated patch applies vulnerability-relevant repair patterns. As a result, a patch can achieve a relatively low CodeBLEU score while still applying the correct repair patterns for the target vulnerability.

These observations suggest that similarity metrics alone are insufficient for evaluating vulnerability repair quality. A patch can score lower on CodeBLEU while being a more complete repair. Conversely, a patch that closely resembles the ground truth fix in structure may leave the underlying vulnerability unaddressed. The findings are consistent with prior studies on vulnerability repair evaluation. Germano et al. noted that a patch can still receive a high CodeBLEU score while missing the fix [20]. The study by Han et al. also revealed the limitation of similarity metrics. Their findings show that passing and failing patches exhibit substantial score overlap, which indicates that match-based similarity metrics do not reliably reflect functional correctness and security [27].

7.2.4 Impact of incorrect vulnerability guidance on repair performance

The stratification analysis in Section 6.2.2 examines how incorrect vulnerability guidance affects the repair performance. The Wrong Type subset shows a CodeBLEU change of -2.83% and a UniXcoder change of -0.23% from Strategy A to Strategy C, indicating negative but limited impact on repair performance.

This suggests that incorrect vulnerability guidance can mislead the repair process. When the injected CWE types do not correspond to the actual vulnerability, the LLM may focus on an irrelevant security issue and apply unsuitable repair patterns. As a result, the generated patch may include unnecessary modifications or fail to repair the actual vulnerability. A similar effect has been observed in prior LLM-

based generation work. When the model is given misleading information, it performs worse than the baseline without any additional information [55].

On the other hand, the negative impact is limited and it suggests that the repair is robust to detection errors. This can be explained by the three-step verification instruction included in Strategies B and C. The prompt instructs the LLM to first check whether the suggested vulnerability types are actually present in the code, and to fall back to its own repair judgment if they do not appear to match. When given incorrect vulnerability types, the LLM could recognise that the suggested types are not consistent with what it observes in the code and proceed with its own judgment instead. This verification and fallback mechanism reduces the damage that detection errors can cause. This mechanism reflects a principle discussed in LLM applications. When the provided context is noisy or conflicting, a model that can fall back on its own knowledge is considered more robust to misleading information [34].

Overall, the results indicate that the repair module is robust to the detection errors. Incorrect vulnerability guidance leads to only limited degradation of repair performance. From a practical perspective, this is important because real-world vulnerability detection classifiers inevitably produce some detection errors. The findings suggest that the proposed pipeline can benefit from correct detections while remaining resilient to the detection errors.

7.3 Implications and Lessons Learned

7.3.1 Implications for Researchers

The detection results suggest that classifier architecture remains an important design factor in vulnerability detection. The specialised binary architecture keeps training and threshold configuration separate for each CWE type, which allows each classifier to be tuned independently. The specialised binary architecture consistently achieved stronger performance than the multiclass architecture, while no single model is consistently strongest across all CWE types. This indicates that future studies could benefit from continue investigating architectural choices rather than focusing only on model selection.

The repair results suggest that structured vulnerability guidance can guide LLM-based repair toward more appropriate repair patterns and improve repair quality. The results also reveal limitations of relying on similarity metrics to evaluate vulnerability repair. A repair patch may take a different repair approach than the ground truth fix and still be a valid repair. So evaluating patches only by similarity metrics would not reflect the actual vulnerability repair improvement. This suggests that vulnerability-oriented metrics could serve as a complement to similarity metrics when assessing repair quality. And studies could consider multiple evaluation perspectives to capture different aspects of vulnerability repair quality.

The findings highlight the value of studying vulnerability detection and repair as

connected stages rather than independent tasks. The stratification analysis indicates that upstream detection accuracy impacts downstream repair quality. This relationship suggests that studies designing vulnerability detection and repair pipelines may benefit from considering the two stages jointly and evaluating the complete pipeline workflow.

7.3.2 Implications for Practitioners

The findings also provide implications for practitioners developing and maintaining security-critical SDV systems. The proposed pipeline provides a workflow for vulnerability detection and repair. The detection module identifies vulnerability types, and the repair module generates candidate patches that developers can review and apply. This approach does not completely replace expert judgement, but it may reduce the effort in the initial stages of vulnerability analysis and repair.

The findings further suggest that detection accuracy affects downstream repair quality. Better detection contributes to better repair performance. Therefore practitioners deploying a detection and repair pipeline could treat the accuracy of the detection stage as an important factor in overall effectiveness and make efforts to improve it.

For practitioners considering whether to integrate a similar pipeline into an SDV development workflow, the repair robustness result is directly relevant. The stratification analysis shows that incorrect detection does not cause severe degradation in repair quality. This means the pipeline does not require a perfect detector to provide value in practice.

Regarding detection model selection, the results indicate that no single model is consistently strongest and the differences are mostly small. Thus the model selection can be guided by practical considerations such as supported input length and inference cost, rather than by assuming one model will always perform best.

7.3.3 Lessons Learned

The first lesson from this study concerns dataset construction. Vulnerability datasets often involve trade-offs between dataset size, vulnerability coverage, and data quality. In this study, we addressed this by combining BigVul and PrimeVul datasets to construct a dataset that is large enough and meet our requirements. However, combining datasets from different sources also introduced challenges. The two datasets were built under different filtering and labelling strategy, which introduced the unexpected CWE-399 validation anomaly discussed in Section 7.1.5. This experience suggests that combining complementary datasets can be a practical approach when no single dataset provides sufficient size or coverage. But the differences in dataset construction should be carefully examined.

The second lesson concerns experiment design. This study evaluated several factors that could affect pipeline performance, including classification architecture, pretrained code model, classification threshold, and prompting strategy. We found it

important to change one variable at a time and keep others fixed, so that the observed differences in results could be attributed to the chosen factor rather than a combination of factors. For example, the three prompting strategies adopt incremental design and the only difference is the level of vulnerability guidance. For example, we configured the maximum input length of 512 tokens for all three detection models, even though UniXcoder supports longer input. This made the comparison controlled and interpretable.

The third lesson concerns the choice of evaluation metrics. For repair evaluation, we initially planned to use CodeBLEU and UniXcoder Similarity as the primary metrics. During the study, we found that these similarity metrics did not always reflect whether the generated patch addressed the targeted vulnerability. This motivated us to introduce VPRR as a complementary metric. It directly checks for vulnerability-relevant repair patterns. However, VPRR addresses the limitation of similarity metrics, but it relies on pre-defined pattern rules and therefore introduces its own limitations and potential threats to validity. This experience highlights that it is important to consider both the strengths and limitations of evaluation metrics. And no single metric is sufficient on its own. Combining different metrics captures different aspects and provides a more complete assessment of repair performance.

7.4 Threats to Validity

7.4.1 Internal Validity

The detection experiments use a maximum input length of 512 tokens for all three pre-trained code models. This setting keeps the model comparison controlled, since 512 tokens is the architectural limit for CodeBERT and GraphCodeBERT, while UniXcoder is capped to the same length for comparability. However, longer functions are truncated before classification, so vulnerability-relevant statements near the end of a function may be unavailable to the detector. This threat is most relevant for CWE-787 and CWE-125, where 46% and 52% of the test functions respectively exceed the 512-token limit. The truncation rates are reported in Section 5.2, and the impact of truncation is discussed in the RQ1 analysis, but the results should still be interpreted as applying to the 512-token setting used in the experiment.

The repair experiments use the detected CWE labels from the UniXcoder classifier from the detection experiments. This means that the detection accuracy of the classifier directly affects the vulnerability type information injected into the repair prompt, and thus affects the repair performance. The stratification analysis in Section 6.2.2 partially accounts for this by separating test samples into subsets with different levels of detection accuracy and measuring repair performance within each subset.

The observed differences between prompting strategies could be attributed to confounding factors in the prompt design rather than to the vulnerability guidance itself. To mitigate this threat, the three prompting strategies share a common instruction

frame and only differ in the level of vulnerability guidance provided. Strategy B adds the detected CWE types on top of Strategy A, and Strategy C further adds repair strategy hints on top of Strategy B. This incremental design ensures that the observed differences between strategies can be attributed to the added vulnerability guidance rather than to other changes in prompt.

7.4.2 External Validity

The experiments are conducted on the BigVul and PrimeVul datasets, which consists of code from open-source C/C++ projects. These projects share characteristics relevant to SDV foundational softwares, such as pointer-intensive code and low-level memory management. However, they are not production SDV codebases and the extent to which the results generalise to SDV software remains open.

7.4.3 Construct Validity

The VPRR metric is based on predefined CWE-specific repair pattern rules implemented as regular expression and keyword checks. These rules check for the presence of expected repair patterns in the generated patch. A patch can satisfy these rules without actually eliminating the underlying vulnerability, for example by introducing the correct pattern in the wrong location. Conversely, a valid repair that takes a different approach than the patterns covered by the predefined rules would be marked as failing. The rules are intended to serve as an approximation of vulnerability repair effectiveness rather than a formal correctness criterion, and their coverage is limited to the predefined repair patterns.

The repair evaluation metrics used in this study measure basic code quality, structural and semantic similarity, and the presence of expected repair patterns. None of these metrics directly verify whether the generated patch passes functional tests or actually eliminates the vulnerability at runtime. A patch that scores well on these metrics may still fail to correct the underlying vulnerability in practice.

The stratification analysis in Section 6.2.2 excludes VPRR because the three stratified subsets defined by detection accuracy contain different CWE distributions, making the CWE-specific pattern rules incomparable across subsets. The stratification results are therefore limited to the basic code quality and similarity dimensions, which provides only a partial view of detection accuracy impact analysis.

7.4.4 Conclusion Validity

The repair experiments are evaluated on 319 test functions, and per-CWE subsets contain relatively few samples. The observed differences between CWE may be affected by sample variation. In addition, we did not conduct statistical significance testing on the repair metrics. Therefore, the reported differences between Strategy A, B, and C should be interpreted as descriptive observations rather than statistically validated differences.

8

Conclusion and Future Work

8.1 Conclusion

This thesis investigated a two-stage detection and repair pipeline for function-level C/C++ memory-safety vulnerabilities in software relevant to SDVs. The study examined how classification strategy, model selection, and inference-time threshold selection affect detection performance. It also examined whether structured vulnerability information can support LLM-based repair generation under the investigated benchmark conditions.

Regarding RQ1, the clearest result is that classification strategy had the most consistent effect on detection performance. Separating the four target CWE types into specialised binary classifiers provided higher per-type detection performance than the shared multiclass architecture under the evaluated benchmark conditions. Model selection did not produce a stable ranking, since no evaluated pre-trained code model was strongest across all target CWE types.

For threshold selection, the result concerns pipeline behaviour rather than architecture ranking. It does not change the classification-strategy result. Instead, it shows that the decision threshold used by the detector changes how often the ground-truth CWE label reaches the repair prompt. This distinction matters in the pipeline because classifier-level F1 measures the balance between precision and recall for each CWE, while the repair stage depends on whether the correct vulnerability type is included in the detection output.

The RQ1 findings are bounded by the function-level setting, the selected CWE types, and the BigVul and PrimeVul datasets used in the evaluation. Within this scope, the main conclusion is that specialised binary classifiers are the strongest evaluated detection architecture. The model-selection result shows that this pattern is not tied to one pre-trained code model. The threshold-selection result shows that threshold choice should be interpreted at the pipeline level, not only as classifier tuning. It should therefore be read together with classifier-level F1 and downstream repair behaviour.

Regarding RQ2, the experimental results show that detection-augmented prompting improves vulnerability repair quality while preserving basic code quality. The gen-

erated patches remain syntactically valid and maintain high static-analysis quality. For similarity with the ground truth fixes, structural similarity decreases as more vulnerability guidance is added while semantic similarity remains stable, suggesting that detection-augmented prompting leads the LLM to produce patches with more extensive modifications that differ in surface structure from the ground truth fix but preserve the underlying semantic intent.

The most notable improvement is in vulnerability repair quality, which increases consistently with each additional level of vulnerability guidance. Vulnerability type information is the primary driver of this improvement, and additional repair strategy hints provide smaller further benefits. The stratification analysis further shows that the improvement from detection-augmented prompting depends on detection accuracy, while incorrect vulnerability guidance causes limited negative impact on repair performance.

Taken together, the findings suggest that classification strategy, model selection, and inference-time threshold selection all affect function-level memory-safety vulnerability detection under the investigated benchmark conditions. The findings also suggest that detection-augmented prompting improves vulnerability repair quality by injecting vulnerability guidance. The dependence of repair improvement on detection accuracy further highlights that the two stages of the pipeline are closely connected, with more accurate detection enabling higher-quality repair.

8.2 Limitations and Future Work

Several limitations of the current study suggest directions for future work.

First, the experiments were conducted on the BigVul and PrimeVul datasets, which act as open-source proxies for system-level C/C++ software. Although these datasets contain memory-management patterns relevant to SDV development, validating the transferability of the pipeline to proprietary automotive codebases remains an open question. Future collaboration with automotive manufacturers or suppliers could help assess the pipeline under industrial development conditions.

Second, UniXcoder supports input sequences of up to 1024 tokens by design [24], but all experiments in this study used a shared 512-token limit in order to maintain identical input conditions across the evaluated models. Approximately 52% of the CWE-125 test functions exceeded this limit, meaning that vulnerability-relevant context may have fallen outside the visible input window during inference. Re-evaluating UniXcoder with extended input lengths would allow investigation of whether additional context affects detection performance for vulnerability categories associated with longer functions.

Third, the current repair evaluation relies primarily on similarity metrics and heuristic vulnerability-pattern checks. While these metrics provide useful comparative signals, they do not directly evaluate functional correctness or runtime security behaviour. Future work could therefore incorporate compilation-based validation,

targeted unit tests, or execution-based evaluation to provide stronger evidence regarding the behaviour of generated patches.

Fourth, the repair experiments were conducted using a single generative model DeepSeek-V3.2. The results show that detection-augmented prompting is effective under this setup. Future work could extend the evaluation to additional LLMs to assess whether the benefit of detection-augmented prompting generalises across different model families and scales.

Fifth, the current study focuses on four function-level memory-safety vulnerability categories. Extending the pipeline to additional CWE types, as well as to multi-function or cross-file vulnerability patterns requiring interprocedural analysis, represents another direction for future research.

Bibliography

[1] AUTOSAR Consortium. AUTOSAR classic platform – overview. Technical report, Automotive Open System Architecture, 2023. Release R23-11. Available at `https://www.autosar.org/standards/classic-platform`.

[2] R. Baldoni, Emilio Coppa, Daniele Cono D'Elia, C. Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51:1 – 39, 2016.

[3] Paul E. Black. A software assurance reference dataset: Thousands of programs with known bugs. *Journal of Research of the National Institute of Standards and Technology*, 123:1, 2018.

[4] Gerardo Canfora, Andrea Di Sorbo, Sara Forootani, Matias Martinez, and Corrado A. Visaggio. Patchworking: Exploring the code changes induced by vulnerability fixing activities. *Information and Software Technology*, 142:106745, 2022.

[5] Xiansheng Cao, Junfeng Wang, and Peng Wu. Enhancing vulnerability repair through the extraction and matching of repair patterns. *Journal of Systems and Software*, 230:112528, 2025.

[6] Partha Chakraborty, Krishnaveni Arumugam, Mahmoud Alfadel, M. Nagappan, and Shane McIntosh. Revisiting the performance of deep learning-based vulnerability detection on realistic datasets. *IEEE Transactions on Software Engineering*, 50:2163–2177, 2024.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2021.

[8] Marco De Vincenzi, Mert D Pesé, Chiara Bodei, Ilaria Matteucci, Richard R Brooks, Monowar Hasan, Andrea Saracino, Mohammad Hamad, and Sebastian Steinhorst. Contextualizing security and privacy of software-defined vehicles: State of the art and industry perspectives. *arXiv preprint arXiv:2411.10612*, 2024.

[9] DeepSeek-AI, Aixin Liu, Aoxue Mei, Ban Lin, Bing Xue, Bing-Li Wang, Bin Xu, Bochao Wu, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 4171–4186, 2019.

[11] Mahdi Dibaei, Xi Zheng, Kun Jiang, Sasa Maric, Robert Abbas, Shigang Liu, Yuexin Zhang, Yao Deng, Sheng Wen, Jun Zhang, et al. An overview of attacks and defences on intelligent connected vehicles. *arXiv preprint arXiv:1907.07455*, 2019.

[12] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Bhatt, Kai Wang, Yves Gros, Claire Le Goues, Miltiadis Allamanis, and Baishakhi Ray. Vulnerability detection with code language models: How far are we? In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.

[13] G. Dolcetti, V. Arceri, E. Iotti, S. Maffeis, A. Cortesi, and E. Zaffanella. Helping LLMs improve code generation using feedback from testing and static analysis. *Discover Artificial Intelligence*, 6(1):314, 2026.

[14] Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *ACM Trans. Softw. Eng. Methodol.*, 34(3):1–22, 2025.

[15] Zeinab El-Rewini, Karthikeyan Sadatsharan, Dawood Francy Selvaraj, Siby Jose Plathottam, and Prakash Ranganathan. Cybersecurity challenges in vehicular communications. *Vehicular Communications*, 23:100214, 2020.

[16] Jiahao Fan, Yi Li, Xiaoxin Wang, and Zheng Li. C/c++ vulnerability dataset from code changes: The bigvul dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 55–59, 2020.

[17] Sehrish Fatima, Hadi Hemmati, and Lionel C. Briand. Flakyfix: Using large language models for predicting flaky test fix categories and test code repair. *IEEE Transactions on Software Engineering*, 50(12):3146–3171, 2024.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Jianyin Xia, Jian Yin, and Ming Jiang. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[19] M. Ferrag, A. Battah, Norbert Tihanyi, Ridhi Jain, Diana Maimut, Fatima Alwahedi, Thierry Lestable, Narinderjit Singh Thandi, Abdechakour Mechri, M. Debbah, and L. C. Cordeiro. Securefalcon: Are we there yet in automated

software vulnerability detection with llms? *IEEE Transactions on Software Engineering*, 51:1248–1265, 2023.

[20] Lucas Bastos Germano, Ronaldo Ribeiro Goldschmidt, Ricardo Choren Noya, and Julio Cesar Duarte. A systematic review on detection, repair, and explanation of vulnerabilities in source code using large language models. *IEEE Access*, 13:192263–192293, 2025.

[21] GitHub, Inc. GitHub CodeQL. `https://codeql.github.com/`, 2026. Accessed: 2026-04-11.

[22] Neville Grech and Y. Smaragdakis. P/taint: unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages*, 1:1 – 28, 2017.

[23] Anastasiia Grishina, Max Hort, and L. Moonen. The earlybird catches the bug: On exploiting early layers of encoder models for more efficient code classification. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[24] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 7212–7225, 2022.

[25] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations*, 2021.

[26] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei-Jun Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Transactions on Software Engineering*, 49:5154–5188, 2023.

[27] Woorim Han, Yeongjun Kwak, Miseon Yu, Kyeongmin Kim, Younghan Lee, Hyungon Moon, and Yunheung Paek. Rethinking the capability of fine-tuned language models for automated vulnerability repair. *arXiv preprint arXiv:2512.22633*, 2025.

[28] Nafis Tanveer Islam, Mohammad Bahrami Karkevandi, and Peyman Najafirad. Code security vulnerability repair using reinforcement learning with large language models. *arXiv preprint arXiv:2401.07031*, 2024.

[29] Arvinder Kaur and Ruchikaa Nayyar. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029, 2020.

[30] Kaveh Bakhsh Kelarestaghi, Mahsa Foruhandeh, Kevin Heaslip, and Ryan Gerdes. Intelligent transportation system security: Impact-oriented risk assessment of in-vehicle networks. *IEEE Intelligent Transportation Systems Magazine*, 13(2):91–104, 2019.

[31] Arshiya Khan, Guannan Liu, and Xing Gao. Code vulnerability repair with large language model using context-aware prompt tuning. In *2025 IEEE Security and Privacy Workshops (SPW)*, pages 283–287. IEEE, 2025.

[32] Zhihong Liang et al. Vulfix: Software vulnerability repair via data propagation chain representation and local semantic enhancement. In *2025 6th International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, pages 1–6. IEEE, 2025.

[33] Zhenxuan Liao. Long-range context modeling for software vulnerability detection using an xlnet-based approach. *Scientific Reports*, 16(1):5338, 2026.

[34] Chenyu Lin, Yilin Wen, Du Su, Hexiang splinter Tan, Fei Sun, Muhan Chen, Chenfu Bao, and Zhonghou Lyu. Resisting contextual interference in RAG via parametric-knowledge reinforcement. *arXiv preprint arXiv:2506.05154*, 2025.

[35] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '22, pages 544–555. ACM, 2022.

[36] LLVM Project. Clang static analyzer. `https://clang-analyzer.llvm.org/`, 2026. Accessed: 2026-04-11.

[37] Mahdin70. merged_bigvul_primevul. `https://huggingface.co/datasets/mahdin70/merged_bigvul_primevul`, 2023. Accessed: 2026-05-03.

[38] Daniel Marjamäki. Cppcheck: A tool for static C/C++ code analysis. `https://cppcheck.sourceforge.io/`, 2024. Accessed: 2025.

[39] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, R. Watson, and Peter Sewell. Exploring c semantics and pointer provenance. *Proceedings of the ACM on Programming Languages*, 3:1 – 32, 2019.

[40] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. In *Black Hat USA 2015*, 2015.

[41] MITRE Corporation. CWE-787: Out-of-bounds Write. Common Weakness Enumeration, 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/787.html. [Accessed: May 5, 2026].

[42] MITRE Corporation. Common Weakness Enumeration. Common Weakness

Enumeration, 2026. [Online]. Available: https://cwe.mitre.org/. [Accessed: May 23, 2026].

[43] José D'Abruzzo Pereira and Marco Vieira. On the use of open-source c/c++ static analysis tools in large projects. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 97–102. IEEE, 2020.

[44] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. Towards causal deep learning for vulnerability detection. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1888–1898, 2023.

[45] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[46] Amirali Sajadi, Kostadin Damevski, and Preetha Chatterjee. How safe are AI-generated patches? a large-scale study on security risks in LLM and agentic automated program repair on SWE-bench. arXiv preprint arXiv:2507.02976, 2025.

[47] Shaunak Samant. Syntax is not enough: An empirical study of small transformer models for neural code repair. *arXiv preprint arXiv:2512.22216*, 2025.

[48] Madhu Selvaraj and Gias Uddin. A large-scale study of iot security weaknesses and vulnerabilities in the wild. *ACM Transactions on Software Engineering and Methodology*, 34:1 – 40, 2023.

[49] Miaomiao Shao, Yixin Ding, Cuiyun Gao, Junru Wang, and Guoqing Zhu. Fix pattern-aware vulnerability patch generation via in-context learning. *ACM Transactions on Software Engineering and Methodology*, 2026.

[50] Barry Sheehan, Finbarr Murphy, Martin Mullins, and Cian Ryan. Connected and autonomous vehicles: A cyber-risk classification framework. *Transportation research part A: policy and practice*, 124:523–536, 2019.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008, 2017.

[52] Quanjun Wang et al. A survey of LLM-based automated program repair: Taxonomies, design paradigms, and applications. arXiv preprint arXiv:2506.23749, 2025.

[53] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th*

International Conference on Software Engineering, pages 1482–1494, 2023.

[54] Xu Yang, Shaowei Wang, Jiayuan Zhou, and Wenhan Zhu. One-for-all does not work! enhancing vulnerability detection by mixture-of-experts (moe). *Proceedings of the ACM on Software Engineering*, 2:446 – 464, 2025.

[55] Linda Zeng, Rithwik Gupta, Divij Motwani, Yi Zhang, and Diji Yang. Worse than zero-shot? a fact-checking dataset for evaluating the robustness of RAG against misleading retrievals. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2026.

[56] Tieming Zhang et al. A case study of LLM for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the ACM AIware Workshop*, 2024.

[57] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–31, 2024.

[58] Xin Zhou et al. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[59] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[60] Xiaogang Zhu, Sheng Wen, Seyit Ahmet Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys (CSUR)*, 54:1 – 36, 2022.

[61] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μsvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18:2224–2236, 2019.

A

Hyperparameters

This appendix provides the complete hyperparameter configuration used for classifier fine-tuning, as referenced in Section 5.3.

Table A.1: Complete hyperparameter configurations for classifier fine-tuning.

Parameter	**Value**	**Notes**
Pre-trained Models	CodeBERT; GraphCodeBERT; UniXcoder	Pre-trained code models evaluated under the same training configuration.
Max Sequence Length	512 tokens	Applied uniformly across all models; longer inputs are truncated.
Learning Rate	2e-5	Shared across all experiments.
Batch Size	32 (fp16)	Mixed-precision training enabled.
Optimizer	AdamW (wd = 0.01)	Weight decay = 0.01.
Warmup	10% of steps	Linear warmup schedule.
Loss Function	Weighted cross-entropy	Minority-class weighting applied during training.
Max Epochs	10	Early stopping applied based on validation F1-score.
Early Stopping	Patience = 3	Training stopped after three epochs without validation F1 improvement.
Random Seeds	{100, 333, 512, 2025}	Four independent runs per (CWE × model) configuration.
Positive:Negative Ratio	1:2 (all splits)	Negative samples downsampled to twice the positive count.

B

Prompt Templates

This appendix provides the complete prompt templates for all three prompting strategies evaluated in Section 5.4.

Listing B.1: Prompt templates for Strategy A (Unguided), Strategy B (Type-Guided), and Strategy C (Detail-Guided).

```
# -- CWE-specific repair strategy hints for Strategy C ------------------
CWE_REPAIR_HINTS = {
    'CWE-787': (
        'Common fixes include: adding bounds checks before write operations, '
        'validating destination buffer sizes, using safe functions (strncpy,
    snprintf), '
        'checking array indices before write access, adding overflow protection,
    '
        'and ensuring write operations stay within allocated memory boundaries.'
    ),
    'CWE-476': (
        'Common fixes include: adding NULL pointer checks before dereferencing, '
        'validating return values from allocation functions (malloc, calloc), '
        'checking function parameters for NULL, adding defensive NULL checks, '
        'and ensuring pointers are initialized before use.'
    ),
    'CWE-125': (
        'Common fixes include: adding bounds checks before array/pointer access,
    '
        'validating indices against array length, checking pointer validity, '
        'and ensuring loop boundaries do not exceed allocated memory.'
    ),
    'CWE-399': (
        'Common fixes include: ensuring all allocated resources are properly
    released on all paths, '
        'preventing double-free or use-after-free by setting pointers to NULL
    after free, '
        'closing file descriptors, and adding cleanup logic in error handling
    paths.'
    ),
}

def build_prompt_A(code: str, cwe_id: str = None) -> str:
    """
    Strategy A: Unguided Baseline
    """
```

```
    return (
        '### Task:\n'
        'Fix the vulnerability in the code below.\n\n'
        '### Requirements:\n'
        '- Make MINIMAL necessary changes\n'
        '- Preserve the original function signature and logic flow\n'
        '- Do NOT add defensive code, logging, or refactoring unless required for
     the fix\n\n'
        '- Ensure the output starts with "```c" and ends with "```".\n\n'
        '### Output Format:\n'
        'Provide ONLY the **complete and entire** fixed function code in a single
     code block:\n'
        '```c\n'
        '<your fixed code here>\n'
        '```\n\n'
        '### Vulnerable Code:\n'
        f'```c\n{code}\n```\n'
    )


def build_prompt_B(code: str, detected_cwes_list: list) -> str:
    """
    Strategy B: Type-Guided
    """

    cwe_labels = ""
    for c in detected_cwes_list:
        c_id = f"CWE-{c}"
        cwe_labels += f"**{c_id}**: ({CWE_DESCRIPTIONS.get(c_id, f'{c_id}')})"

    return (
        '### Detected Vulnerability:\n'
        f'A security analysis suggests this code may contain the following
    vulnerability: **{cwe_labels}**\n\n'
        '### Task:\n'
        f'1. First, verify if the detected vulnerability type ({cwe_labels}) is
    present in the code\n'
        '2. Fix the identified vulnerability with MINIMAL changes.\n'
        '3. If none of the suggested types strictly match but a vulnerability
    exists, fix the actual issue.\n\n'
        '### Requirements:\n'
        '- Make MINIMAL necessary changes to fix the vulnerability\n'
        '- Preserve the original function signature and logic flow\n'
        '- Do NOT add defensive code, logging, or refactoring unless required for
     the fix\n\n'
        '- Ensure the output starts with "```c" and ends with "```".\n\n'
        '### Output Format:\n'
        'Provide ONLY the **complete and entire** fixed function code in a single
     code block:\n'
        '```c\n'
        '<your fixed code here>\n'
        '```\n\n'
        '### Vulnerable Code:\n'
        f'```c\n{code}\n```\n'
    )
```

```
def build_prompt_C(code: str, detected_cwes_list: list) -> str:
    """
    Strategy C: Type + Strategy-Guided
    """

    cwe_labels = ""
    combined_hints = ""
    for c in detected_cwes_list:
        c_id = f"CWE-{c}"
        cwe_labels += f"**{c_id}**: ({CWE_DESCRIPTIONS.get(c_id, f'{c_id}')})"
        combined_hints += f"- **{c_id}**: {CWE_REPAIR_HINTS.get(c_id, '')}\n"

    return (
        '### Detected Vulnerability:\n'
        f'A security analysis suggests this code may contain the following
    vulnerabilities: **{cwe_labels}**\n\n'
        f'### Recommended Repair Strategies for specific CWE:\n'
        f'{combined_hints}\n\n'
        '### Task:\n'
        f'1. First, verify if the detected vulnerability type ({cwe_labels}) is
    present in the code\n'
        '2. Apply the MOST appropriate strategy and fix the identified
    vulnerability with MINIMAL changes.\n'
        '3. If none of the suggested types strictly match but a vulnerability
    exists, fix the actual issue.\n\n'
        '### Requirements:\n'
        '- Make MINIMAL necessary changes to fix the vulnerability\n'
        '- Preserve the original function signature and logic flow\n'
        '- Do NOT add defensive code, logging, or refactoring unless required for
     the fix\n\n'
        '- Ensure the output starts with "```c" and ends with "```".\n\n'
        '### Output Format:\n'
        'Provide ONLY the **complete and entire** fixed function code in a single
     code block:\n'
        '```c\n'
        '<your fixed code here>\n'
        '```\n\n'
        '### Vulnerable Code:\n'
        f'```c\n{code}\n```\n'
    )
```

C

VPRR Rules

This appendix provides the complete per-CWE regex/keyword rule sets used to compute the VPRR, as referenced in Section 5.4.4.

Listing C.1: VPRR CWE-specific repair pattern rules.

```
def compute_vprr(vuln_code: str, repaired: str, cwe_id: str) -> dict:

    checks = {}
    vuln_lower = vuln_code.lower()
    repair_lower = repaired.lower()

    # CWE-787: Out-of-bounds Write

    if cwe_id == 'CWE-787':
        # 1. Boundary check
        bounds_keywords = [
            'sizeof', 'strlen', 'strnlen', 'size_t',
            '< sizeof', '<= sizeof', '< size', '<= size',
            'buffer_size', 'array_size', 'max_len',
            'if (len', 'if(len', 'if (size', 'if(size'
        ]
        bounds_orig = sum(1 for k in bounds_keywords if k in vuln_lower)
        bounds_repair = sum(1 for k in bounds_keywords if k in repair_lower)
        checks['bounds_check_added'] = bounds_repair > bounds_orig

        # 2. Replacement of unsafe functions
        unsafe_replacements = {
            'strcpy': ['strncpy', 'strlcpy', 'memcpy'],
            'strcat': ['strncat', 'strlcat'],
            'sprintf': ['snprintf'],
            'vsprintf': ['vsnprintf'],
            'gets': ['fgets', 'fgetc'],
            'scanf': ['sscanf'],
        }

        for unsafe, safe_alternatives in unsafe_replacements.items():
            if unsafe in vuln_lower:
                if any(safe in repair_lower for safe in safe_alternatives):
                    checks[f'replaced_{unsafe}'] = True
                elif unsafe not in repair_lower:
                    checks[f'removed_{unsafe}'] = True
```

```
        # 3. Verification before write operation
        write_validation = [
            'if (dst', 'if(dst',  # pointer
            'if (n <', 'if(n<',   # size
            'memset', 'memcpy',   # explicit memory operations
        ]
        checks['write_validation_added'] = any(
            k in repair_lower and k not in vuln_lower for k in write_validation
        )


    # CWE-476: NULL Pointer Dereference

    elif cwe_id == 'CWE-476':
        # 1. NULL check
        null_check_patterns = [
            'if (', 'if(',           # Condition
            '!= null', '== null',     # NULL
            '!ptr', '!p',             # Pointer
            'assert', 'bug_on',       # Assertion
        ]
        null_checks_orig = sum(vuln_lower.count(k) for k in null_check_patterns)
        null_checks_repair = sum(repair_lower.count(k) for k in
    null_check_patterns)
        checks['null_check_increased'] = null_checks_repair > null_checks_orig

        # 2. Memory allocation check
        alloc_functions = ['malloc', 'calloc', 'realloc', 'kmalloc', 'kzalloc']
        for alloc in alloc_functions:
            if alloc in vuln_lower:
                # Check if it checks the return value after allocation
                alloc_pattern = f'{alloc}('
                if alloc_pattern in vuln_code:
                    # null check to handle memory allocation failure
                    alloc_pos = repaired.lower().find(alloc_pattern)
                    if alloc_pos != -1:
                        nearby_code = repaired[alloc_pos:alloc_pos+200].lower()
                        if 'if' in nearby_code and ('null' in nearby_code or '!'
    in nearby_code):
                            checks[f'{alloc}_checked'] = True

        # 3. early return
        early_return = ['return null', 'return -', 'return 0', 'goto']
        early_orig = sum(vuln_lower.count(k) for k in early_return)
        early_repair = sum(repair_lower.count(k) for k in early_return)
        checks['early_return_added'] = early_repair > early_orig


    # CWE-125: Out-of-bounds Read

    elif cwe_id == 'CWE-125':
        # 1. Boundary check
        bounds_keywords = [
            'sizeof', 'strlen', 'strnlen', 'array_size',
            '< size', '<= size', '< len', '<= len',
            '< count', '<= count', '< n', '<= n',
```

```
            'if (i <', 'if(i<', 'if (idx', 'if(idx',
        ]
        bounds_orig = sum(1 for k in bounds_keywords if k in vuln_lower)
        bounds_repair = sum(1 for k in bounds_keywords if k in repair_lower)
        checks['bounds_check_added'] = bounds_repair > bounds_orig

        # 2. Loop boundary check
        loop_keywords = ['for (', 'while (', 'for(', 'while(']
        loop_orig = sum(vuln_lower.count(k) for k in loop_keywords)
        loop_repair = sum(repair_lower.count(k) for k in loop_keywords)

        if loop_orig > 0:
            bound_cond = ['< size', '<= size', '< len', '<= len', '< count']
            has_bound_orig = any(k in vuln_lower for k in bound_cond)
            has_bound_repair = any(k in repair_lower for k in bound_cond)
            checks['loop_bounds_improved'] = has_bound_repair and not
    has_bound_orig

        # 3. pointer validation
        pointer_checks = [
            'if (ptr', 'if(ptr', 'if (!ptr', 'if(!ptr',
            '!= null', '== null', 'assert(ptr', 'assert(p',
        ]
        ptr_orig = sum(1 for k in pointer_checks if k in vuln_lower)
        ptr_repair = sum(1 for k in pointer_checks if k in repair_lower)
        checks['pointer_validation_added'] = ptr_repair > ptr_orig

        # 4. array index verification
        index_checks = ['if (index', 'if(index', 'if (idx', 'if(idx']
        checks['index_validation_added'] = any(
            k in repair_lower and k not in vuln_lower for k in index_checks
        )


    # CWE-399: Resource Management Errors

    elif cwe_id == 'CWE-399':
        # 1. memory cleanup
        res_kw = ['free(', 'kfree(', 'realloc(', 'delete ', 'destroy(', 'cleanup'
    ]
        r_orig  = sum(vuln_lower.count(k) for k in res_kw)
        r_patch = sum(repair_lower.count(k) for k in res_kw)
        checks['memory_cleanup_added'] = r_patch > r_orig

        # 2. added handle close
        fd_kw = ['close(', 'fclose(', 'closedir(', 'CloseHandle(', 'Close(']
        fd_orig = sum(vuln_lower.count(k) for k in fd_kw)
        fd_patch = sum(repair_lower.count(k) for k in fd_kw)
        checks['fd_cleanup_added'] = fd_patch > fd_orig

        # 3. safe free pattern
        import re
        null_pattern = r'free\s*\(\s*([a-zA-Z0-9_.\->]+)\s*\)\s*;\s*\1\s*=\s*(
    NULL|0|nullptr)'
        checks['safe_free_pattern'] = bool(re.search(null_pattern, repair_lower,
    re.IGNORECASE))
```

```
    # 4. unified error handling, resource cleanup
    cleanup_labels = ['goto err', 'goto cleanup', 'goto out', 'goto fail']
    checks['error_path_cleanup_added'] = any(k in repair_lower and k not in
vuln_lower for k in cleanup_labels)

return {
    'pass_rule': positive_checks > 0,
    'details': checks,
}
```