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

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

Posted Date: 10 June 2025

doi: 10.20944/preprints202506.0788.v1

Keywords: Rust; LLVM IR; Vulnerability Detection; Code Embedding; GraphCodeBERT, Machine Learning

 doi:10.20944/preprints202506.0788.v1

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; sjannatulbo@tamusa.edu (S.J.B.); jyang@tamusa.edu (J.Y.); zcao@tamusa.edu (Z.C.); gling@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. Using LLVM IR provides a language-neutral, semantically rich view of the program, capturing data and control flow, and reducing the noise of high-level syntax differences. Our method leverages a transformer model, GraphCodeBERT, to embed the IR and CatBoost classifier to classify code as vulnerable or safe. When evaluated on a mix of known buggy and safe code, this method obtained 98.11% overall accuracy, with a recall of 99.31% for safe code and 93.67% for vulnerable code. Our evaluation utilizes a diverse dataset of over 2,300 CVE-linked and Rust snippets compiled to LLVM IR, facilitating wide-range of coverage across real-world crates.

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

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. From low-level testing to attack simulation, existing methods struggle to scale across large codebases without generating excessive false positives. This paper addresses that gap by introducing a transformer-based machine learning approach - **Rust-IR-Bert** which operates directly on LLVM IR - to improve accuracy in detecting false positives in vulnerable Rust source code detection. Rust is considered to be the safest system-level programming language of this time [3]. Bugden and Alahmar have conducted an analysis among leading programming languages to evaluate safety and performance, and found that Rust outperforms the other languages [4]. 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 [5,6].

Rudra [7] recently revealed that by employing unsafe blocks in error-prone spots can identify a lot of memory-safety bugs in the Rust ecosystem. 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

 doi:10.20944/preprints202506.0788.v1

employ unsafe code and the common misuse patterns that trigger safety breaches [8]. Regarding 'Concurrency Bugs', Rust enforces strict rules to prevent data races, but complex multi-threaded programs can still have logical concurrency bugs that aren't caught by the compiler [9].

Despite Rust's safety guarantees, even within 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 [3,10]. 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 done before publishing the software.

Rust initially lacks a mechanism against timing attacks, which could lead to iteration time varying based on the data [11]. 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 [12].

```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. Existing ML-based approaches [13–15] often focus on source code or abstract representations; in contrast, our work directly utilizes Rust's LLVM Intermediate Representation (IR) as input. Our novel pipeline encodes LLVM IR using a transformer-based code model, GraphCodeBERT [16] to capture rich code semantics, then classifies vulnerabilities with a CatBoost [17] classifier and optimizes predictions via threshold tuning. 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. Experimental results in **Table 1** demonstrate that our updated approach delivers significantly higher accuracy and F1-scores than prior baselines.

2. Background

This section provides some background of Rust, its safety and unsafe mechanisms, language-independent intermediate representation (IR), Vulnerability Databases and Embeddings.

 doi:10.20944/preprints202506.0788.v1

2.1. Rust's Safety Mechanism

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. Microsoft 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].

2.2. Unsafe Rust

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 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 less 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 [6].

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 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.

 doi:10.20944/preprints202506.0788.v1


```rust
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.

2.3. RustSec and OSV Vulnerability Databases

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, GitHub 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 RustSec Advisory and OSV in vulnerability predictive modeling, proving that information from these databases can anticipate which software components are most likely at risk [3].

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 [14] have used CVE-labeled code devices to train BLSTM models for vulnerability detection.

2.4. Language-Independent Intermediate Representation (LLVM IR)

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

 doi:10.20944/preprints202506.0788.v1

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 has 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].

2.5. Embedding-Based Vulnerability Classifier

Our embedding-based classifier employs CatBoost [31]. Pre-trained, 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 pre-trained 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 pre- training [16].

2.6. 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 complies Rust code to intermediate representation which turns the source code into a detailed version. Also it doesn't require large-scale self-replication and distribution, but only needs to be placed the source - .ll or .rs file into the ML pipeline to determine the safety. A well-known open source tool AIBughunter [32], 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. Besides, 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 [33]. 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. [12]. VulBERTa pre-trains RoBERTa on millions of lines of C/C++ source with their custom code-aware tokenizer, then fine-tunes for vulnerability classification, which achieves 99.6% F1 on the muVulDeePecker benchmark [14]. 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 [34].

Our model uses LLVM IR, which exclusively captures low-level behavior, improving semantic understanding. Cipollone [35] introduced a transformer-based framework that classifies CVE linked GitHub issues using embedding models and XGBoost [36], demonstrating that natural-language signals can provide early vulnerability alerts. In contrast, our ML-driven pipeline employs CatBoost as classifer. SySeVR model [13] 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

 doi:10.20944/preprints202506.0788.v1

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% [37].

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 is 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 [38]. 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 presents the system architecture of our proposed detection pipeline. The pipeline begins by compiling Rust source code into LLVM IR, preserving program semantics wile providing a lower-level representation. The resulting IR text is tokenized and processed by a pretrained GraphCodeBERT model, 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 [31]. 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 separate validation dataset by enhancing 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 (typically between 0.3 and 0.5) for final predictions. 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.

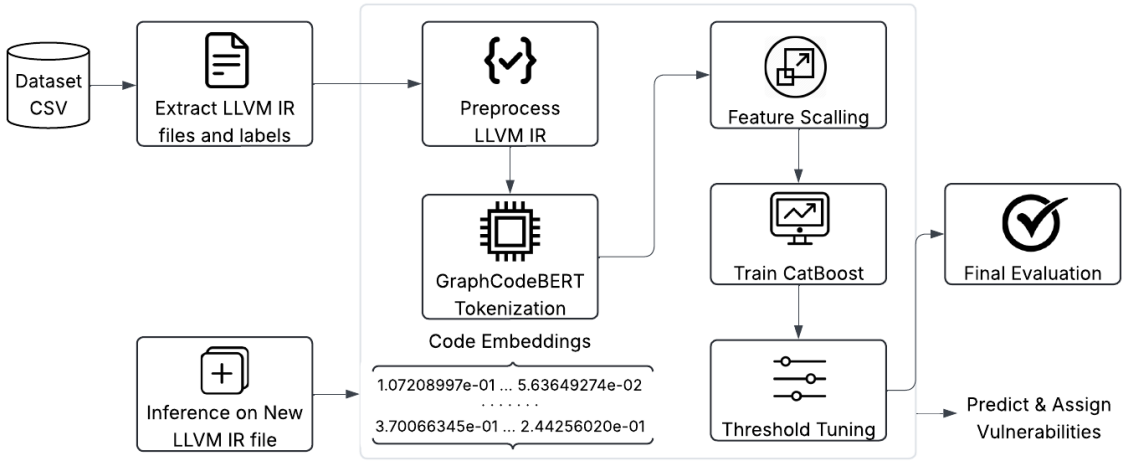

Figure 3. ML Pipeline.

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

 doi:10.20944/preprints202506.0788.v1

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 Pre-processing

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 (OSV) 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 2,300 labeled .rs/.ll pairs for downstream embedding and classification tasks, ensuring comprehensive coverage across the Rust ecosystem. Figure 4 illustrates this data collection pipeline.

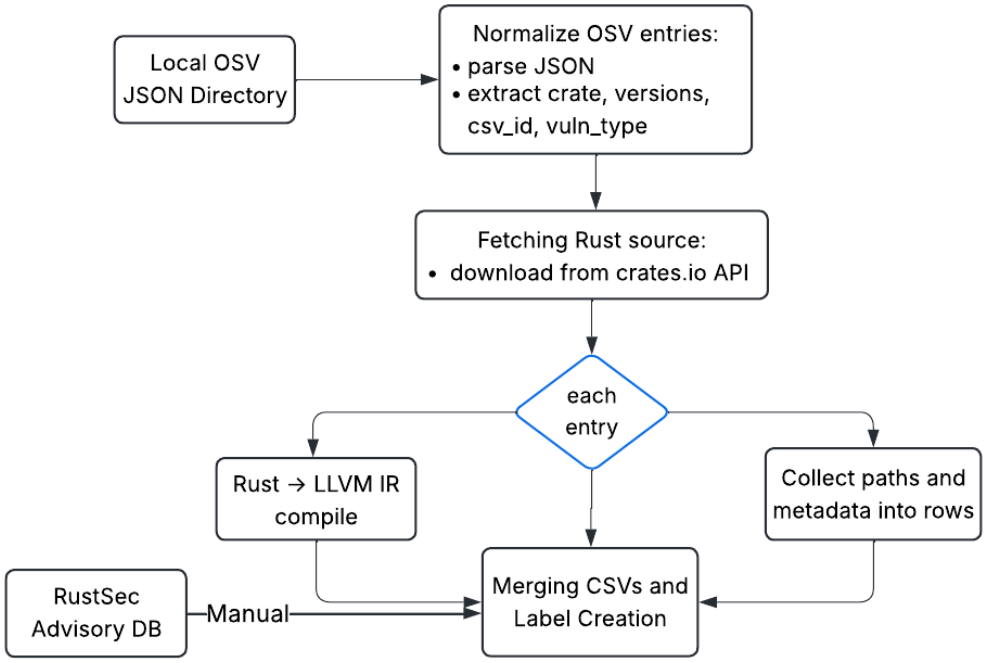

Figure 4. Data Collection Pipeline.

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

 doi:10.20944/preprints202506.0788.v1

paths, and advisory descriptions as of April 25, 2025 enabling scalable, repeatable updates as new advisories appear.

The bar chart from Figure 5 presents 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, 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.

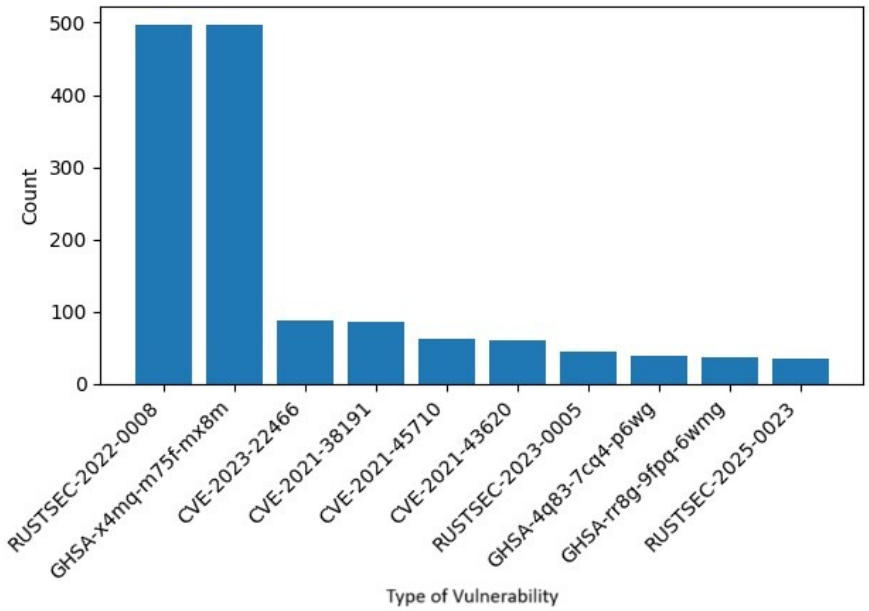

Figure 5. Bar Chart of Top 10 Types of Vulnerabilities Detected by the Model.

4.3. Rust Snippet Wrapping

To prepare each Rust snippet for LLVM-IR emission, we automatically embed it within a minimal scaffolding layer. This involves prepending dummy stubs for commonly missing modules and, injecting a fn main() entry point when necessary.

Key Integration Challenges:

- **Missing `main()` Functions:** Many examples consist solely of library functions. Our wrapper detects any code lacking main function and appends a minimal fn main() so that rustc –emit=llvm-ir will 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 only self-contained snippets proceed.

A dedicated pipeline was developed to automate the handling of these issues, adding for missing main, injecting the necessary boilerplate, writing the augmented .rs file into local folder and then invoking rustc to produce the corresponding .ll file - all without manual intervention.

 doi:10.20944/preprints202506.0788.v1

4.4. 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.

4.5. Code Pre-processing 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. For which we added a function to remove unnecessary comments and blannk/white spaces from the LLVM IR code [39]. 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 light-weight 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.

```
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(';')
            )
```

Listing 1: Pre-processing .ll Files.

4.6. Embedding-Based Feature Extraction

Embeddings translate complex code structures into fixed-dimensional numerical vectors that preserves the meaning and functionality of the code. 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 [40].

```
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)
```

Listing 2: Extraction of CLS Embedding from LLVM IR.

Tokenization and Data -Flow Construction- Extracting GraphCodeBERT Embeddings:

GraphCodeBERT is a pre-trained Transformer model specially designed to capture both data-flow and control-flow dependencies in source code [16]. 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

 doi:10.20944/preprints202506.0788.v1

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).

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.

4.7. 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 2,305 Rust functions (769 vulnerable, 1,536 safe) drawn from CVE-linked and benign code. Splitting 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).

```
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}")
```

Listing 3: Threshold Tuning on Validation Set.

On the test set, the final model achieved 98.10% accuracy, precision = 0.983, recall = 0.974 (F1 = 0.981) (Figure 6) and the normalized confusion matrix confirms robust generalization to unseen samples as shown in Figure 7.

Test Accuracy: 0.9810810810810811

	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.

 doi:10.20944/preprints202506.0788.v1

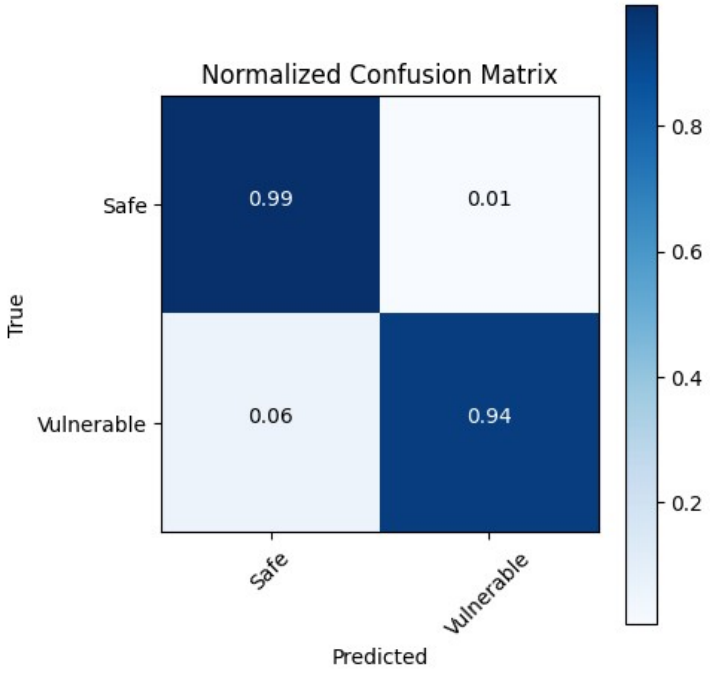

Figure 7. Normalized Confusion Matrix for Test Set Predictions.

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, our combined pre-trained BERT model + supervised algorithms-based classifier pipeline achieved 98.11% overall accuracy on the processed set of 2,305 LLVM-IR samples, 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). 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 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 8, 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.

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 [33], SySeVR [13], VulBERTa [15]), graph-structured representations (AI4VA) [41], or binary assembly (Unsafe's Betrayal [12]), our pipeline is unique; embeds Rust's LLVM-IR into 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 [13] converts C/C++ slices into semantic vectors, covering 92.9% of vulnerabilities with 1.68 percent code coverage.

 doi:10.20944/preprints202506.0788.v1

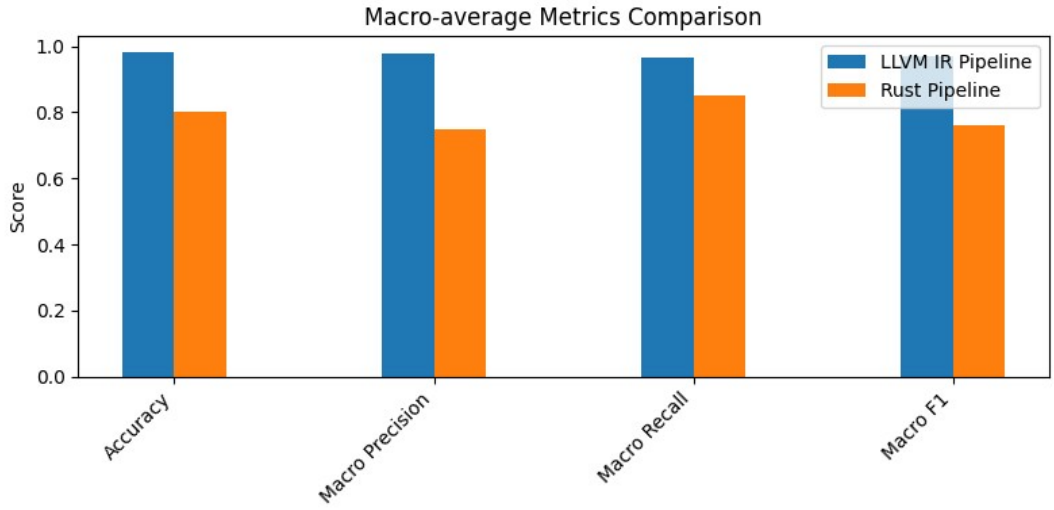

Figure 8. LLVM IR vs Rust in the Same Pipeline.

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 [12] 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 [15] pre-trains 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 differentiates between vulnerable and safe Rust code samples.

Answering RQ3, the normalized confusion matrix (Figure 7), 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:

- **Safe example (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 benign code (see Figure 7).

 doi:10.20944/preprints202506.0788.v1

- **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 8).

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

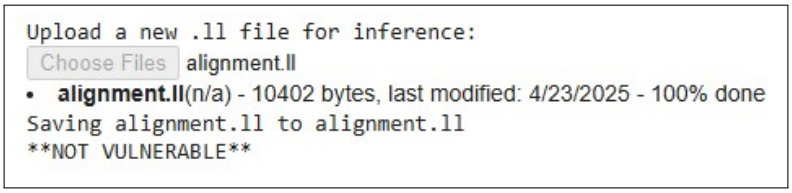

Figure 9. Non-Vulnerable LLVM IR File.

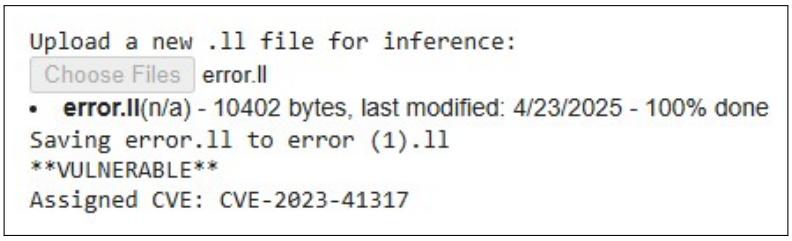

Figure 10. Vulnerable LLVM IR File.

Answering RQ4, each .ll file is first pre-processed, then tokenized by pre-trained 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.

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 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 [14] and SySeVR's syntax semantic representation learning by operating at the IR level and specifically representing data-flow edges [13]. 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 [42]. Unlike SCL-CVD's deep-learning-based classification layer [37], 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 error-free dataset extension, facilitating continuous learning as new advisories are published. Our framework explains transformer-based code embeddings enhance security and can be seamlessly integrated into Continuous Integration/Continuous Deployment practices of software development to automatically detect vulnerabilities before deployment.

 doi:10.20944/preprints202506.0788.v1

7. Threats to Validity

Although our findings are encouraging, a few practical limitations might hamper overall performance, which are minor. Our dataset draws exclusively on OSV-registered advisories, so previously unreported or zero-day vulnerabilities are not represented. The use of dummy stubs and a generic fn main() while necessary for LLVM-IR compilation, was meant to ensure proper compilation from Rust. However, these compilation challenges were acute while working with RustSec Advisory, as that process was done manually, which was time-consuming yet accurate.

Relying solely on GraphCodeBERT might also be a potential issue. Although it is pre-trained on multiple languages, it may underperform on Rust-specific idioms which might miss out during pretraining. 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 and insights of 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 leveraging IR's uniform control-flow and data-flow semantics, which provides cleaner, language-neutral input for the model, and in combining transformer-based embeddings with a robust tree-based classifier. The extensive evaluation on over 2,300 CVE-linked and manual snippets alos demonstrates strong generalization to unseen vulnerabilities and real-world Rust code.

Beyond the quantitative results, Rust-IR-Bert's broader impact lies in its practical significance for secure software development. Our pipeline can be integrated into CI/CD pipelines or IDE toolchains to continuously scan Rust code for vulnerabilities prior to deployment. It can even suggest an associated CVE ID or vulnerability type if an issue arises. By integrating Rust-IR-Bert into development workflows, developers can automatically flag subtle bugs beyond compile-time checks, augmenting Rust's inherent safety as it has very low false-alarm alarms. This work demonstrates the power of IR-centric analysis for ML-driven security and paves the way for further advances in automated vulnerability detection.

8.1. Future Work

Future work will explore constructing Code Property Graphs (CPGs) directly from LLVM IR to capture richer program semantics combining AST, data-flow and control-flow into a unified representation. By integrating these CPG features with existing GraphCodeBERT embeddings and experimenting with graph neural networks, we aim to further enhance vulnerability detection accuracy and granularity.

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.

References

1. Kishiyama, B.; Lee, Y.; Yang, J. Improving VulRepair's Perfect Prediction by Leveraging the LION Optimizer. *Applied Sciences* **2024**, *14*, 5750. Number: 13 Publisher: Multidisciplinary Digital Publishing Institute, https://doi.org/10.3390/app14135750.
2. Yang, J.; Lodgher, A. Fundamental Defensive Programming Practicec with Secure Coding Modules. *International Conference on Security and Management* **2019**.
3. 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*, 34:1–34:30. https://doi.org/10.1145/3624738.

 doi:10.20944/preprints202506.0788.v1

4. Bugden, W.; Alahmar, A. The Safety and Performance of Prominent Programming Languages. *International Journal of Software Engineering and Knowledge Engineering* **2022**, *32*, 713–744. https://doi.org/10.1142/S0218194022500231.
5. Zhu, S.; Zhang, Z.; Qin, B.; Xiong, A.; Song, L. Learning and programming challenges of rust: a mixed-methods study. In Proceedings of the Proceedings of the 44th International Conference on Software Engineering, New York, NY, USA, 2022; ICSE '22, pp. 1269–1281. https://doi.org/10.1145/3510003.3510164.
6. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs | ACM Transactions on Software Engineering and Methodology.
7. 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 Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event Germany, 2021; pp. 84–99. https://doi.org/10.1145/3477132.3483570.
8. 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 Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, London UK, 2020; pp. 763–779. https://doi.org/10.1145/3385412.3386036.
9. Zeming Yu.; Linhai Song.; Yiying Zhang. Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software. *ArXiv* **2019**, *abs/1902.01906*.
10. Hassnain, M.; Stanford, C. Counterexamples in Safe Rust. In Proceedings of the Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, New York, NY, USA, 2024; ASEW '24, pp. 128–135. https://doi.org/10.1145/3691621.3694943.
11. How to write a timing-attack-proof comparison function ('Ord::cmp', lexicographic) for byte arrays? - help, 2023. Section: help.
12. Park, S.; Cheng, X.; Kim, T. Unsafe's Betrayal: Abusing Unsafe Rust in Binary Reverse Engineering via Machine Learning. 2022.
13. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* **2022**, *19*, 2244–2258. arXiv:1807.06756 [cs], https://doi.org/10.1109/TDSC.2021.3051525.
14. 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 Proceedings 2018 Network and Distributed System Security Symposium, 2018. arXiv:1801.01681 [cs], https://doi.org/10.14722/ndss.2018.23158.
15. 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). IEEE, 2022, pp. 1–8. Place: Padua, Italy tex.eventtitle: 2022 International Joint Conference on Neural Networks (IJCNN), https://doi.org/10.1109/IJCNN55064.2022.9892280.
16. 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, 2021. arXiv:2009.08366 [cs], https://doi.org/10.48550/arXiv.2009.08366.
17. 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. Curran Associates, Inc., 2018, Vol. 31.
18. Ralf Jung.; Jacques-Henri Jourdan.; R. Krebbers.; Derek Dreyer. RustBelt: securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages* **2017**, *2*, 1–34. https://doi.org/10.1145/3158154.
19. Yanovski, J.; Dang, H.H.; Jung, R.; Dreyer, D. GhostCell: separating permissions from data in Rust. *Proceedings of the ACM on Programming Languages* **2021**, *5*, 1–30. https://doi.org/10.1145/3473597.
20. Jung, R.; Jourdan, J.H.; Krebbers, R.; Dreyer, D. Safe systems programming in Rust. *Communications of the ACM* **2021**, *64*, 144–152. https://doi.org/10.1145/3418295.
21. AWS' sponsorship of the Rust project | AWS Open Source Blog, 2019. Section: Developer Tools.
22. Klabnik, S. *The Rust Programming Language, 2nd Edition*; No Starch Press: New York, 2023.
23. About RustSec › RustSec Advisory Database.
24. OSV - Open Source Vulnerabilities.
25. Computer Security Division. *NIST* **2008**. Last Modified: 2022-04-11T08:23-04:00.
26. Cheng, X.; Zhang, G.; Wang, H.; Sui, Y. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In Proceedings of the Proceedings of the 31st ACM SIGSOFT International

 doi:10.20944/preprints202506.0788.v1

Symposium on Software Testing and Analysis, Virtual South Korea, 2022; pp. 519–531. https://doi.org/10.1145/3533767.3534371.
27. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks, 2019. arXiv:1909.03496 [cs], https://doi.org/10.48550/arXiv.1909.03496.
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, 2004. CGO 2004., San Jose, CA, USA, 2004; pp. 75–86. https://doi.org/10.1109/CGO.2004.1281665.
29. Moses, W.S. Understanding High-Level Properties of Low-Level Programs Through Transformers. 2022.
30. Mahyari, A. A Hierarchical Deep Neural Network for Detecting Lines of Codes with Vulnerabilities, 2022. arXiv:2211.08517 [cs], https://doi.org/10.48550/arXiv.2211.08517.
31. Prokhorenkova, L.; Gusev, G.; Vorobev, A.; Dorogush, A.V.; Gulin, A. CatBoost: unbiased boosting with categorical features, 2019. arXiv:1706.09516 [cs], https://doi.org/10.48550/arXiv.1706.09516.
32. 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. *Empirical Software Engineering* **2024**, *29*, 4. https://doi.org/10.1007/s10664-023-10346-3.
33. HALURust: Exploiting Hallucinations of Large Language Models to Detect Vulnerabilities in Rust.
34. 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**, *abs/2006.08614*.
35. Cipollone, D.; Wang, C.; Scazzariello, M.; Ferlin, S.; Izadi, M.; Kostic, D.; Chiesa, M. Automating the Detection of Code Vulnerabilities by Analyzing GitHub Issues, 2025. arXiv:2501.05258 [cs], https://doi.org/10.48550/arXiv.2501.05258.
36. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In Proceedings of the Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 2016; KDD '16, pp. 785–794. https://doi.org/10.1145/2939672.2939785.
37. Wang, R.; Xu, S.; Tian, Y.; Ji, X.; Sun, X.; Jiang, S. SCL-CVD: Supervised contrastive learning for code vulnerability detection via GraphCodeBERT. *Computers & Security* **2024**, *145*, 103994. https://doi.org/10.1016/j.cose.2024.103994.
38. 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. *International Journal for Research in Applied Science and Engineering Technology* **2024**, *12*, 5121–5128. https://doi.org/10.22214/ijraset.2024.61156.
39. 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, 2024. arXiv:2312.12575 [cs], https://doi.org/10.48550/arXiv.2312.12575.
40. Mittal, A. Code Embedding: A Comprehensive Guide, 2024. Section: Artificial Intelligence.
41. Sahil Suneja.; Yunhui Zheng.; Yufan Zhuang.; Jim Laredo.; Alessandro Morari. Learning to map source code to software vulnerability using code-as-a-graph. *ArXiv* **2020**, *abs/2006.08614*.
42. 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.