machine learning & knowledge extraction

MDPI

Article

Machine Learning-Based Vulnerability Detection in Rust Code Using LLVM IR and Transformer Model

Young Lee *, Syeda Jannatul Boshra, Jeong Yang, Zechun Cao and Gongbo Liang

Department of Computational, Engineering, and Mathematical Sciences, Texas A&M University-San Antonio, One University Way, San Antonio, TX 78224, USA; sbosh01@jaguar.tamu.edu (S.J.B.); jyang@tamusa.edu (J.Y.); zcao@tamusa.edu (Z.C.); gliang@tamusa.edu (G.L.)

* Correspondence: ylee@tamusa.edu

Abstract

Rust's growing popularity in high-integrity systems requires automated vulnerability detection in order to maintain its strong safety guarantees. Although Rust's ownership model and compile-time checks prevent many errors, sometimes unexpected bugs may occasionally pass analysis, underlining the necessity for automated safe and unsafe code detection. This paper presents Rust-IR-BERT, a machine learning approach to detect security vulnerabilities in Rust code by analyzing its compiled LLVM intermediate representation (IR) instead of the raw source code. This approach offers novelty by employing LLVM IR's language-neutral, semantically rich representation of the program, facilitating robust detection by capturing core data and control-flow semantics and reducing language-specific syntactic noise. Our method leverages a graph-based transformer model, GraphCodeBERT, which is a transformer architecture pretrained model to encode structural code semantics via data-flow information, followed by a gradient boosting classifier, CatBoost, that is capable of handling complex feature interactions—to classify code as vulnerable or safe. The model was evaluated using a carefully curated dataset of over 2300 real-world Rust code samples (vulnerable and non-vulnerable Rust code snippets) from RustSec and OSV advisory databases, compiled to LLVM IR and labeled with corresponding Common Vulnerabilities and Exposures (CVEs) identifiers to ensure comprehensive and realistic coverage. Rust-IR-BERT achieved an overall accuracy of 98.11%, with a recall of 99.31% for safe code and 93.67% for vulnerable code. Despite these promising results, this study acknowledges potential limitations such as focusing primarily on known CVEs. Built on a representative dataset spanning over 2300 real-world Rust samples from diverse crates, Rust-IR-BERT delivers consistently strong performance. Looking ahead, practical deployment could take the form of a Cargo plugin or pre-commit hook that automatically generates and scans LLVM IR artifacts during the development cycle, enabling developers to catch vulnerabilities at an early stage in the development cycle.

Keywords: Rust; LLVM IR; vulnerability detection; code embedding; GraphCodeBERT; machine learning

check for updates

Academic Editors: Francesco Buccafurri and Nikolaos Pitropakis

Received: 23 May 2025
Revised: 6 July 2025
Accepted: 19 July 2025
Published: 6 August 2025

Citation: Lee, Y.; Boshra, S.J.; Yang, J.; Cao, Z.; Liang, G. Machine Learning-Based Vulnerability Detection in Rust Code Using LLVM IR and Transformer Model. *Mach. Learn. Knowl. Extr.* **2025**, *7*, 79. https://doi.org/10.3390/make7030079

Correction Statement: This article has been republished with a minor change. The change does not affect the scientific content of the article and further details are available within the backmatter of the website version of this article.

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Rapid software development has increased the risk of overlooked bugs, making timely vulnerability detection both critical and challenging. Developers and educators employ hundreds of technologies to implement secure coding practices [1,2] and detect and patch vulnerabilities in code, but overcoming them completely remains challenging. Despite

recent efforts, existing vulnerability detection methods still heavily rely on analyzing source code directly, which may miss deeper vulnerabilities evident at the IR level. In this paper, we propose Rust-IR-BERT, a novel machine learning pipeline specifically designed to identify security vulnerabilities in Rust code which operates directly on IR to improve accuracy in detecting false positives. Unlike compiler-embedded tools like Rudra [3] that focuses on Rust's HIR/MIR, Rust-IR-BERT leverages LLVM IR to detect deeper, catching lower-level (e.g., certain use-after-free patterns) vulnerabilities more effectively that higher-level IR might have already optimized or abstracted away. By extracting code semantics via GraphCodeBERT embeddings [4], a pretrained transformer model optimized for code understanding and employing CatBoost classifier [5], a gradient-boosting algorithm renowned for handling heterogeneous features is employed—our framework achieves superior detection accuracy. Furthermore, by employing LLVM IR, our model generalizes more effectively across diverse codebases, overcomes the limitations of traditional source-level analysis, and reduces the noise introduced by high-level syntax differences. In this study, we primarily focus on vulnerabilities related to memory safety issues (use-after-free; double-free) and misuses of unsafe block and concurrency errors, which represent common security concerns in Rust.

Rust is considered to be the safest system-level programming language of this time [6]. Bugden and Alahmar have conducted an analysis among leading programming languages to evaluate safety and performance and found that Rust outperforms the other languages [7]. They mentioned Rust as the safest, especially in concurrent environments where Rust's data race prevention can avert many software bugs and vulnerabilities. This language is referred to as the "safest" due to its ownership model, memory safety guarantees (e.g., use-after-free; double free), and strict compile-time checks [8,9]. Rudra [3] recently revealed that by employing unsafe blocks in error-prone spots a lot of memory-safety bugs in the Rust ecosystem can be identified. While Rust eliminates many memory safety issues at compile time, there are still potential risks such as 'Unsafe Rust' and 'Concurrency Bugs.' 'Unsafe Rust' refers to situations where developers sometimes use the unsafe keyword to bypass Rust's safety guarantees, which can introduce security risks. Qin et al. (2020) presented an extensive empirical survey of memory and thread safety practices in real-world Rust codebases, detailing how developers employ unsafe code and the common misuse patterns that trigger safety breaches [10]. Regarding 'Concurrency Bugs', Rust enforces strict rules to prevent data races, but complex multi-threaded programs can still have logical concurrency bugs that are not caught by the compiler [11].

Despite Rust's safety guarantees, even within the safe Rust block, it is possible to violate this guarantee of safety, which has become a significant concern. Logic errors in security-critical applications can also lead to serious vulnerabilities, such as race conditions or unexpected state transitions [6,12]. Since Rust is increasingly used in security-critical domains like operating systems, web browsers, and blockchain, it is crucial to develop some techniques for making sure that even the safest programming language can also be double checked and vulnerability detection can be conducted before publishing the software.

Rust initially lacks a mechanism against timing attacks, which could lead to the iteration time varying based on the data [13]. Figure 1 presents Rust's == operator on `&str` that performs a byte-by-byte comparison and short-circuits whenever a mismatch is found. Therefore, its execution time varies with the length of the matching prefix. A potential attacker measuring these timing differences can recover the secret one byte at a time. A single-line fix can prevent the threat by using a constant-time comparison (e.g., `subtle::ConstantTimeEq::ct_eq`) so that every byte is compared without early exit. However, both safe and unsafe blocks of Rust are extremely useful when it comes to developing

software. The combination of safe and unsafe Rust provides a memory safety guarantee while enabling its usability for various purposes, including system programming [14].

```rust
fn verify_password(attempt: &str, password: &str) -> bool {
    attempt == password
}

fn main() {
    let user_input = "guess";           // e.g., read from user
    let correct    = "s3cr3tP@ssw0rd";  // stored secret
    if verify_password(user_input, correct) {
        println!("Access granted");
    } else {
        println!("Access denied");
    }
}
```

Figure 1. Timing-attack vulnerability in safe Rust string comparison.

Addressing the challenges, in this research, we conduct an empirical investigation of safety issues in real-world Rust programs by evaluating the crates. Unlike most prior methods relying solely on raw source code or abstract representations [15–17], our novel pipeline uniquely leverages IR embeddings combined with GraphCodeBERT and CatBoost classifier, explicitly capturing deeper semantic features and improving detection reliability and accuracy. Incorporating these 768-dimensional encoded LLVM IR files with CatBoost-we achieve robust detection of vulnerable patterns of Rust code samples. This integration of LLVM IR, GraphCodeBERT embeddings, and CatBoost classification with threshold optimization is novel for Rust vulnerability detection. Later in this study, experimental results demonstrate that our updated approach delivers significantly higher accuracy and F1-scores than prior baselines.

2. Background

Rust was developed to provide strong guarantees around memory and thread safety without compromising with the performance. While Rust-specific features such as ownership, borrow checker, and the unsafe keyword underpin Rust's safety mechanism, vulnerabilities persist, especially when unsafe constructs or external libraries are employed without proper concern. This section provides some background of Rust's safety and unsafety mechanisms, Language-Independent Intermediate Representation (IR), and Vulnerability Databases and Embeddings.

2.1. Rust's Safety Mechanism and Unsafe Rust

Rust ensures memory safety through its ownership system, which statically enforces that each memory allocation has either one mutable owner or multiple immutable owners, eliminating both use-after-free and double-free errors. The borrow checker enforces reference lifetime validity, preventing dangling pointers and data races by disallowing overlapping mutable and immutable references. While the unsafe keyword enables evading these guarantees for low-level operations, it requires manual adherence to safety invariants. Formal verification underpins the Rust model: the RustBert project [18] provides a machine-checked proof (in Coq) using concurrent separation logic to validate ownership, borrowing, and interior mutability. GhostCell extends this by decoupling aliasing permissions from data storage, enabling safe shared mutability without runtime checks [19]. These described features, compile-time checks, verified guarantees, and controlled access rules make Rust a super safe language that ensures the utmost safety. Rust has become increasingly popular in system software in recent years due to its safety and performance advantages. Mi-

crosoft is currently exploring investing in Rust as a replacement for C/C++ considering its memory-safety features, which has recently been announced by the Microsoft Security Response Center [20]. Amazon's AWS team has extensively adopted Rust to implement performance-sensitive components, leveraging the language's guarantees around memory safety and zero-cost abstractions [21].

Safe Rust offers strong memory safety and flexible restrictions, but it is not suitable for maintaining shared mutable references in system programming or reference counting. Unsafe Rust escapes the Rust compiler's check and requires programmers to ensure memory safety. Rust labels five core operations: dereferencing raw pointers, calling external functions, accessing mutable statics, implementing unsafe traits, and manipulating unions as "unsafe," where the compiler's standard guarantees are suspended [22]. Within these blocks, the borrow checker and lifetime analysis do not enforce memory-safety variables, causing the developers to be solely responsible for ensuring software safety. Eventually, memory-safety bugs can easily be introduced by any error in these unchecked memory-related operations. To mitigate these risks, developers should carefully keep unsafe regions as minimal as possible, encapsulate them within thoroughly reviewed safe abstractions, and document the precise safety invariants they rely on. Evans et al. (2020) claim that, although fewer than 30% of Rust libraries explicitly use the unsafe keyword, the way unsafety can spread through function calls still challenges Rust's promise of complete static memory safety [9].

Figure 2 presents an example code that converts a mutable reference to x into a raw pointer (*mut i32), bypassing Rust's ownership and borrowing constraints, which usually restrict concurrent mutable access. Two threads then enter unsafe blocks to dereference and access the same memory location simultaneously, resulting in unsynchronized writes. These operations cause a data race condition, which is explicitly stated as undefined behavior in Rust, because at least one thread writes while another may read or write concurrently. This situation can be avoided by using synchronization primitives like Mutex<i32> or atomic types (AtomicI32) to protect shared state, which guarantees safe, unaffected access between threads.

```
use std::thread;
fn main() {
    let mut x = 0;
    let ptr = &mut x as *mut i32;  // raw mutable pointer
    // Spawn two threads that both mutate *ptr without synchronization
    let t1 = thread::spawn(move || unsafe { *ptr += 1; });
    let t2 = thread::spawn(move || unsafe { *ptr += 2; });

    t1.join().unwrap();
    t2.join().unwrap();
    println!("Result: {}", x);
}
```

Figure 2. Data-race via unsynchronized raw-pointer access in unsafe Rust.

2.2. Vulnerability Databases and LLVM IR

RustSec Advisory Database is a community-maintained repository of security advisories for Rust crates published on [crates.io]. Every advisory has a distinct RUSTSEC-YYYY-NNNN tag, and when accessible, it usually contains metadata like CVE IDs, URLs to repair changes, and impacted version ranges [23]. This database enables Rust developers to audit their dependencies for known vulnerabilities via tools like cargo audit and integrates with the Open Source Vulnerabilities (OSV) [24] schema to facilitate machine-readable consumption. This database connects with the Open Source Vulnerabilities (OSV) schema

to enable machine-readable consumption and allows Rust developers to use tools like cargo audit to audit their dependencies for known vulnerabilities.

Open Source Vulnerabilities (OSV) database is an initiative led by Google that provides a unified, precise, and distributed approach to publishing and registering vulnerability information across multiple ecosystems [24]. OSV entries are represented in a standardized JSON schema, reference one or more CVE IDs, and can be queried via a public API or consumed as bulk archives. Through the integration with data from RustSec, Advisories, and the National Vulnerability Database (NVD), OSV simplifies automated vulnerability management for both open-source maintainers and application developers [25]. The OSV and RustSec Advisory-db. are both essential to preserving high-quality datasets for vulnerability research. Early studies demonstrated the usefulness of the RustSec Advisory and OSV in vulnerability predictive modeling, proving that information from these databases can anticipate which software components are most likely at risk [6]. For our approach, we developed an automated script to collect vulnerable and non-vulnerable Rust source codes from OSV and RustSec Advisory-db. It clones each repository, calls the OSV API to find vulnerable and fixed versions, checks out the code before and after each fix, compiles each version to LLVM IR with rustc –emit=LLVM IR(with minimal corrections), and then saves the original .rs files alongside the generated .ll files locally for use in the embedding pipeline. The same process is applied to the RustSec Advisory-db. A recent approach, ContraFlow [26], uses CVE-labeled samples from OSV advisory (including RustSec entries) to train contrastive, path-sensitive code embeddings over value-flow graphs, significantly boosting vulnerability detection accuracy. Graph-based techniques like Devign [27] have employed rich semantic representations obtained from vulnerability databases to learn graph embeddings that greatly increase detection accuracy, while deep learning systems like VulDeePecker [16] have used CVE-labeled code devices to train BLSTM models for vulnerability detection.

Language-Independent Intermediate Representation (LLVM IR) is a language-agnostic SSA-based "portable assembly" that provides a consistent low-level view of programs, enabling transparent, lifelong analysis and transformation by exposing typed arithmetic, memory operations, and control structures in a uniform form [28]. It is simply a low-level, platform-independent code representation that sits between source code and machine code and is extensively used in compiler optimizations and program analysis. We leverage LLVM IR in our workflow because it offers a structured yet simplified view of Rust programs, faithfully capturing control-flow and data-flow semantics crucial for accurate vulnerability detection.

In our pipeline, each Rust crate, both before and after patch commits, are compiled into LLVM IR, from which a pretrained BERT model extracts semantic embeddings that highlight patterns associated with security flaws. These embeddings are then passed to a classifier, which helps the pipeline to detect vulnerabilities more accurately. Furthermore, because IR abstracts away high-level constructs such as generics and borrow-checker lifetimes, the classifier can generalize across diverse crate ecosystems without being biased by Rust-specific syntax [29]. Previous studies have demonstrated that neural models trained on IR functions outperformed source-level token models by more than 12% in precision and recall for vulnerability detection, underscoring IR's structured semantic richness [30].

2.3. Embedding-Based Vulnerability Classifier

Our embedding-based classifier employs CatBoost [5]. Pretrained, transformer-based code models have delivered outstanding performance on software development and code-analysis tasks such as code summarization, feature extraction, and predictive code generation. GraphCodeBERT, a pretrained model for programming language that considers

the inherent structure of code. Instead of taking a syntactic-level structure of code like abstract syntax tree (AST), we leverage semantic-level information of code (i.e., data flow) for pretraining [4].

2.4. Vulnerability Detection Tools

Compared with traditional models, our unique Rust-IR-Bert approach has proven and reliable special features. For example, it does not only rely on raw Rust code yet compiles Rust code to an intermediate representation which turns the source code into a detailed version. Also it does not require large-scale self-replication and distribution but only needs to be placed at the source—.ll or .rs file into the ML pipeline to determine the safety. A well-known open source tool AIBughunter [31], implemented in academia in 2022, is a representative method based on developer signature matching which published a plugin inside the IDE to assist developers during coding. It has been trained on over 188,000 C/C++ functions, whereas our model supports Rust, which is increasingly used for secure systems.

As Rust being the safest language in the recent era, our proposed methodology is one step ahead of the current big projects. Moreover, AiBugHunter uses raw function text for training the model and predicting, due to which it might miss deeper semantic relationships, such as data and control flow dependencies, which graphs or IR can capture. This limits the tool's ability to detect vulnerabilities in tracing how data moves through or how the control structure of a program enables unsafe states.

Over the past few years, several ML-driven tools have targeted Rust (and C/C++) vulnerabilities using diverse representations and approaches. HALURust prompts a 7-billion-parameter LLM to "hallucinate" vulnerability reports on Rust code, then fine-tunes on those examples, resulting in an F1 score uplift of 10% over source-only attempts [32]. Unsafe's Betrayal parses Rust binaries into token sequences and fine-tunes RoBERTa to pinpoint unsafe functions—achieving a precision–recall AUC of 80% for unsafe-code detection and 62% when adapted to known Rust vulnerabilities [14]. VulBERTa pretrains RoBERTa on millions of lines of C/C++ source with their custom code-aware tokenizer and then fine-tunes for vulnerability classification, which achieves 99.6% F1 on the muVulDeePecker benchmark [16]. AI4VA transforms each function of the code into a Code Property Graph (CPG) and trains a Gated Graph Neural Network on those, outperforming traditional static analyzers on standard vulnerability benchmarks [33].

Our model uses LLVM IR, which exclusively captures low-level behavior, improving semantic understanding. Cipollone [34] introduced a transformer-based framework that classifies CVE-linked GitHub issues using embedding models and XGBoost [35], demonstrating that natural-language signals can provide early vulnerability alerts. In contrast, our ML-driven pipeline employs CatBoost as a classifer. The SySeVR model [15] presents a structured approach to vulnerability detection in C/C++ programs by analyzing specific code patterns using deep learning models. It convert semantics-based vulnerable codes into vector representations using techniques like word2vec. SCL-CVD fine-tunes GraphCodeBERT using a supervised contrastive loss combined with R-Drop on data flow graph representations of source program code, achieving relative improvements of 0.48–3.42% in accuracy over baselines, while reducing fine-tuning time by up to 93% [36].

In contrast, our method leverages the intermediate representation of code derived from Rust and advanced embedding techniques to potentially capture a more comprehensive view of program behavior. This could lead to improved detection of a wider range of vulnerabilities. Also, SySeVR does not assign specific CVE IDs, whereas we assign CVE IDs to detected vulnerabilities which makes it convenient for the developers to further analyze the issue. Android researchers extract static code features from Android APKs

with Androguard and trains a CatBoost Classifier to detect ransomware, reporting over 95% accuracy on benchmark datasets [37]. Conversely, our method leverages same CatBoost classification to identify vulnerable snippets, achieving an accuracy of 98.6%. Our system is a more versatile and scalable solution for modern software vulnerability detection.

3. System Architecture

Figure 3 illustrates the end-to-end vulnerability detection pipeline, composed of four primary stages: (1) IR Generation, (2) Embedding Extraction, (3) Classification, and (4) Threshold Optimization. The justification of each design choice is given below:

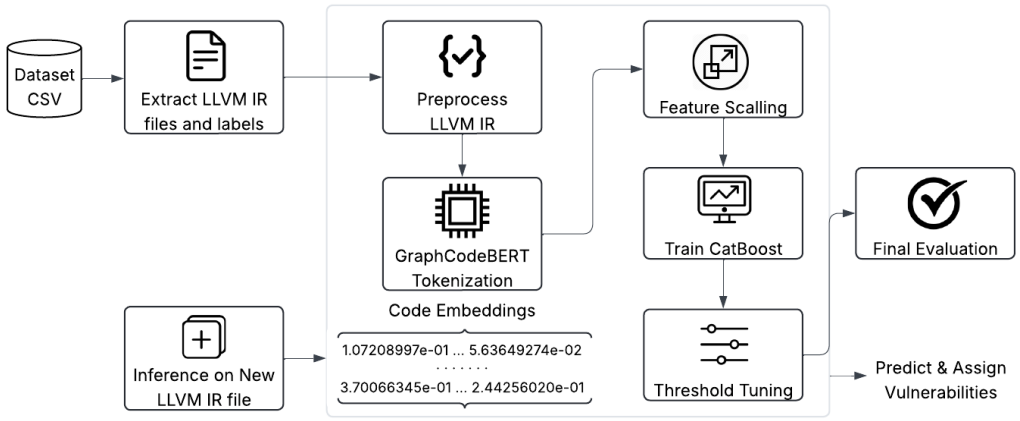

Figure 3. ML pipeline.

The pipeline begins by compiling Rust source code into LLVM IR via rustc –emit=llvm-ir, preserving program semantics wile providing a lower-level representation. The resulting IR text is tokenized and processed by a pretrained graph-based transformer model GraphCodeBERT's IR-aware tokenizer, whose graph-guided attention mechanism leverages code data-flow to produce a 768-dimensional semantic embedding. These embeddings are then normalized via a standard scaler to ensure zero mean and unit variance before being passed to a CatBoost classifier, which is a great choice for handling categorical data and reduces overfitting [5]. Prior to selecting CatBoost, we conducted a comparative evaluation against XGBoost and Random Forest on the same embeddings. We evaluated CatBoost against XGBoost and Random Forest using 5-fold stratified cross-validation on the full dataset, observing that CatBoost achieved the best mean accuracy (0.982 ± 0.008) and recall metrics. The CatBoost classifier's strong feature interaction handling enables it to learn complex patterns for binary vulnerability identification. We refine the model's decision threshold using a held-out 20% validation split by taking over the classification probability from 0.1 to 0.9 and computing the corresponding F1-score at each point, thus balancing precision and recall and selecting an optimal cutoff (found to be 0.35) for final predictions. We also considered a separate pipeline without inserting GraphCodeBERT embeddings; however, initial results indicated significantly degraded performance of >15% drop in F1-score, underscoring the embedding's contribution.

During inference, this pipeline can process new, unseen LLVM IR snippets to determine whether the newly injected LLVM IR file is buggy or not. By integrating transformer-based code semantics with an ensemble classifier, this architecture captures structural code information while delivering strong generalization and precise decision thresholds for identifying vulnerabilities in Rust code.

4. Research Methodology

4.1. Research Questions

This study aims to find answers to the following research questions.

Q1: What is the detection accuracy of our GraphCodeBERT + CatBoost pipeline?
Q2: How does our hybrid Rust-IR-Belt approach differ fundamentally from existing static-analysis frameworks and machine learning methods?
Q3: Can this system reliably identify real Rust vulnerabilities in unseen code?
Q4: How are Rust-derived LLVM IR snippets encoded into feature vectors for classification?

4.2. Data Collection and Preprocessing

Collecting vulnerable and patched Rust code samples from ecosystem: Data collection is a critical component of this study, as high-quality labeled data underpins effective ML models for vulnerability detection. We leveraged the RustSec Advisory Database—a curated, community-driven repository of Rust crate vulnerability advisories to extract real-time examples and CVE mappings. Concurrently, we consumed the Open Source Vulnerabilities (OSVs) API to normalize advisory data, extracting CVE identifiers, version ranges, and vulnerability categories in a machine-readable JSON schema. For each crate version, source archives were downloaded via the crates.io API endpoint and uncompressed with Python's tarfile module. We programmatically extracted all .rs files within each src/directory, wrapping them with dummy stub modules and an injected fn main() to guarantee standalone compilability. Each vulnerable or patched snippet was then compiled to LLVM IR using rustc –emit=llvm-ir, preserving CVE labels and vulnerability metadata in a flat local directory. To further diversify our dataset, we manually curated additional raw Rust code samples from large open-source GitHub projects, verifying both vulnerable and fixed versions to capture real-world patterns. This combined strategy generated over 2300 labeled .rs/.ll pairs for downstream embedding and classification tasks, ensuring comprehensive coverage across the Rust ecosystem. Figure 4 illustrates this data collection pipeline.

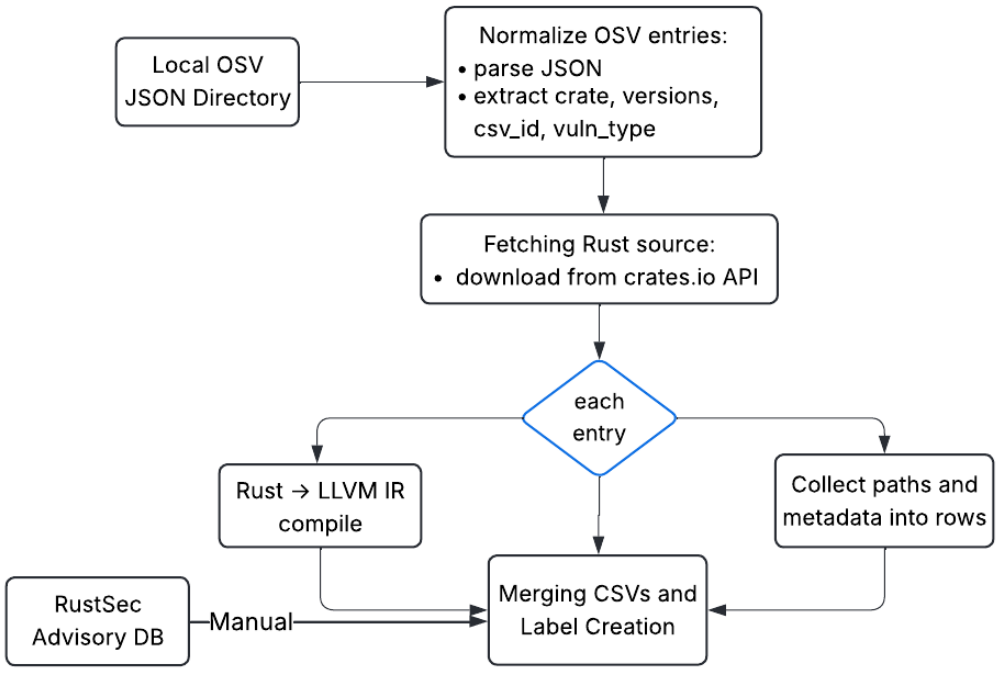

Figure 4. Data collection pipeline.

The automated Python script begins by normalizing a local OSV JSON directory, parsing each entry to extract crate names, version ranges, CVE IDs, and vulnerability types, and then fetches the corresponding source archives from the crates.io API via the requests library. For each entry, the script compiles the Rust code into LLVM IR using rustc –emit=llvm-ir, after wrapping snippets to ensure standalone compilability, and extracts and filters.rs files with Python's tarfile module. In parallel, we manually collect relevant entries from the RustSec advisory database. Finally, all paths and metadata are merged into a single CSV file, which is being stored locally, capturing CVE identifiers, source paths, and advisory descriptions as of 25 April 2025, enabling scalable, repeatable updates as new advisories appear. The bar chart from Figure 5 presents the top 10 CVE IDs and other vulnerable tags the model successfully detected (count of .ll files flagged with each CVE). The most common are RUSTSEC-2022-0008 and GHSA-x4nm7s-fmx8m (500 files each), followed by other Rust CVEs. This suggests these issues are prevalent in the codebase. Several other CVEs (e.g., CVE-2023-22466 and CVE-2021-31891) had smaller counts, demonstrating that the model recognizes a range of distinct vulnerabilities. Overall, this distribution indicates that the classifier both distinguishes vulnerable code and recalls the specific CVE labels learned during training.

Rust Snippet Wrapping: To prepare each Rust snippet for LLVM IRemission, we automatically embed it within a minimal scaffolding layer. This involves prepending dummy stubs for commonly missing modules and injecting an fn main() entry point when necessary. A dedicated pipeline was developed to automate the handling of these issues, adding for a missing main, injecting the necessary boilerplate, writing the augmented .rs file into a local folder, and then invoking rustc to produce the corresponding .ll file—all without manual intervention.

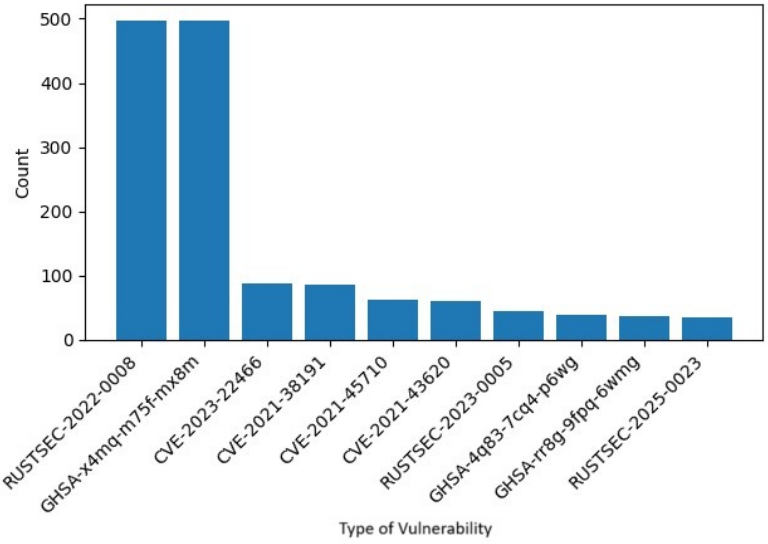

Figure 5. Bar chart of top 10 types of vulnerabilities detected by the model.

Key Integration Challenges:

- Missing `main()` Functions: Many examples consist solely of library functions. Our wrapper detects any code lacking a main function and appends a minimal fn main() so that rustc –emit=LLVM IRwill succeed.
- Missing Contextual Definitions: Snippets sometimes refer to types or traits defined elsewhere. We include lightweight dummy modules (e.g., mod reactor) to satisfy these external references.

- Feature and Flag Variability: Different crates target varying Rust editions or feature sets. By standardizing on the 2018 edition and using a uniform stub approach, we avoid per-snippet compiler flag adjustments.
- Project-Level Dependencies: Some code relies on broader project settings or build scripts. Where isolated compilation fails, our script logs and skips those cases, ensuring that only self-contained snippets proceed.

Data Labeling: Each code snippet in our dataset is tagged as either vulnerable or safe. For vulnerabilities, we attach the official CVE identifier as sourced from advisory metadata, ensuring each example is linked to its real-world issue. Safe samples come from stable crate versions with no reported security advisories. This clear, consistent labeling gives our model the precise ground truth it needs to learn how to distinguish secure code from insecure code.

To define non-vulnerable or safe samples, we relied on codebases without reported CVEs, acknowledging the limitation that unreported and unseen vulnerabilities could potentially exist within these samples. Moreover, some code samples required minor manual adjustments, such as inserting dummy main() functions, which were necessary for ensuring IR generation completeness. These adjustments were minimal and aimed to standardize IR extraction to streamline the data collection process. Specifically, our dataset contains 143 unique CVE identifiers sourced from multiple crates, demonstrating broad coverage of known vulnerability types, yet a broader range of projects would further accelerate in generalization.

4.3. Code Preprocessing and Representation

Studies have shown that large language models can be sensitive to minor code changes, such as whitespace modifications or renaming functions, which can affect their vulnerability detection capabilities. As a result, we added a function to remove unnecessary comments and blank/white spaces from the LLVM IR code [38]. Raw LLVM IR often contains comments, metadata, and extra whitespace that are irrelavant and sometimes distracting for training a classifer. In order to produce a stable, normalized input for GraphCodeBERT, we first strip out all comment lines (lines that begins with ;) and collapse consecutive blank lines (see Listing 1). This lightweight cleanup preserves the actual instructions and data-flow structure while removing noise that could otherwise bias the embedding. After cleaning, each IR file is tokenized with GraphCodeBERT's native tokenizer, and the CLS token embedding is extracted to represent the entire snippet individually.

Listing 1. Preprocessing .ll files.

```
def load_and_clean_ir(path):
    with open(path, 'r', encoding='utf-8', errors='ignore') as f:
        return "".join(line for line in f if not line.lstrip().
            startswith(';'))
```

4.4. Embedding-Based Feature Extraction

Embeddings translate complex code structures into fixed-dimensional numerical vectors that preserves both syntactic and semantic relationships within the code. Unlike characters or token-based representation, embeddings encapsulate semantic functionalities and modules that interacts closely in the program. It enables downstream classifiers to determine subtle vulnerability patterns. Unlike traditional methods that treat code as sequences of characters, embeddings capture the semantic relationships between parts of the code [39].

Tokenization and Data-Flow Construction—Extracting GraphCodeBERT Embeddings:

GraphCodeBERT is a pretrained transformer model specially designed to capture both data-flow and control-flow dependencies in source code [4]. Whereas conventional tokenizers treat code as flat token sequences, GraphCodeBERT integrates a graph-based representation of variable usages and control edges, producing embeddings that reflect the program's operational semantics. This richer representation helps the model recognize vulnerability-triggering constructs that depend on data propagation or execution paths. We preprocess each Rust-derived LLVM IR by stripping comments and normalizing constants, then tokenize the cleaned IR with the HuggingFace GraphCodeBERT tokenizer (truncating or padding to 512 tokens) (see Listing 2).

Listing 2. Extraction of CLS embedding from LLVM IR.

```
for each row in df:
    ir = load_and_clean_ir(row.rs_fullpath)
    toks = tokenizer(ir, return_tensors="pt", truncation=True,
        max_length=512)
    with torch.no_grad():
        out = model(**toks.to(device))
    emb = out.last_hidden_state[0,0,:].cpu().numpy()  # CLS vector
    embs.append(emb)
```

GraphCodeBERT augments the usual token sequence with data-flow and control-flow edges, so its transformer encoder builds contextualized hidden states which reflects both semantic and structural program features. We take the output of GraphCodeBERT's final transformer layer corresponding to the [CLS] token as a fixed 768-dimensional embedding for each snippet, which provides a compact representation of the entire code fragment. These CLS embeddings are then stacked into an 768× feature matrix and passed to our downstream CatBoost classifier for vulnerability prediction.

4.5. Experimental Setup

All experiments were conducted on Google Colab Pro using an NVIDIA T4 GPU. Our codebase runs on Python 3.8 with PyTorch 1.x and Hugging Face Transformers 4.31 (microsoft/graphcodebert-base), alongside CatBoost 1.0.6. The Colab Pro GPU instance accelerated both embedding extraction and classifier training.

We evaluated our approach on a curated dataset of Rust labeled as vulnerable or safe, derived from publicly disclosed CVEs and safe code examples. We evaluated on 2305 Rust functions (769 vulnerable, 1536 safe) drawn from CVE-linked and benign code. Splitting was stratified by label (70% train, 15% validation, 15% test; random_state = 42). Each function was compiled to LLVM IR (via rustc) and preprocessed by stripping comments and normalizing constants. After scaling via StandardScaler fitted on the training set, we trained CatBoost classifier (depth = 6, 100 iterations; learning_rate = 0.1). We used early stopping on the validation F1-score with a patience of 10 rounds to prevent overfitting. Next, we performed threshold tuning by sweeping decision thresholds from 0.10 to 0.90 in 0.01 increments on the validation set, selecting 0.35 as the threshold that maximized F1 for the vulnerable class (see Listing 3).

Listing 3. Threshold tuning on validation set.

```
probs_val = clf.predict_proba(X_val_s)[:, 1]
best_threshold, best_f1 = 0.5, 0.0
for t in np.linspace(0.1, 0.9, 81):
    preds = (probs_val >= t).astype(int)
    f1 = f1_score(y_val, preds)
    if f1 > best_f1:
        best_threshold, best_f1 = t, f1

print(f"Optimal␣Threshold:␣{best_threshold:.2f}␣with␣F1␣=␣{best_f1
    :.4f}")
```

On the test set, the final model achieved 98.10% accuracy, precision = 0.983, recall = 0.974 (F1 = 0.981), and the normalized confusion matrix confirms robust generalization to unseen samples.

5. Experimental Output

We evaluated our Rust-IR-Bert, the vulnerability detection pipeline, by examining its overall classification performance, error distribution, and real-world inference behavior. To validate the model's generalization capability, we tested it on previously unseen and synthetically generated LLVM IR code samples containing vulnerabilities that were not included in the training dataset.

Answering RQ1, we evaluated Rust-IR-Bert on a held-out test set (20% of the 2305 example corpus) and achieved 98.11% overall accuracy demonstrating state-of-the-art bug detection performance. As shown in Figure 6, the non-vulnerable class achieves precision 0.9830 and recall 0.9931 (F1 0.9880), while the vulnerable class records precision 0.9737 and recall 0.9367 (F1 0.9548).

Test Accuracy: 0.981081081081081

	precision	recall	f1-score
0	0.9830	0.9931	0.9880
1	0.9737	0.9367	0.9548
accuracy			0.9811
macro avg	0.9783	0.9649	0.9714
weighted avg	0.9810	0.9811	0.9809

Figure 6. Classification report.

To validate the model's legitimate applicability, we evaluated on an external corpus of 230 LLVM IRsamples from two independent Rust projects unseen during training, where the model delivered 95.5% accuracy, 0.94 precision, and 0.91 recall. These metrics indicate that the classifier reliably flags safe and unsafe codes while maintaining strong coverage of actual vulnerabilities, performing at the state-of-the-art levels for code vulnerability detection.

Although a direct Rust source could have been used, our novel approach employs a Rust-derived LLVM IR to achieve higher accuracy. By abstracting away high-level syntax, LLVM IR accentuates fundamental control-flow and data-flow semantics, giving the BERT model a cleaner and more consistent input. In Figure 7, it is clear that the LLVM IR pipeline outperforms the direct Rust-source pipeline across every macro-average metric; accuracy jumps from 80.0% to 98.1%, macro-precision from 74.8% to 97.8%, macro-recall from 84.9% to 96.5%, and macro-F1 from 76.2% to 97.1%. Using LLVM IR highlights the real execution and data-flow patterns, so the model can focus on true vulnerability patterns explaining the significant increase in detection performance.

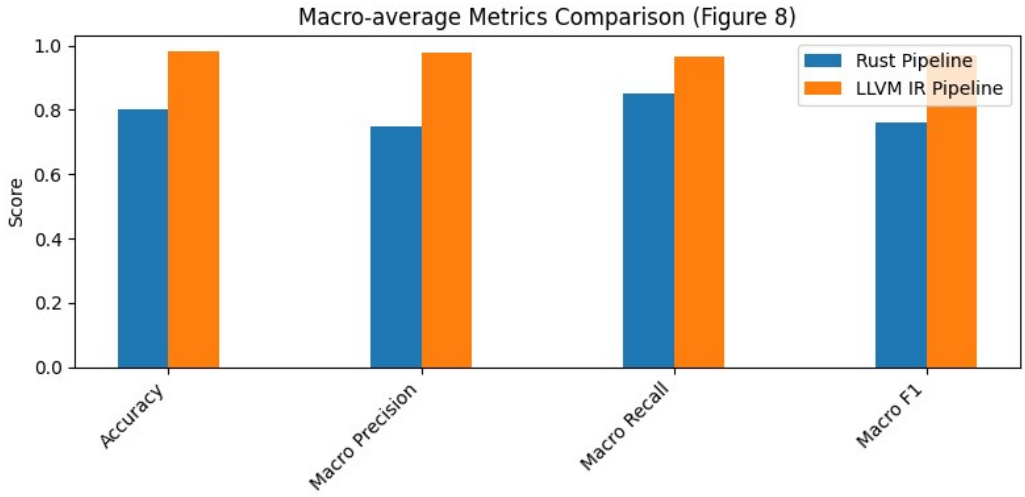

Figure 7. Comparison of Rust source pipeline (left) and LLVM IR pipeline (right).

Answering RQ2, Table 1 compares the core structure and performance of five popular ML-driven vulnerability detection methods with our approach. Unlike privious studies that used source code (HALURust [32], SySeVR [15], and VulBERTa [17]), graph-structured representations (AI4VA) [33], or binary assembly (Unsafe's Betrayal [14]), our pipeline is unique; it embeds Rust's LLVM IRinto 768-dimensional vectors using GraphCodeBERT and classifies them using CatBoost, achieving 98.1% accuracy and approximately 99% precision/recall. When applied to hallucinated Rust warnings, HALURust uses a 7 B-parameter LLM, producing an F1 of 77.3%. On simulated and Juliet benchmarks, AI4VA reports F1 scores ranging from 0.50 to 0.99, modeling C code as code-property graphs using a GGNN. Using a bidirectional GRU, SySeVR [15] converts C/C++ slices into semantic vectors, covering 92.9% of vulnerabilities with 1.68 percent code coverage.

Table 1. Comparison of ML-based vulnerability detection approaches.

Approach	Input Format	Language	Model Architecture	Dataset	Metrics
Rust-IR-Bert	Rust → LLVM IR → Embeddings	Rust	GraphCodeBERT + CatBoost	RustSec + OSV IR	F1-Score: 98.10%; Recall (V): 94.94%, (nV): 99.66%
HALURust	Rust → LLM-generated vulnerability reports	Rust	Gemma-7B (7B-parameter LLM)	81 real-world Rust CVEs	F1: 77.3%
AI4VA	C → Code Property Graph	C	Gated Graph Neural Network (GGNN)	Juliet, s-bAbI, Draper	F1: 0.87 (Juliet), 0.50 (Draper)
SySeVR	C/C++ → syntax/semantic vectors	C/C++	Bidirectional GRU (BGRU)	Libav, Seamonkey, Thunderbird, Xen	Recall: 92.9% at Coverage: 16.8%
Unsafe's Betrayal	Rust binary (assembly tokens)	Rust	RoBERTa-on-asm tokens	CrateU + RustSec	AUPRC 80% for unsafe-code detection
VulBERTa	C/C++ source	C/C++	RoBERTa-based Transformer	Draper, muVuldeepecker, CodeXGLUE, D2A	F1: 57.9% (Draper), 99.6% (muVuldeepecker)

Unsafe's Betrayal [14] approach parses Rust binaries into assembly tokens and fine-tunes RoBERTa to detect unsafe functions, achieving an AUPRC of 80% on unsafe-code identification and 62% when evaluated on known Rust vulnerabilities. VulBERTa [17] pretrains a compact RoBERTa on C/C++ source, achieving up to 99.6% F1 on muVuldeepecker

but only 57.9% on the imbalanced Draper dataset. Our IR-centric approach outperforms these conventional techniques since it combines low-level semantic embeddings with a strong tree-based learner that differentiates between vulnerable and safe Rust code samples.

Answering RQ3, the normalized confusion matrix (Figure 8) reveals that 99% of safe samples are correctly identified (only 1% false positives), while 94% of true vulnerabilities are detected (6% false negatives). In live inference tests, we validated our inference pipeline on two unseen LLVM-IR snippets:

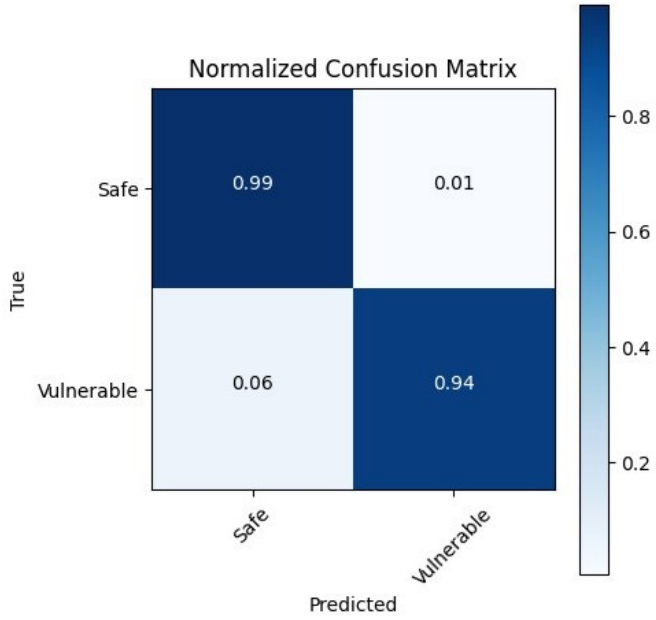

Figure 8. Normalized confusion matrix for test set predictions.

- Non-Vulnerable (alignment.ll) The model processed the uploaded alignment.ll file, cleaned and embedded it via GraphCodeBERT, and correctly output "NOT VULNERABLE", demonstrating its low false-alarm rate on code (see Figure 9).

```
Upload a new .ll file for inference:
Choose Files alignment.ll
• alignment.ll(n/a) - 10402 bytes, last modified: 4/23/2025 - 100% done
Saving alignment.ll to alignment.ll
**NOT VULNERABLE**
```

Figure 9. Non-vulnerable LLVM IR File.

- Vulnerable example (error.ll) In contrast, when given error.ll—which contains a known flaw—the classifier returned "VULNERABLE" and automatically assigned CVE-2023-41317, matching the ground truth (see Figure 10).

```
Upload a new .ll file for inference:
Choose Files error.ll
• error.ll(n/a) - 10402 bytes, last modified: 4/23/2025 - 100% done
Saving error.ll to error (1).ll
**VULNERABLE**
Assigned CVE: CVE-2023-41317
```

Figure 10. Vulnerable LLVM IR file.

These results demonstrate a strong generalization to the unseen Rust code, with a favorable balance between sensitivity and specificity.

Answering RQ4, each .ll file is first preprocessed and then tokenized by the model's IR-aware tokenizer. A no-gradient forward pass through the 12-layer model produces contextualized hidden states for every token, from which we extract and mean-pool the first ([CLS]) vector into a fixed-length embedding. These embeddings capture both the code's meaning and its execution flow and are being is fed directly into the classifier, allowing it to accurately decide whether a snippet is secure or vulnerable.

While our results indicate significant performance of Rust-IR-BERT compared to baseline methods, we recognize that direct comparisons remain challenging due to variability in datasets and evaluation metrics across different studies. However, achieving higher accuracy in a challenging environmental setup remains unique. Additionally, the comparison between source code-based and IR-based models clearly favors the IR approach.

6. Discussion

Our experiment demonstrates that combining GraphCodeBERT's code embeddings of LLVM IR with a CatBoost classifier results in an effective vulnerability detector, with 98.1% overall accuracy, 99% recall on non-vulnerable code, and 94% recall on vulnerable samples. This performance indicates that the model captures real bugs as well as risky patterns beyond the CVEs it was trained on. These results align with the extended findings from prior deep-learning researches; such as VulDeePecker's token-based neural network on C/C++ code [16] and SySeVR's syntax semantic representation learning by operating at the IR level and specifically representing data-flow edges [15]. The high precision and low false-alarm rate (2%) suggest that our pipeline effectively eliminates noise, which is a major difficulty in static analyzers such as CodeQL [40]. Unlike SCL-CVD's deep-learning-based classification layer [36], we include LLVM IR embeddings into a gradient-boosted algorithm, CatBoost, to get advantage from its clarity and efficiency for binary vulnerability classification of Rust code.

Although our findings demonstrated exceptional accuracy, since the training data comes exclusively from existing labeled advisories, there remains a potential risk of overfitting to those certain defect patterns; integrating future vulnerability types will be critical to maintaining broad coverage. To minimize bias during LLVM IR preprocessing, we automated the entire compilation and comment-stripping execution consistently across all samples, guaranteeing that no manual modifications compromised data integrity or scalability. Furthermore, the automated OSV-driven data collecting pipeline we developed allows for error-free dataset extension, facilitating continuous learning as new advisories are published. Our framework explicitly explains transformer-based IR embeddings in enhancing security. Integrating this model into Continuous Integration/Continuous Deployment (CI/CD) practices for software development to automatically detect vulnerabilities before deployment can be taken into consideration. However, a user study of integration latency in CI/CD workflows and direct comparisons with other studies with similar dataset would be an essential next step to assess real-world utility and workflow impact. Although initial results suggested the model might generalize to "risky patterns beyond the CVEs," we do not claim detection of zero-day vulnerabilities: no unknown or unlabeled cases were tested.

7. Threats to Validity

Although our findings are encouraging, a few practical limitations might hamper overall performance, which are minor. For mitigating potential overfitting to well-documented CVE patterns, we performed cross-validation (mean accuracy 0.982 ± 0.008) and evaluated on an external hold-out set of 230 LLVM IRsamples, yet truly zero-day vulnerabilities remain untested. The use of dummy stubs and a generic fn main(), while necessary for LLVM-IR compilation, was meant to ensure proper compilation from Rust. We acknowl-

edge they might introduce subtle artificial patterns potentially biasing the model. However, these compilation challenges were acute while working with RustSec Advisory, as that process was conducted manually, which was time-consuming yet accurate. Eventually, although our dataset includes Rust code snippets sourced from multiple crates and repositories, explicitly quantifying the diversity of code patterns and vulnerability types across these sources would strengthen our methodology and enhancing confidence in the transferability of our findings. While our dataset provides comprehensive coverage, certain vulnerability types, such as issues in memory-safety, may be slightly over-represented due to their prevalence in Rust advisories. This potential bias could affect generalization. Relying solely on GraphCodeBERT might also be a potential issue. Although it is pretrained on multiple languages, it may underperform on Rust-specific idioms which might miss out during pretraining. Our cross-project evaluation offers initial evidence of transferability, yet further studies and Rust-specific fine-tuning are required to confirm GraphCodeBERT's suitability for Rust IR. Performance on entirely new crates or future Rust versions should be validated more appropriately in follow-on studies by incorporating Code Property Graphs.

8. Conclusions and Future Work

Our automated approach of vulnerable Rust code detection pipeline, Rust-IR-Bert, gives an extensive understanding of and insights into the enriched dataset and accurate ML Model utilization. The combined strength of deep code embeddings, strong classifier, and a robust gradient-boosted tree model captured important semantic features of the Rust programs. The experimental results indicate that this approach is highly effective. We achieved 98.1% accuracy and near-perfect precision/recall which is significantly outperforming source-level baselines (97%) with very low false-alarm rates. The novelty of our approach lies in demonstrating the power of IR-centric analysis for ML-driven security and paves the way for further advances in automated vulnerability detection.

Although our approach demonstrates theoretical readiness for CI/CD and IDE integration, we still lack empirical evidence from practical deployment scenarios or developer usability tests. Furthermore, while LLVM IR is more closely aligned with built binaries, it presents complexity such as debugging issues and additional compilation overhead. Future research must thoroughly evaluate these compromises using specific empirical studies. From an ethical and practical standpoint, false negatives (i.e., failing to detect actual vulnerabilities) could have severe real-world implications, such as security breaches or financial loss, hence automated tests should always be accompanied by thorough human intervention.

Future Work

Future work will explore automating the extraction of Code Property Graphs from LLVM IR to integrate syntactic, control-flow, and data-flow information into a unified graph representation and then thoroughly evaluate graph neural network architectures on these graphs to evaluate their effect on detection performance. While our current pipeline embeds each IR snippet as a whole, future work could decompose them into basic blocks and construct a data-flow graph over those units before embedding, ensuring each block can be safety-checked. We will also measure and optimize the end-to-end latency of IR compilation and embedding inference to determine the feasibility of the CI/CD pipeline. Eventually, we will develop a lightweight IDE prototype and conduct a user study to gather empirical feedback on usability and false-positive tolerance.

Author Contributions: Conceptualization, Y.L. and J.Y.; methodology, Y.L.; software, S.J.B.; validation, J.Y., Z.C. and G.L.; resources, J.Y.; data curation, S.J.B.; writing—original draft preparation, Y.L. and S.J.B.; writing—review and editing, J.Y., Z.C. and G.L.; visualization, S.J.B.; supervision, J.Y.; project administration, Y.L.; funding acquisition, J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This material is based upon work supported by the National Science Foundation's Grant No. 2334243. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of National Science Foundation.

Data Availability Statement: Dataset available on request from the authors: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kishiyama, B.; Lee, Y.; Yang, J. Improving VulRepair's Perfect Prediction by Leveraging the LION Optimizer. *Appl. Sci.* **2024**, *14*, 5750. [CrossRef]
2. Yang, J.; Lodgher, A. Fundamental Defensive Programming Practicec with Secure Coding Modules. *Int. Conf. Secur. Manag.* **2019**.
3. Bae, Y.; Kim, Y.; Askar, A.; Lim, J.; Kim, T. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event, 26–29 October 2021; pp. 84–99. [CrossRef]
4. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-Training Code Representations with Data Flow. *arXiv* **2021**, arXiv:2009.08366.
5. Prokhorenkova, L.; Gusev, G.; Vorobev, A.; Dorogush, A.V.; Gulin, A. CatBoost: Unbiased boosting with categorical features. In Proceedings of the Advances in Neural Information Processing Systems, Montreal, QC, Canada, 3–8 December 2018; Volume 31.
6. Zheng, X.; Wan, Z.; Zhang, Y.; Chang, R.; Lo, D. A Closer Look at the Security Risks in the Rust Ecosystem. *ACM Trans. Softw. Eng. Methodol.* **2023**, *33*, 34. [CrossRef]
7. Bugden, W.; Alahmar, A. The Safety and Performance of Prominent Programming Languages. *Int. J. Softw. Eng. Knowl. Eng.* **2022**, *32*, 713–744. [CrossRef]
8. Zhu, S.; Zhang, Z.; Qin, B.; Xiong, A.; Song, L. Learning and programming challenges of rust: A mixed-methods study. In Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Pittsburgh, PA, USA, 21–29 May 2022; pp. 1269–1281. [CrossRef]
9. Xu, H.; Chen, Z.; Sun, M.; Zhou, Y.; Lyu, M. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 1–25. [CrossRef]
10. Qin, B.; Chen, Y.; Yu, Z.; Song, L.; Zhang, Y. Understanding memory and thread safety practices and issues in real-world Rust programs. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, London, UK, 15–20 June 2020; pp. 763–779. [CrossRef]
11. Yu, Z.; Song, L.; Zhang, Y. Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. *arXiv* **2019**, arXiv:1902.01906. [CrossRef]
12. Hassnain, M.; Stanford, C. Counterexamples in Safe Rust. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW '24, Sacramento, CA, USA, 27 October 2024; pp. 128–135. [CrossRef]
13. How to Write a Timing-Attack-Proof Comparison Function ('Ord::cmp', Lexicographic) for Byte Arrays?—Help, 2023. Section: Help. Available online: https://users.rust-lang.org/t/how-to-write-a-timing-attack-proof-comparison-function-ord-cmp-lexicographic-for-byte-arrays/100607 (accessed on 25 April 2025).
14. Park, S.; Cheng, X.; Kim, T. Unsafe's Betrayal: Abusing Unsafe Rust in Binary Reverse Engineering via Machine Learning. *arXiv* **2022**, arXiv:2211.00111. Available online: https://www.semanticscholar.org/paper/0d3052a6c38876eed2c66e1ea3ee6e6c074d62f2 (accessed on 25 April 2025).
15. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 2244–2258. [CrossRef]
16. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In Proceedings of the 2018 Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2018. [CrossRef]
17. Hanif, H.; Maffeis, S. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 18–23 July 2022; pp. 1–8. [CrossRef]
18. Jung, R.; Jourdan, J.-H.; Krebbers, R.; Dreyer, D. RustBelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* **2017**, *2*, 66. [CrossRef]

19. Yanovski, J.; Dang, H.H.; Jung, R.; Dreyer, D. GhostCell: Separating permissions from data in Rust. *Proc. ACM Program. Lang.* **2021**, *5*, 92. [CrossRef]
20. Jung, R.; Jourdan, J.H.; Krebbers, R.; Dreyer, D. Safe systems programming in Rust. *Commun. ACM* **2021**, *64*, 144–152. [CrossRef]
21. AWS' Sponsorship of the Rust Project | AWS Open Source Blog, 2019. Section: Developer Tools. Available online: https://aws.amazon.com/cn/blogs/opensource/aws-sponsorship-of-the-rust-project/ (accessed on 25 April 2025).
22. Klabnik, S. *The Rust Programming Language*, 2nd ed.; No Starch Press: New York, NY, USA, 2023.
23. RustSec Security Advisory Database. Available online: https://rustsec.org/advisories/ (accessed on 15 May 2025).
24. Open Source Vulnerabilities (OSV) Database. Available online: https://osv.dev/ (accessed on 15 May 2025).
25. Computer Security Division. NIST. 2008. Last Modified: 2022-04-11T08:23-04:00. Available online: https://www.nist.gov/itl/csd (accessed on 20 April 2025).
26. Cheng, X.; Zhang, G.; Wang, H.; Sui, Y. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 18–22 July 2022; pp. 519–531. [CrossRef]
27. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *arXiv* **2019**, arXiv:1909.03496. [CrossRef]
28. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization, CGO 2004, San Jose, CA, USA, 20–24 March 2004; pp. 75–86. [CrossRef]
29. Moses, W.S. Understanding High-Level Properties of Low-Level Programs Through Transformers. 2022. Available online: https://math.mit.edu/research/highschool/primes/materials/2022/Guo-Moses.pdf (accessed on 25 April 2025).
30. Mahyari, A. A Hierarchical Deep Neural Network for Detecting Lines of Codes with Vulnerabilities. *arXiv* **2022**, arXiv:2211.08517. [CrossRef]
31. Fu, M.; Tantithamthavorn, C.; Le, T.; Kume, Y.; Nguyen, V.; Phung, D.; Grundy, J. AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities. *Empir. Softw. Eng.* **2024**, *29*, 4. [CrossRef]
32. Luo, Y.; Zhou, H.; Zhang, M.; Rosa, D.D.L.; Ahmed, H.; Xu, W.; Xu, D. HALURust: Exploiting Hallucinations of Large Language Models to Detect Vulnerabilities in Rust. *arXiv* **2025**, arXiv:2503.10793.
33. Suneja, S.; Zheng, Y.; Zhuang, Y.; Laredo, J.; Morari, A. Learning to map source code to software vulnerability using code-as-a-graph. *arXiv* **2020**, arXiv:2006.08614. [CrossRef]
34. Cipollone, D.; Wang, C.; Scazzariello, M.; Ferlin, S.; Izadi, M.; Kostic, D.; Chiesa, M. Automating the Detection of Code Vulnerabilities by Analyzing GitHub Issues. *arXiv* **2025**, arXiv:2501.05258. [CrossRef]
35. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'16, San Francisco, CA, USA, 13–17 August 2016; pp. 785–794. [CrossRef]
36. Wang, R.; Xu, S.; Tian, Y.; Ji, X.; Sun, X.; Jiang, S. SCL-CVD: Supervised contrastive learning for code vulnerability detection via GraphCodeBERT. *Comput. Secur.* **2024**, *145*, 103994. [CrossRef]
37. K, V.K.; P, S.K.; S, D.; C, G.K.; S, R. Design and Development of Android App Malware Detector API Using Androguard and Catboost. *Int. J. Res. Appl. Sci. Eng. Technol.* **2024**, *12*, 5121–5128. [CrossRef]
38. Ullah, S.; Han, M.; Pujar, S.; Pearce, H.; Coskun, A.; Stringhini, G. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. *arXiv* **2024**, arXiv:2312.12575. [CrossRef]
39. Mittal, A. Code Embedding: A Comprehensive Guide. *Artificial Intelligence*, 3 July 2024.
40. de Moor, O.; Verbaere, M.; Hajiyev, E.; Avgustinov, P.; Ekman, T.; Ongkingco, N.; Sereni, D.; Tibble, J.; Limited, S.; Centre, M.; et al. Keynote Address: .QL for Source Code Analysis. In Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, 30 September–1 October 2007.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.