Comprehensive Journal of Science
Volume (11), Issue (42), (Sapt, 2026)
ISSN: 3014-6266

مجلة العلوم الشاملة
المجلد (11) العدد 42 سبتمبر 2026
ردمد: 3014-6266

Memory Safety in Kernel Development: An AI-Assisted Comparative Study of Rust- and C-Based Approaches to Preventing Memory Corruption

Asma Mustafa Alhadi[1], Nuha Omran Abokhdair*[2]

1 Department of Computer Engineering and Sciences, Faculty of Science, University of Zawia, Libya

2 Department of Computer Sciences, Faculty of Science, University of Zawia, Libya

*Corresponding author email: abo.khdeir@zu.edu.ly

Received: 2/8/2026 Revised: 10/8/2026 Accepted: 30/8/2026 Publication: 21/9/2026

ABSTRACT

Memory corruption remains a major source of security vulnerabilities in low-level and kernel-oriented software, particularly in systems implemented in C, where manual memory management exposes developers to errors such as buffer overflows, use-after-free, double-free, null-pointer dereferences, and memory leaks. This paper presents KernelMemSafe-AI, an AI-assisted comparative framework for evaluating memory-safety behavior in C- and Rust-based approaches within a kernel-development context. The proposed framework integrates two complementary components: an AI-assisted vulnerability-pattern classifier and a practical experimental comparison of C and Rust implementations.

A controlled and balanced dataset of 520 C/C++ and kernel-style code snippets was constructed and labeled as vulnerable or safe across multiple memory-corruption categories. The final hybrid classifier combines token and bigram analysis, Naive Bayes learning, rule-based memory-safety indicators, confidence scoring, and five-fold cross-validation. The classifier achieved 95.58% accuracy, 96.11% precision, 95.00% recall, and a 95.55% F1-score. In addition, five representative memory-safety experiments were implemented to compare language behavior in buffer-overflow, use-after-free, double-free, null-pointer-dereference, and memory-leak scenarios.

The experimental results show that C code often compiles successfully yet exhibits runtime failures, undefined behavior, allocator-level aborts, garbage values, or unreleased memory. In contrast, Rust prevents or mitigates several unsafe memory operations through ownership and move semantics, bounds checking, Option<T>, and automatic cleanup using Drop. The findings indicate that Rust provides stronger language-level protection against common memory-corruption patterns, while AI-assisted analysis can support the early identification of unsafe C code patterns. Overall, the study shows that combining memory-safe programming mechanisms with AI-assisted vulnerability analysis can improve the reliability and security of kernel-oriented software.

Keywords: Memory Safety, Rust, C Programming, Kernel Development, Memory Corruption, Vulnerability Detection, Artificial Intelligence, Ownership, Buffer Overflow, Use-After-Free.

سلامة الذاكرة في تطوير النواة: دراسة مقارنة بمساعدة الذكاء الاصطناعي بين نهجي Rust و C في منع تلف الذاكرة

أسماء مصطفى الهادي[1]، نهى عمران أبو خذير[2]

هندسة وعلوم الحاسب، كلية العلوم، جامعة الزاوية، الزاوية، ليبيا 1

قسم علوم الحاسب، كلية العلوم، جامعة الزاوية، الزاوية، ليبيا 2

ملخـــــــــص البحـــــــــث

تُعدّ مشكلات تلف الذاكرة من أهم مصادر الثغرات الأمنية في البرمجيات منخفضة المستوى والبرمجيات المرتبطة بتطوير نواة نظام التشغيل، وخصوصًا في الأنظمة المطوّرة بلغةC ، حيث يؤدي الاعتماد على إدارة الذاكرة يدويًا إلى ظهور أخطاء مثل تجاوز

حدود الذاكرة، واستخدام الذاكرة بعد تحريرها، والتحرير المزدوج للذاكرة، والوصول إلى مؤشر فارغ، وتسرب الذاكرة. تقدم هذه الورقة إطارًا مقارنًا بمساعدة الذكاء الاصطناعي يسمى KernelMemSafe-AIلتقييم سلوك سلامة الذاكرة في النهجين المعتمدين على لغتي C و Rustضمن سياق تطوير النواة. يجمع الإطار المقترح بين مكوّنين متكاملين: مصنّف ذكي لاكتشاف أنماط الثغرات، وتجارب عملية تقارن بين تطبيقات C و .Rustتم إنشاء مجموعة بيانات متوازنة ومضبوطة تتكون من 520 مقطعًا برمجيًا بلغة C/C++وبأسلوب قريب من برمجة النواة، وتم تصنيفها إلى مقاطع آمنة ومقاطع معرضة للثغرات عبر عدة فئات من أخطاء تلف الذاكرة. اعتمد المصنّف الهجين على تحليل الرموز والثنائيات النصية، وخوارزميةNaive Bayes ، ومؤشرات قائمة على قواعد سلامة الذاكرة، ودرجات الثقة، والتحقق المتقاطع بخمسة أجزاء. حقق المصنّف دقة بلغت 95.58%، ودقة إيجابية 96.11%، واسترجاعًا 95.00%، وقيمة F1 بلغت 95.55%. كما تم تنفيذ خمس تجارب عملية لقياس سلوك اللغتين في حالات تجاوز حدود الذاكرة، واستخدام الذاكرة بعد تحريرها، والتحرير المزدوج، والوصول إلى مؤشر فارغ، وتسرب الذاكرة. أظهرت النتائج أن كود Cغالبًا ما يُترجم بنجاح رغم أنه قد يؤدي أثناء التشغيل إلى فشل وقت التنفيذ أو سلوك غير معرّف أو انهيار في مخصّص الذاكرة أو قيم غير صحيحة أو ذاكرة غير محررة. في المقابل، تمنع Rust أو تقلل العديد من هذه الأخطاء من خلال الملكية، ونقل الملكية، وفحص حدود المصفوفات، واستخدام<Option<T> ، والتنظيف التلقائي عبر .Drop وتبيّن النتائج أن Rust توفر حماية أقوى على مستوى اللغة ضد أنماط تلف الذاكرة الشائعة، بينما يساعد التحليل المدعوم بالذكاء الاصطناعي في الاكتشاف المبكر لأنماط كود Cغير الآمنة. وبشكل عام، توضح الدراسة أن الجمع بين آليات البرمجة الآمنة للذاكرة والتحليل الذكي للثغرات يمكن أن يعزز موثوقية وأمن البرمجيات الموجهة لتطوير النواة.

الكلمات الدالة: سلامة الذاكرة، لغةRust ، لغةC ، تطوير النواة، تلف الذاكرة، اكتشاف الثغرات، الذكاء الاصطناعي.

1. INTRODUCTION

Memory safety remains a major challenge in low-level and kernel-oriented software development. Kernel-level code executes with high privileges and interacts directly with memory, device drivers, hardware resources, and core system services.

Therefore, even a small memory-management error may lead to system crashes, privilege escalation, information leakage, denial of service, or exploitable vulnerabilities. For decades, C has been the dominant language for operating-system and kernel development because of its efficiency, deterministic behavior, and direct hardware control. However, C relies on manual memory management, leaving developers responsible for pointer handling, bounds checking, allocation, and deallocation. This manual model makes C-based systems vulnerable to memory-corruption errors such as buffer overflows, use-after-free, double-free, null-pointer dereferences, and memory leaks. These risks have been repeatedly emphasized by security organizations and industry reports. The National Security Agency recommended adopting memory-safe languages to reduce exploitable memory-related vulnerabilities [1], while Microsoft reported that memory-safety issues have historically represented a major proportion of security vulnerabilities in its software ecosystem [2]. These observations confirm that memory safety is not only a programming concern but also a critical cybersecurity and reliability issue.

Rust has emerged as a promising alternative for safer systems programming. It provides low-level control while enforcing stronger safety rules through ownership, borrowing, move semantics, lifetime checking, and automatic cleanup. Unlike C, Rust detects many unsafe memory operations before execution because the compiler rejects invalid ownership or borrowing patterns. Recent Rust-for-Linux studies have examined Rust adoption, developer experience, security impact, compatibility limitations, and the gradual coexistence of Rust with legacy C kernel infrastructure [3]–[7]. These studies show that Rust is increasingly considered a practical approach to improving kernel safety, although full integration remains technically complex. Comparative research has also examined spatial and temporal memory safety in C and Rust [8].

In parallel, artificial intelligence has become an important direction in software vulnerability detection. Learning-based models can identify unsafe source-code patterns from labeled examples, and recent studies have proposed C/C++ vulnerability datasets, transformer-based models, graph-based techniques, and interpretable vulnerability-detection methods [9]–[18]. Recent research has also investigated improved vulnerability-detection models and the role of large language models in software security [19], [20]. However, AI-based detection alone does not eliminate unsafe behavior from the language itself. Instead, it can serve as an analytical layer for identifying risky C patterns, while Rust provides language-level mechanisms that prevent or mitigate such errors. Motivated by these directions, this paper proposes KernelMemSafe-AI, an AI-assisted comparative framework for evaluating memory-safety behavior in C- and Rust-based approaches within a kernel-development context. As illustrated in Figure 1, the framework combines a controlled vulnerability-pattern detection layer with practical C-versus-Rust experiments.

A balanced dataset of 520 C/C++ and kernel-style snippets was constructed, including 260 vulnerable and 260 safe samples. The dataset covers representative memory-corruption categories such as buffer overflow, use-after-free, double-free, null-pointer dereference, memory leak, and dangerous function usage. A hybrid classifier based on token and bigram analysis, Naive Bayes learning, and rule-based memory-safety indicators is used to detect unsafe memory-handling patterns.

The classifier also incorporates confidence scoring and five-fold cross-validation to improve the reliability of the evaluation.

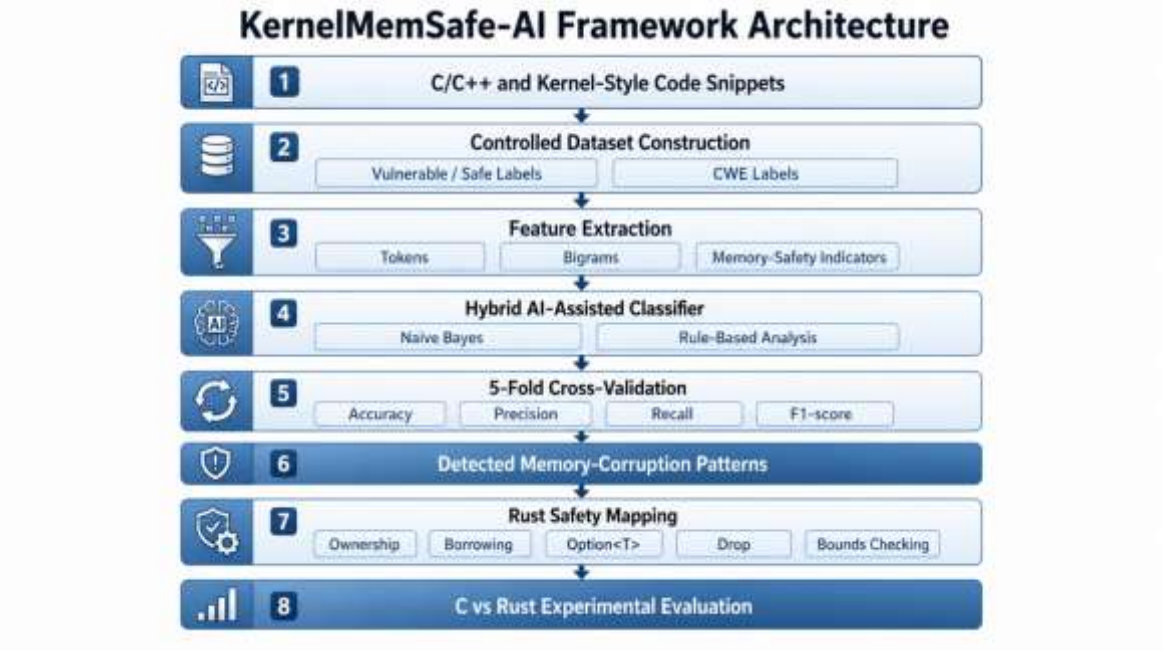

Figure 1. Overall architecture of the proposed KernelMemSafe-AI framework

The experimental component evaluates five representative memory-safety scenarios: buffer overflow, use-after-free, double-free, null-pointer dereference, and memory leak. In the C implementations, unsafe behavior often compiles successfully and appears at runtime as stack-smashing failures, segmentation faults, allocator aborts, garbage values, or unreleased memory. In contrast, Rust either prevents unsafe operations at compile time through ownership and move semantics or enforces safer handling through bounds checking, Option<T>, and automatic cleanup using Drop.

The main contributions of this paper are as follows:

- It proposes KernelMemSafe-AI, a framework that integrates AI-assisted vulnerability-pattern detection with C-versus-Rust memory-safety evaluation.
- It constructs a balanced dataset of 520 C/C++ and kernel-style snippets labeled as vulnerable or safe across multiple memory-corruption categories.
- It evaluates a hybrid AI classifier using five-fold cross-validation and reports standard classification metrics.
- It implements five practical experiments comparing how C and Rust handle representative memory-safety failures.
- It maps AI-detected unsafe C patterns to Rust safety mechanisms, including ownership, borrowing, bounds checking, Option<T>, and Drop.

The rest of this paper is organized as follows. Section 2 reviews related work on memory safety, Rust for Linux, and AI-based vulnerability detection. Section 3 presents the proposed methodology. Section 4 describes the implementation and experimental setup. Section 5 presents the evaluation and results. Section 6 discusses the findings. Section 7 presents the threats to validity. Finally, Section 8 concludes the paper and outlines future research directions.

2. LITERATURE REVIEW

The transition toward memory-safe systems programming has become an important research direction in modern kernel and low-level software development. This section reviews the main studies related to memory safety in C-based systems, Rust adoption in kernel development, AI-assisted vulnerability detection, and the research gap addressed by the proposed KernelMemSafe-AI framework.

2.1 Memory Safety in Low-Level Systems

Memory safety remains a major concern in systems programming because low-level software often requires direct access to memory, hardware resources, and privileged execution environments. C has historically dominated kernel and operating-system development because of its efficiency and low-level control. However, its reliance on manual memory management exposes developers to serious errors such as buffer overflow, use-after-free, double-free, null-pointer dereference, and memory leaks. These vulnerabilities are especially dangerous in kernel-level software because they may lead to system crashes, privilege escalation, or exploitable memory corruption.

The importance of this issue has been emphasized by major security organizations. The National Security Agency recommended adopting memory-safe languages to reduce exploitable memory-related vulnerabilities [1]. Similarly, Microsoft reported that memory-safety issues have historically represented a large portion of software security vulnerabilities [2]. These reports establish memory safety as a practical cybersecurity issue rather than only a programming-language concern.

2.2 Rust for Kernel Development

Rust has gained increasing attention as a safer alternative for systems programming because it combines low-level control with compile-time safety enforcement. Its ownership model, borrowing rules, lifetime checking, move semantics, and automatic resource cleanup can prevent many memory errors before program execution. This makes Rust particularly relevant to kernel development, where runtime memory failures may have severe consequences.

Recent Rust-for-Linux studies have examined the opportunities and challenges of introducing Rust into the Linux kernel ecosystem. Li et al. analyzed the success, dissatisfaction, and compromises involved in Rust-for-Linux adoption [3]. Other studies investigated the security impact of Rust in the Linux kernel and its role in reducing memory-related defects [4]. Panter and Eisty discussed recent advances in Rust for Linux kernel development [5], while additional works examined the evolving integration of Rust with existing C-based kernel infrastructure and Rust-based driver development [6], [7]. These studies show that Rust offers strong safety advantages, but its adoption remains gradual due to compatibility, ecosystem, and developer-experience challenges.

2.3 Spatial and Temporal Memory Safety

A key distinction in memory-safety research is the difference between spatial and temporal memory safety. Spatial memory safety concerns access within valid memory boundaries, such

as preventing buffer overflows and out-of-bounds access. Temporal memory safety concerns access only during an object's valid lifetime, such as preventing use-after-free and double-free errors. Syalim and Sheradhien compared manual and automatic approaches to spatial and temporal memory safety in C and Rust [8]. Their study supports the view that Rust reduces reliance on manual developer checks by enforcing safety rules through the compiler.

This distinction is important for the present study because KernelMemSafe-AI evaluates multiple types of memory violation. Specifically, it considers buffer overflow, use-after-free, double-free, null-pointer dereference, and memory leak. This broader evaluation provides a clearer comparison between C's manual memory model and Rust's language-enforced safety mechanisms.

2.4 AI-Based Vulnerability Detection

Artificial intelligence has become an important direction in software vulnerability detection. AI-based models can learn vulnerable patterns from labeled code examples and assist in identifying unsafe source-code structures. Fan et al. introduced a C/C++ vulnerability dataset with code changes and CVE summaries, supporting data-driven vulnerability analysis [9]. Chakraborty et al. evaluated deep-learning-based vulnerability detection and emphasized the need for careful validation rather than relying solely on high accuracy values [10].

More recent studies have proposed advanced approaches to vulnerability detection, including transformer-based prediction models such as LineVul [11], large vulnerable source-code datasets such as DiverseVul [12], source-code pretraining models such as VulBERTa [13], fine-grained interpretation methods [14], graph neural-network models [15], and multi-domain knowledge-based vulnerability discovery [16]. Survey studies further highlight the progress and limitations of deep-learning-based vulnerability detection, particularly regarding dataset quality, model generalization, and practical deployment [17], [18]. Recent works also examine improved vulnerability-detection models and the role of large language models in software security [19], [20].

2.5 Research Gap

Although previous studies provide important foundations, most Rust-for-Linux research focuses on adoption, kernel integration, developer experience, and general security implications. In contrast, AI-based vulnerability detection studies mainly focus on classification models, datasets, and prediction accuracy. Few studies combine both directions by using AI to identify unsafe C memory-handling patterns and then mapping those patterns to Rust safety mechanisms.

This gap motivates the proposed KernelMemSafe-AI framework. Unlike studies that discuss Rust memory safety only conceptually, this work includes practical C-versus-Rust experiments across five representative memory-corruption scenarios. Unlike AI-based vulnerability-detection studies that focus only on model performance, this work connects AI-detected unsafe patterns to concrete Rust mechanisms such as ownership, borrowing, bounds checking, Option<T>, and Drop. Therefore, KernelMemSafe-AI contributes an integrated approach that combines AI-assisted vulnerability-pattern detection with practical memory-safety evaluation in a kernel-development context.

2.6 Summary Comparison with Existing Studies

To further clarify the research gap, this subsection summarizes how the proposed KernelMemSafe-AI framework differs from existing studies. Previous Rust-for-Linux research mainly focuses on adoption, integration challenges, developer experience, and the security impact of introducing Rust into the Linux kernel. These studies are important, but they generally do not include an AI-based vulnerability-detection component or a structured mapping between unsafe C patterns and Rust safety mechanisms.

In contrast, AI-based vulnerability-detection studies focus mainly on classification accuracy, datasets, model design, and vulnerability prediction. Although these works are useful for detecting unsafe code patterns, they generally do not compare how different programming languages prevent or mitigate the same memory-safety failures. Therefore, they provide detection capability but not a direct language-level safety comparison.

Some comparative studies of C and Rust discuss spatial and temporal memory safety, showing that Rust can reduce reliance on manual memory management. However, these studies usually focus on language-level comparison and do not integrate AI-assisted vulnerability-pattern detection into the evaluation.

In contrast, the proposed KernelMemSafe-AI framework combines these directions in a single integrated study. It uses AI to identify unsafe memory-handling patterns in C/C++ and kernel-style snippets, evaluates classifier performance using a balanced dataset and five-fold cross-validation, and compares C and Rust through practical memory-safety experiments. The proposed framework therefore differs from previous work by connecting AI-based detection with Rust's language-level prevention mechanisms.

Table 1. Summary Comparison with Existing Studies

Study Category	Main Focus	AI	C/Rust	Tests
Rust-for-Linux studies	Rust adoption and kernel integration	No	Partial	Limited
AI vulnerability detection studies	Vulnerability classification and prediction	Yes	No	No
C-versus-Rust memory-safety studies	Spatial and temporal memory safety	No	Yes	Yes
Proposed KernelMemSafe-AI	AI-assisted C-versus-Rust memory-safety evaluation	Yes	Yes	Yes

3. METHODOLOGY

This section describes the proposed KernelMemSafe-AI methodology for evaluating memory-safety behavior in C- and Rust-based approaches within a kernel-development context. The methodology combines two complementary layers: an AI-assisted vulnerability-pattern detection layer and a practical C-versus-Rust experimental evaluation layer. This design allows the study to analyze unsafe memory-handling patterns from two perspectives: first, by detecting

vulnerable C/C++ and kernel-style code snippets using a controlled classifier; and second, by comparing how C and Rust behave under representative memory-corruption scenarios.

The methodology is organized into five main phases: dataset construction, feature extraction, AI-assisted classification, cross-validation-based evaluation, and practical C-versus-Rust testing. This structure connects AI-based vulnerability detection with language-level memory-safety mechanisms.

3.1 Dataset Construction

A controlled and balanced dataset was constructed to support the AI-assisted vulnerability-pattern classifier. The dataset contains 520 C/C++ and kernel-style code snippets, divided equally into 260 vulnerable samples and 260 safe samples. The vulnerable samples represent common memory-corruption patterns frequently found in low-level and systems programming, while the safe samples represent corrected or safer alternatives.

The dataset covers multiple memory-safety categories, including buffer overflow, use-after-free, double-free, null-pointer dereference, memory leak, and dangerous function usage. These categories were selected because they represent common memory-related weaknesses in C-based systems and are highly relevant to kernel-oriented development. Table 2 summarizes the vulnerability categories used in the dataset.

Table 2. Dataset Vulnerability Categories

Category	**Purpose and Risk**
Buffer Overflow	Represents out-of-bounds memory access that may cause stack corruption or arbitrary code execution
Use-After-Free	Represents access to released memory, which may lead to dangling pointers and undefined behavior
Double Free	Represents releasing the same memory region more than once, which may cause allocator corruption or crashes
Null Pointer Dereference	Represents unsafe access through a null pointer, which may cause segmentation faults or denial of service
Memory Leak	Represents allocated memory that is not released, causing resource exhaustion over time
Dangerous Function Usage	Represents unsafe C functions or unchecked memory operations that may cause buffer overwrites

3.2 Feature Extraction

After dataset construction, each code snippet was processed to extract features relevant to memory-safety analysis. The feature-extraction process focused on both statistical and rule-based indicators. Token-level features were extracted from source-code elements such as function names, pointer usage, array indexing, allocation functions, deallocation functions, and unsafe memory operations. Bigram features were also used to capture short code-pattern relationships, such as allocation followed by unsafe access or deallocation followed by pointer reuse.

In addition to token and bigram features, rule-based memory-safety indicators were included to strengthen classification. These indicators were designed to detect patterns commonly associated with unsafe memory behavior, such as the use of malloc, free, unchecked array indexing, repeated deallocation, null-pointer access, and unsafe string or memory-copy

functions. Combining statistical-learning features with explicit memory-safety indicators improves the classifier's ability to identify vulnerability patterns in controlled code snippets.

3.3 AI-Assisted Vulnerability Classification

The AI component of KernelMemSafe-AI uses a hybrid classification approach. The classifier combines token and bigram analysis, Naive Bayes learning, rule-based memory-safety indicators, and confidence scoring. Naive Bayes was selected because it is lightweight, interpretable, and suitable for proof-of-concept classification using structured textual features. The rule-based layer complements the learning model by explicitly capturing known memory-safety risk indicators.

The classifier predicts whether a given code snippet is vulnerable or safe. It is important to note that the AI component is designed as a controlled proof-of-concept vulnerability-pattern classifier rather than a production-grade vulnerability scanner. Its purpose is to support experimental analysis by identifying unsafe memory-handling patterns and relating them to known memory-corruption categories.

3.4 Cross-Validation and Evaluation Metrics

To evaluate the classifier, a five-fold cross-validation strategy was applied. The dataset was divided into five folds, where four folds were used for training and one fold was used for testing in each iteration. This process was repeated five times so that each fold was used once as the testing subset. The final results were calculated by aggregating the performance across all folds.

The classifier was evaluated using standard classification metrics: accuracy, precision, recall, and F1-score. Accuracy measures the overall correctness of the classifier. Precision measures how many snippets predicted as vulnerable were actually vulnerable. Recall measures how many vulnerable snippets were correctly detected. The F1-score provides a balanced measure of precision and recall. Figure 2 summarizes the expected behavior of C and Rust across the five representative memory-safety scenarios.

"C vs Rust Experimental Results"

Memory-Safety Test	C Behavior	Rust Behavior
Buffer Overflow	⚠ Stack smashing / runtime abort	🛡 Bounds-checking panic
Use-After-Free	⚠ Garbage value after free	🛡 Compile-time error E0382
Double Free	⚠ Double free detected / abort	🛡 Compile-time error E0382
Null Pointer Dereference	⚠ Segmentation fault	🛡 Safe Option<T> handling
Memory Leak	⚠ Missing free	🛡 Automatic cleanup using Drop

Figure 2. Summary of C-versus-Rust behavior across five memory-safety scenarios

3.5 Practical C-versus-Rust Experimental Design

The second layer of the methodology consists of practical experiments comparing C and Rust under representative memory-safety scenarios. Five test cases were implemented: buffer overflow, use-after-free, double-free, null-pointer dereference, and memory leak. These scenarios were selected because they represent common spatial and temporal memory-safety violations in low-level software. For each test case, C and Rust implementations were developed with equivalent logical intent. The C version was used to observe how manual memory management behaves when unsafe operations are introduced. The Rust version was used to evaluate whether the language prevents the same unsafe behavior at compile time or handles it safely at runtime. Table 3 summarizes the experimental test cases.

3.6 Mapping Vulnerability Patterns to Rust Safety Mechanisms

After AI-assisted classification and practical testing, unsafe C patterns were mapped to Rust safety mechanisms. This mapping connects the AI detection layer with the language-level prevention layer. For example, buffer-overflow patterns are related to Rust's bounds checking; use-after-free and double-free patterns are related to ownership and move semantics; null-pointer dereference is related to Option<T>; and memory-leak reduction is related to automatic cleanup through Drop.

This mapping allows the proposed framework to go beyond simple classification. Instead of only predicting whether code is vulnerable, KernelMemSafe-AI analyzes how the same class of vulnerability can be prevented or reduced through Rust's safety model.

Table 3. Experimental Memory-Safety Test Cases

Test Case	C Outcome	Rust Safety Mechanism
Buffer Overflow	Out-of-bounds access may compile and cause stack smashing	Bounds checking prevents invalid index access
Use-After-Free	Released memory may be accessed and produce garbage values	Ownership and move semantics prevent access after a value
Double Free	Repeated release may trigger allocator abort or runtime failure	Single ownership prevents repeated release
NullPointer Dereference	Access through a null pointer may cause a segmentation fault	Option<T> enforces safe optional handling
Memory Leak	Allocated memory may remain unreleased	Drop/RAII releases owned resources automatically

3.7 Methodological Scope

The methodology is designed as a controlled comparative study. The dataset and experiments are intended to demonstrate memory-safety behavior in representative C/C++ and kernel-style scenarios. Therefore, the AI classifier is not claimed to replace real-world static-analysis tools or production security scanners. Similarly, the Rust experiments evaluate

language-level safety behavior in controlled scenarios rather than claiming that Rust can completely replace C in existing kernel ecosystems. This scope ensures that the results remain scientifically accurate and directly connected to the objectives of the study.

4. IMPLEMENTATION AND EXPERIMENTAL SETUP

This section describes the implementation environment and experimental setup used to validate the proposed KernelMemSafe-AI framework. The implementation was designed to support both components of the study: the AI-assisted vulnerability-pattern classifier and the practical C-versus-Rust memory-safety experiments.

4.1 Development Environment

The experiments were conducted in a Linux-based environment using Ubuntu on Windows Subsystem for Linux (WSL). This environment was selected because it provides a practical Linux command-line interface while supporting the development, testing, and compilation of both C and Rust programs. The main tools used in the implementation included Python for AI model development, GCC for compiling C test programs, and rustc for compiling Rust test programs. The AI component was implemented using a Python-based machine-learning pipeline. This setup enabled a consistent comparison between manually managed memory behavior in C and compiler-enforced safety behavior in Rust. Table 4 summarizes the main implementation tools and components.

Table 4. Implementation Environment and Tools

Component	Tool / Description
Operating Environment	Ubuntu on Windows Subsystem for Linux
AI Development	Python-based machine-learning pipeline
C Compilation	GCC compiler
Rust Compilation	rustc compiler
Dataset	520 balanced C/C++ and kernel-style snippets
Classifier	Hybrid Naive Bayes with rule-based indicators
Evaluation Method	Five-fold cross-validation
C/Rust Experiments	Five behavioral memory-safety test cases

4.2 AI Classifier Implementation

The AI-assisted classifier was implemented as a hybrid vulnerability-pattern detection model. The classifier combines token- and bigram-based source-code analysis with rule-based memory-safety indicators. Naive Bayes learning was used as the statistical classification component because it is lightweight, interpretable, and suitable for structured textual features extracted from source code.

The classifier was trained and evaluated using a controlled dataset of 520 C/C++ and kernel-style code snippets. The dataset was equally divided into 260 vulnerable samples and

260 safe samples. Each snippet was represented using extracted textual and memory-safety features, including pointer usage, allocation and deallocation functions, unsafe array access, null pointer patterns, repeated deallocation, and dangerous function usage.

To improve reliability, the final classifier used confidence scoring and five-fold cross-validation. This allowed the model to be evaluated across multiple training and testing splits rather than relying on a single train-test division. The implementation output included the main evaluation metrics: accuracy, precision, recall, F1-score, and confusion matrix values.

4.3 C and Rust Experimental Setup

The practical experiments were designed to compare how C and Rust behave under representative memory-safety failures. Five scenarios were implemented: buffer overflow, use-after-free, double-free, null-pointer dereference, and memory leak. These scenarios were selected because they represent common spatial and temporal memory-safety problems in low-level software. For each scenario, C and Rust programs were created with equivalent logical intent. The C versions were used to observe how unsafe memory operations behave when manual memory management is involved. The Rust versions were used to evaluate whether similar unsafe behavior is prevented during compilation or handled safely at runtime. The observed behavior of each experiment was recorded and classified according to the relevant failure or prevention mechanism. In C, the observed outcomes included runtime failures, undefined behavior, garbage values, allocator aborts, segmentation faults, and unreleased memory. In Rust, the observed outcomes included compile-time rejection, bounds checking, safe optional handling using Option<T>, and automatic cleanup using Drop.

4.4 Experimental Output Collection

The experimental results were collected from two sources. The first source was the AI classifier output, which produced numerical evaluation results based on the controlled dataset and cross-validation process. The second source was the observed behavior of the C and Rust test programs during compilation and execution.

The AI results were reported using standard classification metrics, while the C-versus-Rust experiments were reported as behavioral outcomes rather than performance benchmarks. This distinction is important because the purpose of the C/Rust experiments was to evaluate memory-safety behavior, not execution speed. Therefore, the practical results focus on whether unsafe behavior was allowed, detected at runtime, prevented at compile time, or handled safely.

4.5 Reproducibility Considerations

To support reproducibility, the implementation followed a controlled experimental workflow. The dataset was balanced, the same vulnerability categories were used throughout the evaluation, and the classifier was tested using five-fold cross-validation. The C and Rust experiments were designed with equivalent logic for each scenario to ensure a fair behavioral comparison.

However, the implementation remains a controlled proof-of-concept study. The AI classifier is not intended to replace production-grade vulnerability scanners, and the C/Rust experiments are not intended to represent all possible kernel-level memory-safety cases. Instead, the setup provides a focused and reproducible foundation for evaluating how AI-assisted detection and Rust safety mechanisms can contribute to reducing memory-corruption risks.

5. EVALUATION AND RESULTS

This section presents the evaluation results of the proposed KernelMemSafe-AI framework. The evaluation is divided into two main parts. The first part evaluates the AI-assisted vulnerability-pattern classifier using a controlled and balanced dataset with five-fold cross-validation. The second part analyzes the practical behavior of C and Rust implementations under representative memory-safety scenarios. Together, these results demonstrate both the analytical capability of the AI component and the preventive effect of Rust's language-level safety mechanisms.

5.1 Dataset and Evaluation Setup

The AI-assisted classifier was evaluated using a controlled and balanced dataset consisting of 520 C/C++ and kernel-style code snippets. The dataset includes 260 vulnerable samples and 260 safe samples. This balanced structure was used to reduce class bias and ensure fair evaluation across vulnerable and safe code patterns. The vulnerable samples cover representative memory-corruption categories, including buffer overflow, use-after-free, double-free, null-pointer dereference, memory leak, and dangerous function usage. The safe samples represent corrected or safer versions of similar logic. Figure 3 shows the overall dataset composition.

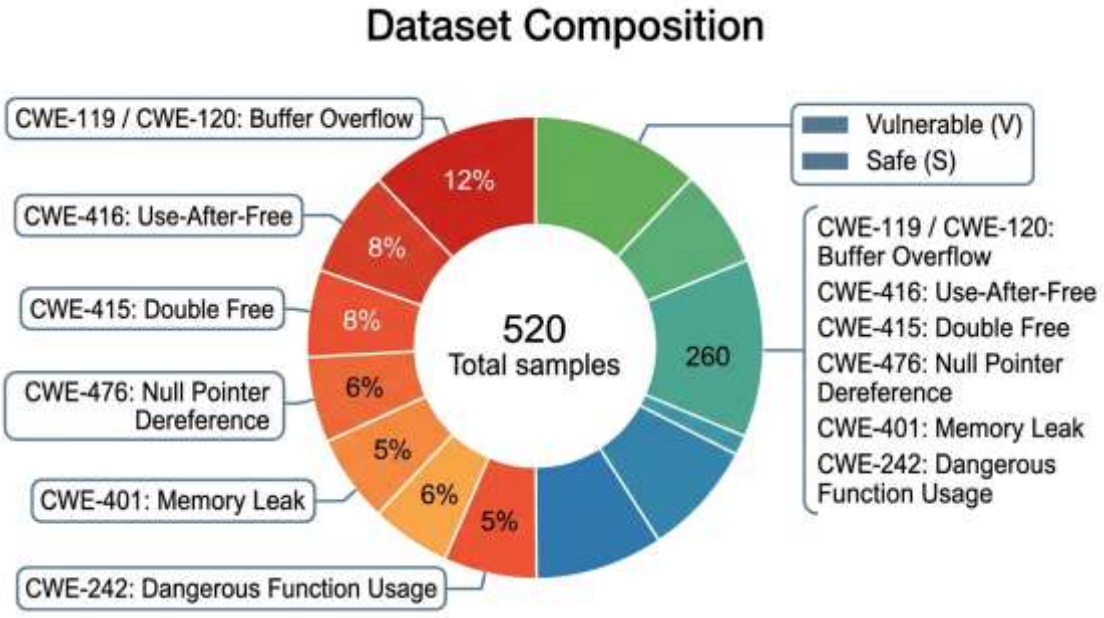

Figure 3. Composition of the controlled memory-safety dataset

5.2 AI Classifier Performance

Using five-fold cross-validation, the final hybrid classifier achieved 95.58% accuracy, 96.11% precision, 95.00% recall, and a 95.55% F1-score. These results indicate that the classifier was able to distinguish vulnerable snippets from safe snippets with high consistency. Accuracy reflects the overall correctness of the model. Precision shows that most snippets predicted as vulnerable were actually vulnerable, reducing false alarms. Recall shows that the

classifier successfully detected most vulnerable snippets. The F1-score confirms a strong balance between precision and recall. Table 5 summarizes the final classifier performance.

Table 5. AI Classifier Performance Metrics

Metric	Result
Accuracy	95.58%
Precision	96.11%
Recall	95.00%
F1-score	95.55%

5.3 Model Evolution

To show that the final performance was achieved through systematic improvement, the classifier was evaluated across three development stages. The initial version, V1, used a text-based Naive Bayes model and achieved 54.55% accuracy with a 44.44% F1-score. The V3 model introduced feature-based Bernoulli Naive Bayes and improved to 82.69% accuracy and an 82.58% F1-score. The final V4 model combined token analysis, bigram features, rule-based memory-safety indicators, confidence scoring, and five-fold cross-validation, reaching 95.58% accuracy and a 95.55% F1-score. This progression shows that the final hybrid model did not achieve strong performance immediately. Instead, the improvement resulted from the gradual refinement of feature representation and the addition of memory-safety-aware indicators. Figure 4 illustrates this improvement across classifier versions.

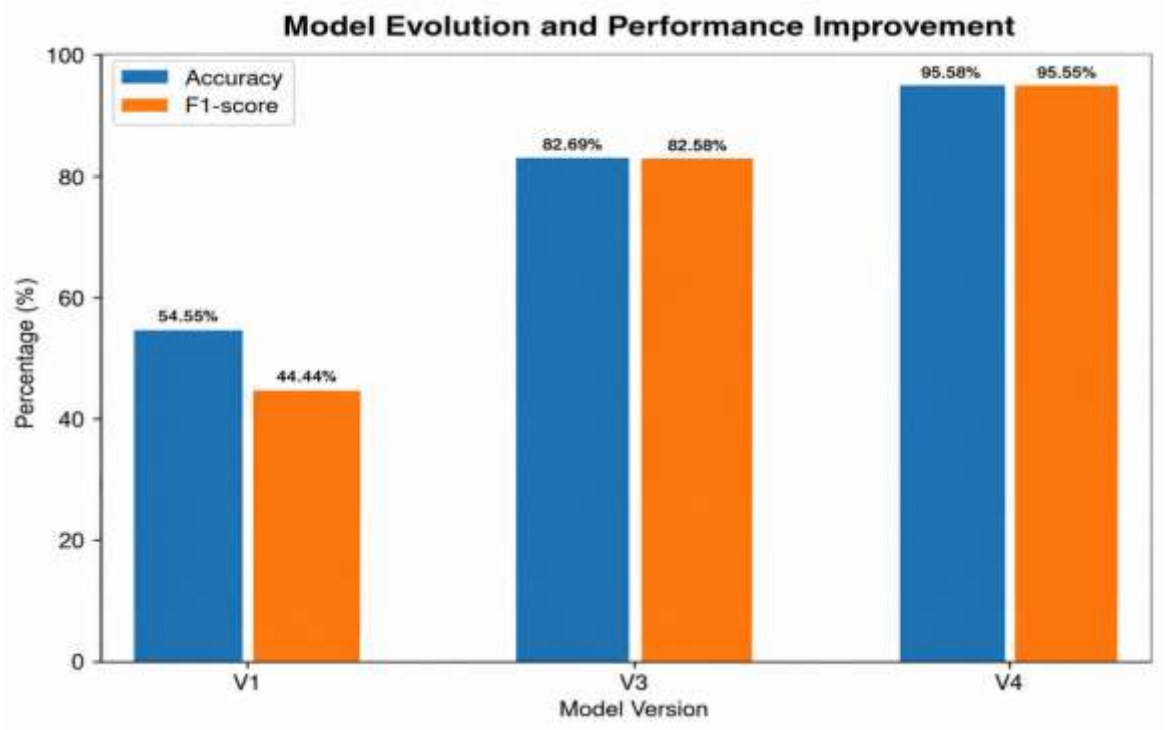

Figure 4. Model evolution and performance improvement across classifier versions

5.4 Confusion Matrix Analysis

The confusion matrix provides a more detailed view of the classifier's prediction behavior. The final model correctly classified 247 vulnerable samples as vulnerable and 250 safe samples as safe. It produced 10 false positives, where safe snippets were incorrectly classified as vulnerable, and 13 false negatives, where vulnerable snippets were incorrectly classified as safe.

The low number of false positives indicates that the classifier does not excessively over-report vulnerabilities. At the same time, the low number of false negatives shows that most unsafe memory-handling patterns were successfully detected. This is important in security analysis because undetected vulnerable code may represent a potential risk. Table 6 summarizes the confusion-matrix values.

Table 6. Confusion Matrix Summary

Result Type	Count
True Positives	247
True Negatives	250
False Positives	10
False Negatives	13

5.5 Practical C-versus-Rust Experimental Results

The practical experiments compared C and Rust across five representative memory-safety scenarios: buffer overflow, use-after-free, double-free, null-pointer dereference, and memory leak. The purpose of these experiments was to observe whether unsafe behavior was allowed, detected at runtime, or prevented before execution. In C, unsafe operations often compiled successfully and later appeared as runtime failures, undefined behavior, garbage values, allocator aborts, segmentation faults, or unreleased memory. In contrast, Rust prevented or mitigated several of these failures through bounds checking, ownership and move semantics, Option<T>, and automatic cleanup using Drop. Table 7 summarizes the observed behavior in both languages.

Table 7. Practical C-versus-Rust Results

Scenario	C Result	Rust Result
Buffer Overflow	Runtime stack-smashing failure	Bounds checking prevents invalid access
Use-After-Free	Garbage value or undefined behavior	Compile-time ownership error
Double Free	Allocator abort or runtime failure	Ownership prevents repeated release
Null Pointer Dereference	Segmentation fault	Safe handling with Option<T>
Memory Leak	Memory remains unreleased	Automatic cleanup with Drop

5.6 Interpretation of Results

The results show a clear difference between C's manual memory-management model and Rust's language-enforced safety model. In C, memory safety depends heavily on developer discipline, and many unsafe patterns are detected only after execution begins. In Rust, several memory-safety checks are shifted earlier to compile time or handled safely through language mechanisms.

For buffer overflow, C produced runtime stack-smashing behavior, while Rust prevented invalid access through bounds checking. For use-after-free and double-free, C allowed unsafe memory behavior that resulted in garbage values or allocator failure, while Rust prevented these patterns through ownership rules. For null-pointer dereference, C produced a segmentation fault, while Rust used Option<T> for safe handling. For memory leaks, C allowed memory to remain unreleased, while Rust automatically cleaned up owned resources using Drop.

The AI results also strengthen the proposed framework. The classifier achieved high accuracy, precision, recall, and F1-score on the controlled dataset. In addition, the model-evolution analysis shows clear improvement from the initial text-based classifier to the final hybrid model. Therefore, KernelMemSafe-AI provides an integrated evaluation approach: the AI component identifies unsafe C memory patterns, while the Rust experiments show how similar patterns can be prevented or mitigated through language-level safety mechanisms.

6. DISCUSSION

The results of this study demonstrate that memory safety can be improved more effectively when vulnerability detection and language-level prevention are considered together. The proposed KernelMemSafe-AI framework does not rely only on AI classification or only on Rust's safety model. Instead, it combines both directions to provide a more complete view of memory-safety behavior in low-level and kernel-oriented software.

The AI-assisted classifier achieved strong performance across all evaluation metrics, with 95.58% accuracy, 96.11% precision, 95.00% recall, and a 95.55% F1-score. These results show that the hybrid approach can successfully identify unsafe memory-handling patterns in the controlled dataset. The low number of false positives indicates that the classifier does not excessively classify safe snippets as vulnerable, while the low number of false negatives suggests that most vulnerable snippets were correctly detected. However, these results should be interpreted within the scope of the study. The classifier was evaluated on a controlled proof-of-concept dataset and therefore should not be considered a replacement for production-level vulnerability scanners or real-world static-analysis tools.

The practical C-versus-Rust experiments further clarify the difference between manual and language-enforced memory safety. In the C implementations, unsafe memory operations frequently compiled successfully and only appeared as failures during runtime. These failures included stack-smashing detection, segmentation faults, allocator-level aborts, garbage values, and unreleased memory. This behavior reflects the permissive nature of C, where the compiler often allows unsafe memory patterns and places the responsibility for correctness on the developer.

In contrast, Rust shifted several memory-safety checks from runtime failure to compile-time prevention. Ownership and move semantics prevented use-after-free and double-free errors. Bounds checking prevented invalid index access. The Option<T> type encouraged

explicit handling of absent values instead of direct null pointer dereferencing. The Drop mechanism supported automatic cleanup of owned resources. These mechanisms show that Rust does not merely detect memory errors after they occur, but prevents or reduces several classes of memory-corruption patterns before they can affect execution.

A key observation from this study is that AI-based vulnerability detection and Rust-based prevention address different but complementary aspects of the memory-safety problem. AI can help identify suspicious or unsafe patterns in existing C/C++ code, especially in legacy codebases where manual auditing may be difficult. Rust, on the other hand, provides structural safety mechanisms that reduce the likelihood of introducing similar errors into new code. Therefore, the combination of AI-assisted analysis and memory-safe programming can support both the modernization of legacy systems and the development of safer future kernel components.

The findings also highlight the importance of maintaining a realistic scope when evaluating memory-safety solutions. Although Rust provides strong safety guarantees, it does not automatically eliminate all risks in kernel development. Low-level operations, hardware interaction, foreign-function interfaces, and legacy C interoperability may still require unsafe blocks. Similarly, AI-based detection can support analysis, but its performance depends on dataset quality, feature design, labeling accuracy, and evaluation conditions. For this reason, KernelMemSafe-AI should be understood as a controlled comparative framework rather than a complete industrial security solution.

Overall, the discussion shows that the strongest contribution of this work is its integrated evaluation approach. The AI component provides analytical evidence by detecting vulnerable memory patterns, while the C-versus-Rust experiments provide behavioral evidence by showing how each language responds to representative memory-safety failures. This combination strengthens the argument that memory-safe programming models, supported by AI-assisted analysis, can improve the reliability and security of kernel-oriented software.

7. THREATS TO VALIDITY

This section discusses the main factors that may affect the interpretation and generalization of the results. The proposed KernelMemSafe-AI framework was evaluated under controlled experimental conditions, which helps provide focused and clear analysis of memory-safety behavior.

First, the dataset used in this study is balanced and controlled, consisting of 520 C/C++ and kernel-style code snippets. This structure supports fair evaluation of vulnerable and safe samples. However, real-world kernel codebases may include more complex control flow, larger dependencies, mixed coding styles, and vulnerability patterns that are not fully represented in the current dataset. Therefore, future validation using larger real-world repositories would further strengthen the generalizability of the results.

Second, the AI-assisted classifier should be interpreted as a proof-of-concept vulnerability-pattern detector rather than a production-grade vulnerability scanner. Its performance depends on dataset quality, feature extraction, rule-based indicators, and labeling accuracy. The reported results demonstrate strong effectiveness within the controlled dataset, while broader deployment would require additional testing against larger and more diverse codebases.

Third, the C-versus-Rust experiments were designed as behavioral memory-safety tests rather than performance benchmarks. The experiments focused on observing whether unsafe behavior is allowed, detected at runtime, prevented at compile time, or handled safely.

Therefore, this study does not claim to provide a complete execution-performance comparison between C and Rust.

Finally, although Rust provides strong memory-safety guarantees, it does not eliminate all risks in kernel development. Low-level hardware interaction, foreign-function interfaces, legacy C interoperability, and the use of unsafe blocks may still introduce memory-safety concerns. For this reason, Rust should be viewed as a strong safety-improving language rather than a complete replacement for careful security analysis.

Overall, these factors define the proper scope of the study without reducing the value of the proposed framework. The results provide controlled experimental evidence that AI-assisted detection and Rust safety mechanisms can support the prevention and analysis of memory-corruption risks in kernel-oriented software.

8. CONCLUSION AND FUTURE WORK

This paper presented KernelMemSafe-AI, an AI-assisted comparative framework for evaluating memory-safety behavior in C- and Rust-based approaches within a kernel-development context. The framework combined two complementary components: a hybrid AI-assisted vulnerability-pattern classifier and a practical C-versus-Rust experimental evaluation. The study focused on representative memory-corruption categories, including buffer overflow, use-after-free, double-free, null-pointer dereference, memory leak, and dangerous function usage.

The AI component was evaluated using a controlled and balanced dataset of 520 C/C++ and kernel-style code snippets, consisting of 260 vulnerable and 260 safe samples. Using five-fold cross-validation, the hybrid classifier achieved 95.58% accuracy, 96.11% precision, 95.00% recall, and a 95.55% F1-score. These results indicate that the proposed classifier can effectively identify unsafe memory-handling patterns within the controlled proof-of-concept dataset. However, the classifier is not intended to replace production-grade vulnerability scanners or real-world static-analysis tools.

The practical experiments further demonstrated the behavioral difference between C and Rust. In the C implementations, unsafe memory operations often compiled successfully and appeared during runtime as stack-smashing failures, segmentation faults, allocator aborts, garbage values, or unreleased memory. In contrast, Rust prevented or mitigated several of these failures through ownership, move semantics, bounds checking, Option<T>, and automatic cleanup using Drop. These findings show that Rust shifts many memory-safety checks from runtime failure to compile-time prevention or safer runtime handling.

Overall, the results support the conclusion that combining AI-assisted vulnerability-pattern detection with memory-safe programming mechanisms can improve the analysis and prevention of memory-corruption risks in kernel-oriented software. KernelMemSafe-AI contributes an integrated perspective by linking unsafe C memory patterns with Rust safety mechanisms, providing both analytical and experimental evidence for improving memory safety. Future work will extend this study in several directions.

First, the dataset can be expanded using larger real-world vulnerability datasets and kernel code repositories to improve generalization.

Second, more advanced models, such as transformer-based and graph-based neural networks, can be evaluated and compared with the lightweight hybrid classifier.

Third, future experiments can include real Rust-for-Linux modules and broader kernel-driver scenarios. Finally, AI-assisted refactoring techniques can be explored to support the migration of unsafe C code toward safer Rust-based implementations while preserving functional behavior.

REFERENCES

[1] National Security Agency, “Software Memory Safety,” Cybersecurity Information Sheet, Nov. 2022.

[2] Microsoft Security Response Center, “We Need a Safer Systems Programming Language,” Microsoft Security Blog, Jul. 2019.

[3] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, “An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise,” in Proc. 2024 USENIX Annual Technical Conference (USENIX ATC), Santa Clara, CA, USA, 2024.

[4] Z. Li, A. Narayanan, M. M. Swift, and S. Jha, “Understanding the Security Impact of Rust in the Linux Kernel,” in Proc. Annual Computer Security Applications Conference (ACSAC), 2024.

[5] S. K. Panter and N. U. Eisty, “Rusty Linux: Advances in Rust for Linux Kernel Development,” in Proc. 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2024, pp. 496–502.

[6] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, “Rust Meets Linux: Lessons from an Evolving Experiment,” ACM Transactions on Computer Systems, 2026.

[7] F. Garber, “Rust in the Linux Kernel: Analyzing Rust Implementations of Virtual General Purpose Input Output Drivers,” M.S. thesis, Technische Universität Wien, Vienna, Austria, 2025.

[8] A. Syalim and D. P. Sheradhien, “C vs Rust: Manual vs Automatic Spatial and Temporal Memory Safety,” The Indonesian Journal of Computer Science, vol. 14, no. 2, pp. 2197–2213, 2025.

[9] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries,” in Proc. 17th International Conference on Mining Software Repositories (MSR), Seoul, South Korea, 2020, pp. 508–512.

[10] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep Learning Based Vulnerability Detection: Are We There Yet?,” IEEE Transactions on Software Engineering, vol. 48, no. 9, pp. 3280–3296, 2022.

[11] M. Fu and C. Tantithamthavorn, “LineVul: A Transformer-Based Line-Level Vulnerability Prediction,” in Proc. 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 2022, pp. 608–620.

[12] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, “DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection,” in Proc. 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Hong Kong, China, 2023, pp. 654–668.

[13] H. Hanif and S. Maffeis, “VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection,” in Proc. International Joint Conference on Neural Networks (IJCNN), 2022.

[14] Y. Li, S. Wang, and T. N. Nguyen, “Vulnerability Detection with Fine-Grained Interpretations,” in Proc. 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021.

[15] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, “BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection,” Information and Software Technology, vol. 136, 2021.

[16] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, “Software Vulnerability Discovery via Learning Multi-Domain Knowledge Bases,” IEEE Transactions on Dependable and Secure Computing, vol. 18, no. 5, pp. 2469–2485, 2021.

[17] C. Liang, L. Wang, Y. Zhang, and Z. Jin, “Survey of Source Code Vulnerability Analysis Based on Deep Learning,” Computers & Security, vol. 146, 2025.

[18] H. Su, X. Li, Y. Zhang, and J. Wang, “Source Code Vulnerability Detection Based on Deep Learning: A Review,” Discover Computing, 2026.
[19] X. He, Y. Wang, and L. Zhang, “An Improved Software Source Code Vulnerability Detection Model Based on Multi-Feature Screening and Integrated Sampling,” Sensors, vol. 25, no. 6, 2025.
[20] Z. Sheng, K. Tian, and D. Lo, “LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights,” ACM Computing Surveys, 2025.