

Embedding-Level Pretraining for Assembly-Level Vulnerability Detection Using Transformer Models

Dominik Ciesiołkiewicz, Mirosław Łazoryszczak and Piotr Dziurzański

0009-0006-1032-0034

0000-0001-7653-5982

0000-0001-9542-652X

West Pomeranian University of Technology in Szczecin,
Faculty of Computer Science and Information Technology,
ul. Żołnierska 49, 71-210 Szczecin, Poland

Email: dciesiolkiewicz, mlazoryszczak, dziurzan@zut.edu.pl

Abstract—This paper presents a deep learning framework for assembly-level vulnerability detection that operates directly on disassembled binary code, addressing scenarios where source code is unavailable. The approach combines pretrained PalmTree instruction embeddings with a RoBERTa-based transformer adapted to process continuous inputs. We introduce an embedding-level masked language modeling strategy that masks in the continuous embedding space using Gaussian noise. This enables the model to capture contextual relationships between assembly instructions without relying on discrete token representations. The method is evaluated on the ROMEO dataset across multiple Common Weakness Enumeration (CWE) classes and demonstrates strong performance on accuracy, precision, recall, F1-score and PR-AUCs. Ablation studies further confirm the effectiveness of embedding-level pretraining and PalmTree embeddings. These results suggest that embedding-aware transformer architectures are a promising direction for vulnerability detection in binary code.

Index Terms—Vulnerability detection, Embedded systems, Deep learning, Representation learning, Assembly code analysis.

I. INTRODUCTION

EMBEDDED systems are ubiquitous in the modern world, ranging from smart home devices and industrial control units to automotive systems and medical equipment [1]. These devices are designed to perform specific tasks and, each year, are increasingly connected to the network, making them attractive targets for cybercriminals.

A major problem in evaluating a single network-connected device is obtaining the source code for the embedded software. Many embedded systems use firmware or third-party libraries that are available only in compiled binary form. In some cases, such as black-box penetration testing of the device, the source code cannot be fully acquired [2]. This means that the only way for an attacker to access any code is to reverse-engineer the binaries running in the device's memory using tools such as Ghidra or BinaryNinja. Because of this, security analysts must work with assembly code, the low-level instructions that processors actually execute, to find potential vulnerabilities. Manual review of assembly code is extremely time-consuming and requires specialized expertise [3].

Traditional vulnerability detection methods (such as static, dynamic and symbolic analysis [4]) are significantly less effective when applied to assembly language. It lacks the clear structure and human-readable syntax of high-level programming languages, making it difficult for both humans and automated tools to understand what the code actually does. The semantic relationships between different parts of the code are harder to identify, and the same vulnerability might look completely different at the assembly level compared to the source code level [5].

Recent advances in artificial intelligence and machine learning have shown promise in automating security analysis tasks. Two notable approaches have emerged: VulBERTa [6], which uses deep learning to detect vulnerabilities in source code, and PalmTree [7], which specializes in understanding assembly language patterns. The former has demonstrated strong performance in finding security flaws in high-level code, while the latter excels at capturing the structure and meaning of assembly instructions.

However, the aforementioned tools typically focus on either source code analysis or basic assembly instruction processing, but not both. To the best of our knowledge, few existing approaches address vulnerability detection directly at the assembly level using deep learning, with VDGraph2Vec being one of the notable examples [8].

This creates a gap in which the strengths of source-code-trained models cannot be effectively combined with the assembly-specific knowledge required for syntax understanding. In addition, there is a shortage of large, high-quality datasets specifically designed to train machine learning models to detect vulnerabilities in assembly code.

This paper introduces a new pipeline that addresses this limitation by combining pretrained PalmTree embeddings with a RoBERTa-based transformer. It can analyze embedded system assembly code to identify potential security vulnerabilities. This system uses machine learning to understand both the low-level structure of assembly instructions and the high-level patterns that indicate security problems.

Experimental results demonstrate that the proposed approach outperforms selected baseline models while remaining

computationally efficient for practical applications in security analysis workflows.

The main contributions of this work are as follows:

- We propose an assembly-level vulnerability detection framework that operates directly on disassembled binary code, eliminating the need for source code access and enabling analysis in realistic embedded system scenarios.
- We introduce a novel embedding-level masked language modeling (MLM) strategy, in which masking is applied in the continuous embedding space using Gaussian noise. This approach allows the model to learn contextual representations of assembly instructions without relying on discrete token vocabularies.
- We design a hybrid architecture that integrates PalmTree instruction embeddings with a RoBERTa-based transformer adapted to process continuous inputs, enabling effective representation learning for low-level code.
- We provide an extensive experimental evaluation across multiple CWE (Common Weakness Enumeration) classes using the ROMEO dataset, including analysis of class imbalance effects and metric sensitivity beyond accuracy.
- We identify key limitations of current evaluation practices for assembly-level vulnerability detection, particularly the reliance on synthetic datasets, and outline directions for improving real-world applicability.

As far as we are aware, this is among the earliest methods that apply embedding-level pretraining directly to assembly-level vulnerability detection.

II. RELATED WORK

Software vulnerabilities frequently arise from outdated architectures, the reuse of vulnerable libraries, and increased exposure of interconnected embedded systems. Additionally, resource constraints limit the use of traditional security tools, enabling attackers to develop specialized malware targeting these devices. On the other hand, vulnerability detection in embedded software has advanced considerably. Conventional methods, such as static and dynamic code analysis, have been widely used but face scalability and accuracy challenges [9].

Recently, advances in artificial intelligence have introduced new possibilities by leveraging deep learning and large datasets to improve detection performance.

A. Traditional approaches

Traditional approaches to identifying vulnerabilities in embedded system code include several well-established techniques. Most methods available for binary code analysis, thoroughly discussed in [9], are summarized in Fig. 1. The following subsections provide a brief overview of the most prominent approaches.

1) *Static analysis*: Static analysis examines source code/binaries without executing the program, enabling detection of security vulnerabilities during the development process [10], [11]. This approach scans code to identify common weaknesses, such as buffer overflows, integer overflows, and similar issues. Modern static analyzers also support compliance with embedded systems coding standards [12], helping ensure the device's portability, reliability, and security throughout the development process. However, static analysis also has limitations. It struggles to detect runtime vulnerabilities or issues that only manifest under specific execution conditions [13], [14].

2) *Dynamic analysis*: Dynamic analysis evaluates applications during runtime by simulating attacks and monitoring system behavior without requiring access to source code [15]. Dynamic Application Security Testing (DAST) tools perform black-box testing, attacking applications from the outside to identify vulnerabilities such as memory errors and time-of-check time-of-use (TOCTOU) errors that static analysis might miss [14]. While dynamic analysis can detect runtime-specific vulnerabilities and provide realistic attack scenarios, it requires a running system and may not achieve complete code coverage. Both static and dynamic approaches are often used in complementary ways in embedded systems security testing to provide comprehensive vulnerability coverage [16].

3) *Specialized tools for embedded systems*: For embedded systems specifically, specialized tools have emerged to address unique challenges. EMBA (EMBedded Analyzer) is an open-source firmware security analyzer designed for product se-

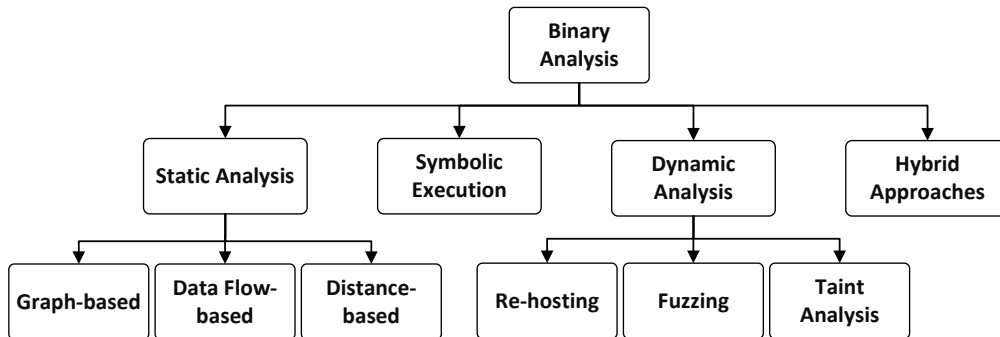


Fig. 1. Binary code vulnerability analysis techniques

curity teams and penetration testers, supporting the complete analysis process from firmware extraction through static and dynamic analysis via QEMU emulator [17], [18]. EMBA automatically identifies potential vulnerabilities in firmware, including insecure binaries, outdated software components, hard-coded passwords, and weak cryptographic implementations. EMBark serves as the enterprise web-based interface for EMBA, providing centralized firmware analysis with multiuser support and comprehensive interactive reporting dashboards [17]. EMBark also allows organizations to track device security over time across firmware versions and integrate firmware analysis into enterprise security workflows.

B. AI-powered tools

Recent advances in artificial intelligence have significantly impacted vulnerability detection, enabling more effective and scalable analysis of software code. This subsection focuses on two prominent AI-powered tools, VulBERTa and PalmTree, that significantly support vulnerability detection in embedded devices.

1) *VulBERTa*: VulBERTa is a transformer-based deep learning model designed specifically for detecting security vulnerabilities in C/C++ source code [6]. This approach leverages the RoBERTa (Robustly Optimized BERT pre-training Approach) architecture as its foundation, but incorporates a custom tokenization pipeline tailored for source code analysis. Unlike traditional natural language processing models, VulBERTa's tokenizer is designed to handle code-specific elements such as function calls and programming constructs, enabling it to capture both syntactic and semantic patterns unique to C/C++ code.

That model operates through a two-phase training methodology: pre-training and fine-tuning. During the pre-training phase, VulBERTa learns general code representations by processing large amounts of C/C++ code from open-source projects using MLM (Masked Language Modeling), similar to BERT's approach, but tailored for programming languages. In the fine-tuning phase, the pre-trained model is combined with either a Multi-Layer Perceptron (MLP) or Convolutional Neural Network (CNN) classification head to perform vulnerability detection tasks. That approach allows the model to first understand general code semantics and then specialize in identifying vulnerable patterns, achieving notable performance across multiple vulnerability-detection benchmarks, such as the Vuldeepecker [19] and Draper [20] datasets.

2) *PalmTree*: PalmTree is a pre-trained assembly language model specifically designed to generate high-quality instruction embeddings for binary code analysis tasks [7]. Unlike existing instruction embedding approaches that treat assembly instructions simplistically, PalmTree addresses the challenges in assembly code representation by capturing complex instruction structures and handling the noise introduced by compiler optimizations.

That model employs three distinct pre-training tasks designed to capture various characteristics of assembly language. First, the Masked Language Modeling (MLM) randomly

masks 15% of instruction tokens to help the model understand instruction context and syntax. Then, Context Window Prediction trains the model to determine whether two instructions appear within a specific context window, which captures the relationships between local instructions. Lastly, the Def-Use Prediction leverages data-flow graph information to predict whether instruction pairs have a definition-use relationship, thereby helping the model better understand data dependencies between instructions. These training tasks enable PalmTree to overcome limitations of other available approaches and generate embeddings that efficiently capture both syntactic and semantic properties of assembly code, leading to superior performance in binary code similarity detection, function prototype inference, and value set analysis.

C. Summary

Prior research in vulnerability detection has predominantly focused on general-purpose software, with existing machine learning-based solutions largely optimized for high-level languages and desktop environments. Although recent advances in representation learning [14], graph-based code modeling [21], and hierarchical attention mechanisms [22] have significantly improved automated vulnerability detection, these methods remain primarily developed and evaluated for source-code settings rather than embedded assembly code. In particular, there is still a clear gap in publicly available pipelines that are tailored to embedded platforms and their low-level code characteristics, including architecture-specific instruction semantics and resource-constrained deployment scenarios. This limitation underscores the need for specialized approaches that can operate effectively on embedded assembly while remaining adaptable to future extensions of the framework.

III. ARCHITECTURE

Our approach is a specialized software pipeline designed to detect vulnerabilities in embedded system assembly code, leveraging machine learning tailored to this task. By leveraging the VulBERTa algorithm and the PalmTree tokenizer, it provides a robust, scalable, and flexible foundation for analyzing binary code that balances efficiency with accuracy.

Our pipeline supports incremental refinements via targeted parameter adjustments. The system is organized into discrete, interoperable modules that collectively enable the processing, tokenization, and feature extraction from raw assembly input. This architecture leverages pretraining and fine-tuning strategies to optimize representation quality and detection accuracy, with particular attention to the challenges posed by low-level code structures. The pipeline's emphasis on modularity ensures that the system may evolve as new threat models and dataset characteristics emerge, making it a robust foundation for ongoing research in vulnerability detection for embedded systems.

Fig. 2 illustrates the complete two-stage workflow of the proposed pipeline. In the pretraining phase, raw assembly code from the pretraining dataset is first converted into instruction-level embeddings by PalmTree, and these embeddings are

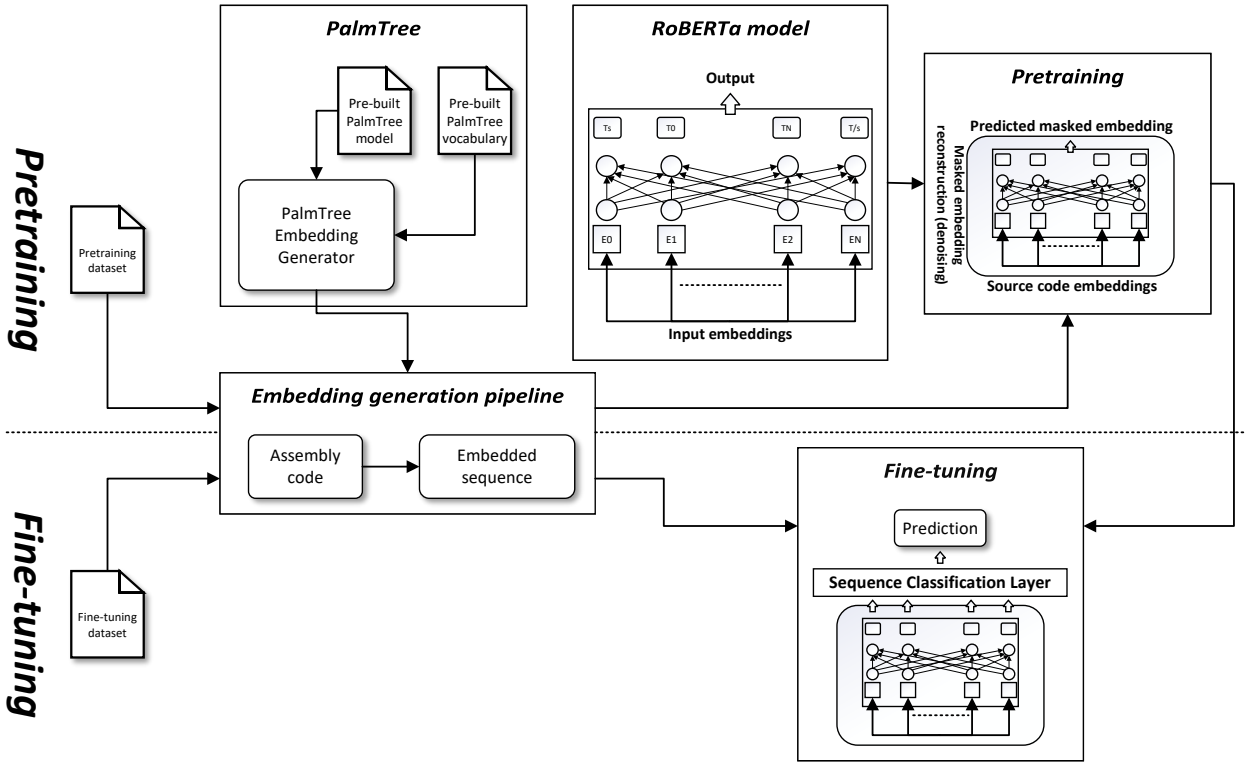


Fig. 2. Diagram of the proposed pipeline

then used as input to the RoBERTa-based masked reconstruction model. In the fine-tuning phase, the same embedding generation pipeline is applied to the fine-tuning dataset, and the resulting embeddings are then used to further fine-tune the model parameters before the sequence classification layer produces the final vulnerability prediction.

Below, we formally describe the proposed pipeline for assembly-level vulnerability detection. The framework operates on disassembled binary code and consists of two main stages: (1) embedding-level pretraining and (2) supervised fine-tuning.

A. Problem Formulation

Let $x = (i_1, i_2, \dots, i_n)$ denote a sequence of assembly instructions extracted from a single function, where each i_k represents one instruction.

The goal is to learn a function:

$$f(x) \rightarrow y \quad (1)$$

where $y \in \{0, 1\}$ indicates whether the input sequence contains a vulnerability.

B. Assembly Embedding Representation

Each instruction i_k is mapped to a continuous vector representation using a pretrained PalmTree model:

$$e_k = \phi(i_k), \quad e_k \in \mathbb{R}^d \quad (2)$$

This results in an embedding sequence:

$$E = (e_1, e_2, \dots, e_n), \quad E \in \mathbb{R}^{n \times d} \quad (3)$$

The embedding function $\phi(\cdot)$ remains fixed during both pretraining and fine-tuning.

C. Embedding-Level Masked Modeling

Unlike traditional masked language modeling, which operates on discrete tokens, we apply masking directly in the continuous embedding space.

A subset $M \subset \{1, \dots, n\}$ of positions (selected arbitrarily to 15%, following standard masked language modeling practices to balance learning signal and task difficulty) is selected for masking. For each $k \in M$, the embedding is replaced by Gaussian noise:

$$\tilde{e}_k = z_k, \quad z_k \sim \mathcal{N}(0, \sigma^2 I), \quad (4)$$

where I denotes the identity matrix, implying independent Gaussian noise applied to each embedding dimension. We replace selected embeddings with samples drawn from a Gaussian distribution rather than setting them to zero, so that the model observes a perturbed but still vector-valued input at those positions. This design encourages robustness to noisy or uncertain context while preserving the original embedding dimensionality and avoiding the degenerate effect of a constant zero vector. The value of $\sigma = 0.02$ was selected empirically.

The resulting corrupted sequence is denoted as:

$$\tilde{E} = (\tilde{e}_1, \dots, \tilde{e}_n) \quad (5)$$

The model is trained to reconstruct the original embeddings, with the learning signal driven primarily by masked positions:

$$\hat{e}_k = f_\theta(\tilde{E})_k \quad (6)$$

The pretraining objective minimizes a reconstruction loss defined over the entire embedding sequence:

$$\mathcal{L}_{MLM} = \frac{1}{n} \sum_{k=1}^n \|\hat{e}_k - e_k\|_2^2 \quad (7)$$

where $\hat{e}_k, e_k \in \mathbb{R}^d$, $\|\cdot\|_2^2$ denotes the squared Euclidean norm, measuring the reconstruction error between the predicted and original embeddings at position k ; $\mathcal{L}_{MLM}$ corresponds to the mean reconstruction error across all positions in the sequence and serves as the masked language modeling loss during pretraining. Euclidean distance was used because it provides a simple and interpretable measure of the straight-line displacement between the original and perturbed embeddings in the continuous embedding space.

D. Transformer Encoder

During pretraining, the corrupted embedding sequence $\tilde{E}$ is processed by a transformer encoder instantiated as a RoBERTa model operating directly on continuous PalmTree embeddings. Concretely, we use the encoder part of RoBERTa and feed it precomputed input embeddings bypassing the standard token embedding layer. This encoder applies multi-head self-attention and feed-forward transformations over the entire sequence and produces contextualized hidden states:

$$H = (h_0, h_1, \dots, h_n), \quad h_i \in \mathbb{R}^d. \quad (8)$$

These representations capture both local instruction patterns and long-range dependencies within the assembly sequence.

Please note that we are not training the embedding generator model itself. The model we used was already trained on a dataset of 3266 various binaries by the authors of PalmTree and provided for further research and testing. We do not exclude the possibility of training PalmTree on a custom dataset tailored to the assembly code of embedded devices in the future.

E. Fine-tuning for Vulnerability Detection

The pre-trained model is then fine-tuned for vulnerability detection. We achieve this by leveraging a dataset of labeled examples in which assembly code snippets are annotated as either vulnerable or non-vulnerable. Details regarding the datasets are provided in Section IV. The fine-tuning process adapts the model's representations and classifier layers to distinguish code with security flaws better.

For the classification task, a sequence-level representation is obtained using the CLS token (CLS stands for classification):

$$h_{CLS} = H_{[CLS]}, \quad (9)$$

where h_{CLS} corresponds to the hidden representation of the special classification token.

The vulnerability prediction logits are then computed as:

$$\mathbf{z} = Wh_{CLS} + b, \quad (10)$$

where $\mathbf{z} \in \mathbb{R}^2$ denotes the unnormalized scores for the vulnerable and non-vulnerable classes, and $W \in \mathbb{R}^{2 \times d}$, $b \in \mathbb{R}^2$ are the learnable parameters of the classification head.

The model is trained using the standard cross-entropy loss between the logits $\mathbf{z}$ and the ground-truth label $y \in \{0, 1\}$:

$$\mathcal{L}_{CLS} = \text{CE}(\mathbf{z}, y). \quad (11)$$

The described fine-tuning process allows the language model to specialize beyond generic code understanding, polishing its detection capabilities for embedded system vulnerabilities through supervised learning. This targeted adaptation is critical for high accuracy and precision in automated vulnerability analysis in embedded devices.

The overall training process of the proposed framework is summarized in Algorithm 1.

IV. DATASET

Selecting an appropriate training dataset for vulnerability detection in assembly code poses significant challenges. Currently, no existing datasets specifically address vulnerabilities occurring in assembly language. Two straightforward solutions to this problem are: (1) creating a fully synthetic dataset, or (2) collecting real-world vulnerable assembly code from a variety of devices. However, generating a synthetic dataset is non-trivial; accurately reproducing multiple distinct instances of the same vulnerability type to ensure robust model training is challenging. Although it is theoretically possible to generate various permutations of the same vulnerability by altering instruction order without affecting semantics, such an approach risks reducing model accuracy. Alternatively, one could attempt to mine the CVE MITRE database [23] for vulnerable source code from embedded devices, but sufficient coverage and diversity cannot be guaranteed. Due to these limitations, an alternative strategy was adopted.

A. Romeo and Juliet

During our study, we identified the NIST dataset called Juliet C/C++ 1.3 [24]. The Juliet dataset is a comprehensive collection of test cases developed by the National Institute of Standards and Technology (NIST) as part of the Software Assurance Reference Dataset (SARD). It comprises thousands of C and C++ code samples designed to represent a wide range of known software weaknesses, categorized under various Common Weakness Enumeration (CWE) identifiers.

The Juliet dataset is primarily composed of C and C++ source code, which is not ideal for training models targeting assembly code. However, our research identified the ROMEO dataset [25], which consists of assembly code generated by pre-compiling the Juliet test cases for the x86-64 ISA. This conversion makes ROMEO a suitable candidate for training

Algorithm 1 Training procedure for assembly-level vulnerability detection

Require: D_{pre} – unlabeled assembly dataset
 D_{cls} – labeled dataset (x_i, y_i)
 $\phi(\cdot)$ – PalmTree embedding generator
 p_{mask} – masking ratio
 σ^2 – Gaussian noise variance

Ensure: Trained model parameters θ, ψ
Initialize transformer encoder f_θ with token-wise prediction head g_θ

- 2: Freeze embedding generator $\phi(\cdot)$
- Stage I: Pretraining**
- 4: **for** epoch = 1 to T_{pre} **do**
- 4: **for all** $x \in D_{\text{pre}}$ **do**
- 6: $E \leftarrow \phi(x)$
- Sample masking set M with ratio p_{mask}
- 8: $\tilde{E} \leftarrow E$
- 9: **for all** $k \in M$ **do**
- 10: $\tilde{E}[k] \leftarrow \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I)$
- 11: **end for**
- 12: $H \leftarrow f_\theta(\tilde{E})$
- 13: $\hat{E} \leftarrow g_\theta(H)$ {token-wise reconstruction}
- 14: $\mathcal{L}_{MLM} \leftarrow \frac{1}{n} \sum_{k=1}^n \|\hat{E}[k] - E[k]\|_2^2$
- Update θ using $\nabla_\theta \mathcal{L}_{MLM}$
- 16: **end for**
- 17: **end for**
- 18: Attach sequence-level classification head g_ψ
- Stage II: Fine-tuning**
- 20: **for** epoch = 1 to T_{cls} **do**
- 21: **for all** $(x, y) \in D_{\text{cls}}$ **do**
- 22: $E \leftarrow \phi(x)$
- 23: $H \leftarrow f_\theta(E)$
- 24: $z \leftarrow H[\text{CLS}]$ {sequence representation}
- 25: $\mathbf{z} \leftarrow g_\psi(z)$ {logits $\mathbf{z} \in \mathbb{R}^2$ }
- 26: $\mathcal{L}_{CLS} \leftarrow \text{CE}(\mathbf{z}, y)$
- Update θ, ψ using $\nabla_{\theta, \psi} \mathcal{L}_{CLS}$
- 28: **end for**
- 29: **end for**
- 30: **return** θ, ψ

our framework, as it directly supports the objective of detecting vulnerabilities in assembly language.

The ROMEO dataset was originally provided in machine code format. To enable its use with our framework, several preprocessing steps were required. First, the dataset was disassembled into a human-readable Intel-syntax assembly representation using GNU Binutils’ `objdump` with the `-d -M intel` options [26], to match the input format required by PalmTree’s assembly tokenization pipeline. Subsequently, a custom script was developed to remove unnecessary elements such as instruction addresses, machine code bytes, and function name headers. To prevent the model from learning spurious patterns from function names, all function call references were anonymized by removing the function names from the calls. Finally, individual functions were extracted and saved as

separate files to facilitate targeted analysis and training.

The pretraining dataset was created by removing all end-of-line characters from the extracted function files and concatenating them, so that each function from the ROMEO dataset is represented on a separate line.

For fine-tuning, additional preprocessing was necessary. Unlike pretraining, which focuses on learning code semantics, fine-tuning aims to enable the model to distinguish vulnerable code from non-vulnerable code. Therefore, the fine-tuning dataset includes labels indicating the vulnerability status of each line, providing the model with essential information for supervised learning.

Each function in the original ROMEO dataset was labeled as either “good” or “bad” by appending a “_good” or “_bad” suffix to its name. However, during pretraining dataset preparation, all function names were removed, eliminating any explicit indication of vulnerability status. To address this, we developed a separate parsing script for the fine-tuning dataset. This script similarly removes function names but appends a “~1” or “~0” marker at the end of each function to denote vulnerable and non-vulnerable code, respectively. This explicit labeling enables the model to distinguish functions based on their vulnerability during training correctly. The diagram of the dataset preparation process is shown in Fig. 3.

In the ROMEO dataset, each function invoked by the main function is also annotated with either a “_good” or “_bad” suffix to indicate its vulnerability status. Notably, if the main function itself is not vulnerable but calls a vulnerable function, the called function is assigned a “G2B” (good to bad) suffix. Conversely, the suffix “B2G” is used when a vulnerable function calls a non-vulnerable one. We have included these functions in our dataset and classified them accordingly. However, by separating all functions into individual files, some contextual and semantic information regarding the overall vulnerable code structure may have been lost. Preserving this information represents an important consideration for future enhancements of the framework.

Creating a single comprehensive fine-tuning dataset would require substantial storage space, demand considerable RAM during training, and result in prolonged processing times. Additionally, the large volume of data raised concerns about potential degradation in model accuracy. To address these challenges, we opted to fine-tune the pretrained model separately for each CWE vulnerability represented in the ROMEO dataset. This approach not only improves detection performance but also enables end-users to target specific vulnerabilities by applying the corresponding specialized model to any given assembly code.

Although ROMEO is based on x86-64 binaries, it serves as a proxy benchmark for assembly-level vulnerability detection applicable to embedded firmware analysis.

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

1) *Implementation Details:* The proposed framework was implemented using the PyTorch deep learning library and

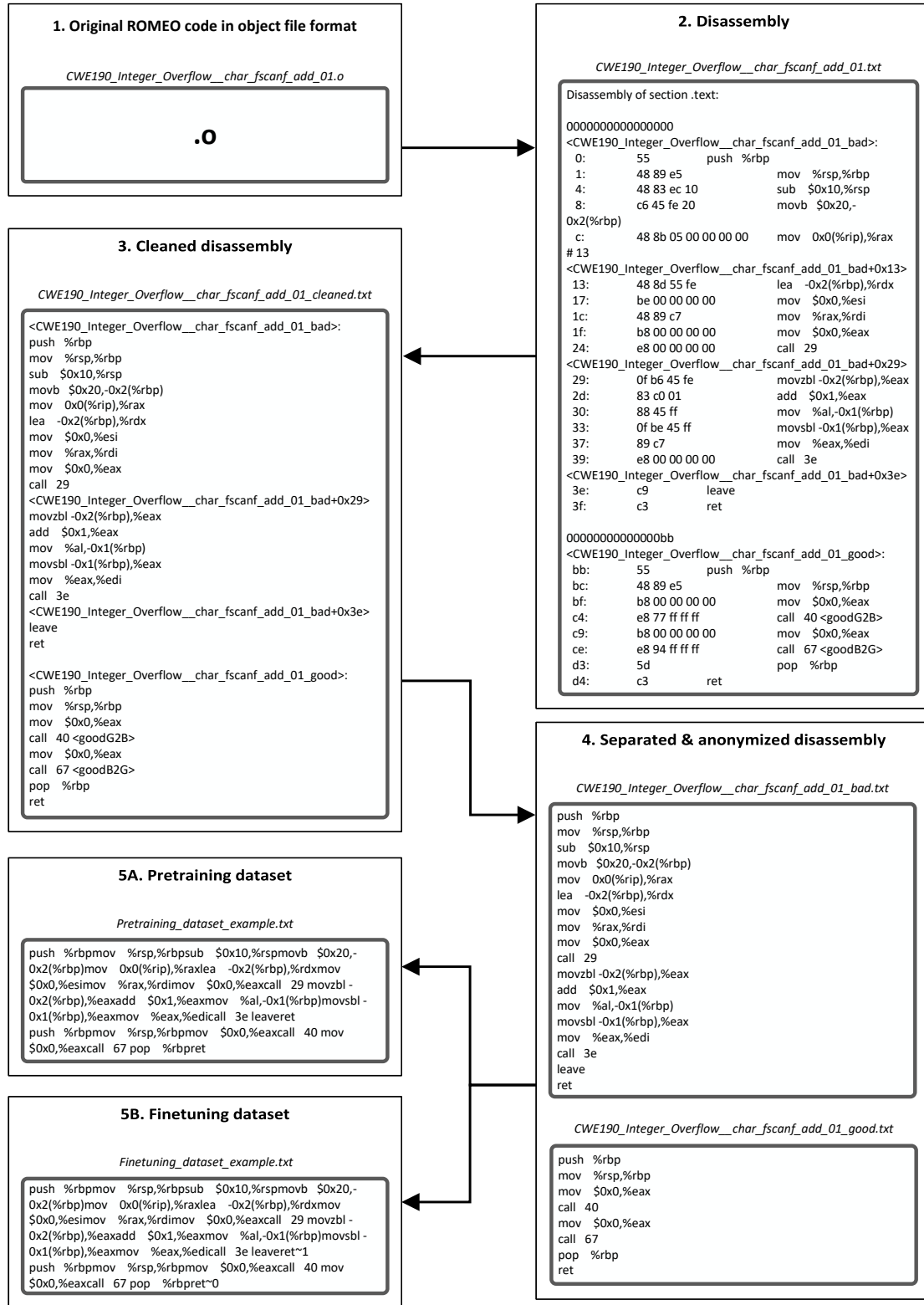


Fig. 3. Example of dataset preprocessing pipeline for assembly-level training data preparation

the HuggingFace Transformers toolkit [27]. The RoBERTa-based model was adapted to accept continuous embedding inputs instead of discrete token IDs. PalmTree was used as a fixed embedding generator without further fine-tuning. All experiments were conducted using Python 3.8.20 on Windows and Linux-based systems.

2) *Model Configuration*: The transformer model follows a RoBERTa-base configuration with modifications to support embedding-level inputs. The key hyperparameters are as follows:

- Number of transformer layers: 6
- Hidden size: 128
- Number of attention heads: 4
- Intermediate (feed-forward) size: 512
- Dropout rate: 0.1
- Maximum sequence length: 512+2

PalmTree embeddings are used as input representations and remain fixed during both pretraining and fine-tuning.

3) *Pretraining Procedure*: In the pretraining stage, we use an embedding-level denoising objective: for each sequence, 15% of embedding positions are replaced with small-variance Gaussian noise, and the model is trained with an MSE loss to reconstruct the original PalmTree embeddings at those positions.

The model is trained to reconstruct the original embeddings using mean squared error (MSE) loss:

- Optimizer: AdamW
- Learning rate: 5×10^{-5}
- Batch size: 32
- Number of epochs: 10

Pretraining is performed on a pretraining dataset comprising 10% of randomly selected functions derived from the ROMEO dataset. The dataset contains 9,320 functions (approximately 3 million instructions).

Fine-tuning datasets are constructed from the ROMEO dataset by extracting every function associated with a given vulnerability type and labeling them as vulnerable or nonvulnerable. For vulnerability types with disproportionately large numbers of nonvulnerable functions, we downsampled the nonvulnerable class to obtain roughly balanced numbers of examples. The exact counts of vulnerable and non-vulnerable functions for each fine-tuning dataset used for testing are reported in Table I.

4) *Fine-tuning Procedure*: During fine-tuning, the pre-trained model is extended with a sequence classification head to predict vulnerability labels (binary classification).

- Loss function: Cross-entropy
- Optimizer: AdamW
- Learning rate: 3×10^{-5}
- Batch size: 4
- Number of epochs: 10

5) *Dataset Splitting*: The dataset is split into training and validation sets using an 80/20 ratio. Splitting is performed at the level of individual functions (lines in the dataset).

To ensure reproducibility, all experiments are conducted with a fixed random seed.

6) *Evaluation Metrics*: We evaluate model performance using multiple metrics to account for class imbalance:

- Accuracy
- Precision
- Recall
- F1-score
- Precision-Recall Area Under the Curve (PR-AUC)

Given the imbalance present in several CWE classes, particular emphasis is placed on F1-Score and PR-AUC as more reliable indicators of model performance.

7) *Hardware Setup*: Experiments were conducted on two computational platforms:

- AMD EPYC 7H12 CPU with 512 GB RAM
- Intel i7-14700 CPU with 128 GB RAM

The training code was later parallelized with GPU support and tested on an NVIDIA RTX 4000 Ada Generation GPU, yielding results consistent with the original CPU-only runs. This indicates that the proposed pipeline is stable and reproducible across different hardware platforms. All finetuning experiments run on a single GPU completed within 80 minutes per configuration, enabling practical retraining and replication of our results. Classification is performed on the fly and does not introduce a noticeable delay in the analysis workflow. However, we did not conduct a dedicated benchmark for classification time, and therefore leave a precise runtime characterization for future work.

B. Results

Across multiple vulnerability classes, the models demonstrate consistently high performance on key metrics, including accuracy (mean: 0.89), F1-score (mean: 0.85), and PR-AUC (mean: 0.89). These results indicate that the proposed method can reliably identify vulnerabilities within embedded system assembly code under a diverse set of conditions. The results of the evaluation are shown in Table I.

The impact of class imbalance is evident in several cases. In most tests, scenarios with vulnerabilities and low positive case proportions exhibit recall and PR-AUC scores notably lower than those with balanced datasets. For example, CWE-367 exhibits high accuracy (0.9879) but significantly lower recall (0.3556), highlighting the misleading nature of accuracy in highly imbalanced scenarios. Such results suggest that despite high accuracy, the model's capacity to detect rare vulnerabilities can be significantly hindered by unbalanced datasets, leading to lower recall and diminished overall effectiveness. Addressing these imbalances through tailored sampling or augmentation techniques remains an important direction for future experiment design.

The pipeline shows competitive performance across a wide range of vulnerability classes. However, the analysis indicates that metrics beyond accuracy, such as F1-score and PR-AUC, are most important for interpreting results correctly, especially in scenarios with sparse positive samples. The results advocate for focused fine-tuning for difficult-to-detect classes such as TOCTOU, which may enhance the pipeline's robustness and real-world applicability.

TABLE I
PER-CWE VULNERABILITY DETECTION PERFORMANCE ON SELECTED ROMEO VULNERABILITIES

CWE	Positive cases	Negative cases	Pos/Neg Ratio	Accuracy	Precision	Recall	F1-Score	PR-AUC
23	5980	6137	0.97	0.9072	0.9326	0.8802	0.9057	0.9754
121	20376	20311	1.00	0.8613	0.8259	0.9141	0.8678	0.9274
122	26688	26322	1.01	0.8740	0.8163	0.9636	0.8839	0.8975
190	40248	20112	2.00	0.9102	0.9059	0.9648	0.9344	0.9556
195	7152	7156	1.00	0.8864	0.8579	0.9296	0.8923	0.9283
196	98	206	0.48	0.9016	0.7600	1.0000	0.8636	0.9249
197	5364	5116	1.05	0.8392	0.8821	0.7991	0.8385	0.9160
366	196	206	0.95	0.9136	0.8571	1.0000	0.9231	0.8965
367	196	20513	0.01	0.9879	0.4324	0.3556	0.3902	0.4473
369	8856	8171	1.08	0.8203	0.7773	0.9276	0.8458	0.8594
398	984	1026	0.96	0.9279	0.9259	0.9390	0.9324	0.9568
401	13467	13247	1.02	0.8645	0.8219	0.9386	0.8764	0.9102
415	8380	20431	0.41	0.8629	0.7294	0.8726	0.7946	0.8301
416	3505	3072	1.14	0.9461	0.9408	0.9628	0.9517	0.9680
476	2838	2049	1.39	0.8456	0.8272	0.9428	0.8812	0.9079
546	490	411	1.19	0.9448	0.9245	0.9800	0.9515	0.9823
563	3196	3073	1.04	0.8636	0.8291	0.9333	0.8781	0.9067
570	64	206	0.31	0.8519	0.7692	0.6667	0.7143	0.8179
571	64	206	0.31	0.8519	0.6667	0.9333	0.7778	0.8290
590	13566	13247	1.02	0.8976	0.8743	0.9350	0.9036	0.9398

Comparison with graph-based methods and token-level transformers is left for future work due to incompatibility with embedding-level inputs.

C. Ablation studies

Using CWE-415 (Double Free) as a case study, we systematically varied the number of negative ROMEO cases while keeping the 8380 positive examples fixed, obtaining different positive-to-negative ratios. This allowed us to explicitly examine how varying levels of class imbalance affect the detector’s robustness. This evaluation shows that class balance yields the best compromise between most metrics (accuracy 0.8689, precision 0.8406, recall 0.9164, F1-score 0.8768, PR-AUC 0.9234), whereas extreme imbalance significantly degrades all scores. The exact metric values for each class ratio are reported in Table II.

TABLE II
IMPACT OF CLASS IMBALANCE ON CWE-415 (DOUBLE FREE)
DETECTION METRICS

Pos.	Neg.	Pos/Neg	Acc.	Prec.	Rec.	F1	PR-AUC
8380	20431	0.41	0.8629	0.7294	0.8726	0.7946	0.8301
8380	61291	0.14	0.9254	0.6842	0.6716	0.6778	0.6683
8380	8173	1.03	0.8689	0.8406	0.9164	0.8768	0.9234

To assess the model’s robustness to random initialization, we repeated training on the CWE-190 finetuning dataset 10 times with different random seeds, keeping all other conditions fixed. Across runs, we observed only minor fluctuations in performance, with standard deviations of 0.0071 for Accuracy, 0.0099 for Precision, 0.0086 for Recall, 0.0049 for F1-Score, and 0.0036 for PR-AUC. This indicates that the model’s behavior is stable and that the reported results are not an artifact of a particularly favorable random seed. Additionally, the training and fine-tuning times remain very similar across runs, further confirming that the experimental setup is repetitive not only in terms of predictive performance but also in

computational cost. The detailed results of these ten runs are summarized in Table III.

TABLE III
ROBUSTNESS TO RANDOM INITIALIZATION ON CWE-190 (INTEGER OVERFLOW)

Seed No.	Accuracy	Precision	Recall	F1-Score	PR-AUC
1	0.9102	0.9059	0.9648	0.9344	0.9556
2	0.9058	0.9119	0.9522	0.9316	0.9561
3	0.9169	0.9079	0.9741	0.9398	0.9589
4	0.8933	0.8861	0.9640	0.9234	0.9473
5	0.9103	0.9021	0.9716	0.9356	0.9534
6	0.9080	0.8975	0.9717	0.9331	0.9526
7	0.9016	0.9038	0.9551	0.9288	0.9492
8	0.9178	0.9183	0.9617	0.9395	0.9543
9	0.9095	0.8975	0.9749	0.9346	0.9517
10	0.9050	0.8892	0.9773	0.9312	0.9493

We also investigate the impact of the maximum input sequence length during finetuning by comparing models trained with maximum sequence lengths of 512 and 256 PalmTree embeddings on the same set of CWE datasets. Surprisingly, reducing the sequence length from 512 to 256 tokens does not degrade performance; on the contrary, the 256-token variant attains slightly higher average Precision (0.824 vs. 0.818), Recall (0.902 vs. 0.892), and F1-Score (0.858 vs. 0.850), with almost identical Accuracy and PR-AUC. This suggests that, for the evaluated datasets, the information captured within the first 256 embeddings is sufficient for effective vulnerability classification, as extending the context to 512 embeddings does not improve the metrics. The detailed per-CWE results for both sequence length settings are summarized in Table IV.

VI. DISCUSSION

The proposed pipeline employs a novel dual-model architecture that combines a RoBERTa-based transformer with pretrained PalmTree embeddings for assembly-level analysis. While this approach requires higher computational overhead

TABLE IV
EFFECT OF FINETUNING SEQUENCE LENGTH (256 VS. 512) ON
VULNERABILITY DETECTION PERFORMANCE

Seq. Len.	CWE	Accuracy	Precision	Recall	F1-Score	PR-AUC
256	23	0.90347	0.92184	0.88427	0.90266	0.96789
512		0.90718	0.93264	0.88020	0.90566	0.97535
256	121	0.85156	0.82739	0.88700	0.85616	0.91759
512		0.86127	0.82590	0.91414	0.86778	0.92735
256	122	0.87380	0.81707	0.96171	0.88351	0.89692
512		0.87399	0.81631	0.96361	0.88387	0.89754
256	190	0.90631	0.91138	0.95113	0.93083	0.95219
512		0.91021	0.90588	0.96475	0.93439	0.95555
256	195	0.87561	0.82971	0.94890	0.88531	0.90986
512		0.88644	0.85787	0.92956	0.89228	0.92831
256	196	0.91803	0.79167	1.00000	0.88372	0.92489
512		0.90164	0.76000	1.00000	0.86364	0.92489
256	197	0.84351	0.84456	0.85845	0.85145	0.91033
512		0.83922	0.88206	0.79909	0.83852	0.91600
256	366	0.91358	0.85714	1.00000	0.92308	0.90622
512		0.91358	0.85714	1.00000	0.92308	0.89652
256	367	0.99010	0.57143	0.35556	0.43836	0.46678
512		0.98793	0.43243	0.35556	0.39024	0.44730
256	369	0.81885	0.77077	0.93812	0.84625	0.85455
512		0.82032	0.77731	0.92762	0.84584	0.85942
256	398	0.93284	0.93458	0.93897	0.93677	0.96244
512		0.92786	0.92593	0.93897	0.93240	0.95677
256	401	0.85121	0.83839	0.87857	0.85801	0.87550
512		0.86450	0.82191	0.93855	0.87637	0.91016
256	415	0.86795	0.74146	0.86808	0.79979	0.84421
512		0.86292	0.72936	0.87264	0.79459	0.83006
256	416	0.92173	0.90194	0.96281	0.93138	0.97275
512		0.94605	0.94078	0.96281	0.95167	0.96801
256	476	0.84458	0.82596	0.94276	0.88050	0.90677
512		0.84560	0.82718	0.94276	0.88120	0.90792
256	546	0.94475	0.92453	0.98000	0.95146	0.97527
512		0.94475	0.92453	0.98000	0.95146	0.98225
256	563	0.84928	0.82041	0.91364	0.86452	0.89978
512		0.86364	0.82907	0.93333	0.87812	0.90673
256	570	0.90741	0.81250	0.86667	0.83871	0.82293
512		0.85185	0.76923	0.66667	0.71429	0.81791
256	571	0.85185	0.66667	0.93333	0.77778	0.84068
512		0.85185	0.66667	0.93333	0.77778	0.82900
256	590	0.90919	0.86291	0.97857	0.91711	0.94859
512		0.89763	0.87432	0.93498	0.90363	0.93981

than single-model solutions, including extended processing times, increased memory requirements, and higher energy consumption, the performance benefits justify these costs. This combination proves particularly effective, especially in scenarios where source code is unavailable, as compiler optimizations and hardware-specific implementations can introduce security vulnerabilities not visible in source code analysis alone.

Despite its synthetic origin, the ROMEO dataset remains one of the few publicly available large-scale resources for assembly-level vulnerability detection, making it suitable for initial benchmarking.

The model performs less reliably on highly imbalanced CWE classes, where rare vulnerability patterns are underrepresented. This suggests that the current approach may struggle with low-frequency vulnerabilities and could benefit from data augmentation or class-balanced training strategies.

The improved performance can be attributed to task-specific finetuning on PalmTree embeddings, which allows the transformer encoder to specialize its contextual representations for vulnerability detection beyond what is captured by the fixed embedding generator.

VII. CONCLUSIONS AND FUTURE WORK

This paper presented a software pipeline for automatically detecting vulnerabilities in embedded systems' assembly code. It integrates two main algorithms: RoBERTa-based transformer architecture and PalmTree. This approach provides an effective solution for vulnerability detection in the resource-constrained environment of embedded systems.

The research contributes to embedded systems security by enabling automated vulnerability detection at the assembly level, bypassing issues with source code availability that often obscure vulnerabilities in embedded firmware. The pipeline's ability to handle diverse vulnerability patterns through masked language modeling and attention mechanisms represents a substantial advancement over traditional static analysis methods that struggle with embedded platform diversity.

Future work should focus on several key areas to enhance the pipeline's effectiveness and practical applicability. First, a comprehensive evaluation on real embedded device firmware is essential to validate the system's performance beyond synthetic vulnerability datasets and demonstrate its effectiveness in production environments. Second, architectural improvements should be explored, particularly investigating graph-based neural networks and hierarchical attention mechanisms that can better capture control-flow relationships and long-range dependencies in assembly code. Finally, developing real-time deployment optimizations and end-user applications will be crucial for translating this research into practical tools that can be integrated into embedded development workflows and security analysis pipelines.

In parallel, a promising direction is to retrain PalmTree from scratch on ARM64 assembly, following the methodology adopted by the original authors for x86. This would adapt the embedding model to ARM64 semantics and better align the representation layer with the embedded systems domain. In the next stage, the project would move toward a fully ARM64-based platform, reflecting the architecture more commonly associated with modern embedded deployments and their operational constraints. We do not exclude future work on RISC-V as well, especially given its growing relevance in embedded and safety-critical systems.

This pipeline establishes a foundation for automated security analysis of embedded systems, becoming increasingly essential as embedded devices proliferate in IoT and critical infrastructure. This work demonstrates that machine learning approaches can effectively address the complexity challenges present in embedded systems vulnerability detection, paving the way for more secure and resilient embedded device ecosystems.

ACKNOWLEDGMENT

This work was supported by Rector of the West Pomeranian University of Technology in Szczecin for PhD students of the Doctoral School, grant number: ZUT/19/2025.

GENERATIVE AI DISCLOSURE STATEMENT

During the preparation of this work, the authors used Perplexity AI (mainly the Nemotron 3 Super model) and Grammarly for text editing and style improvement. The authors take full responsibility for the content of this publication.

REFERENCES

- [1] P. Marwedel, *Embedded system design: Embedded systems foundations of cyber-physical systems*. Dordrecht: Springer Netherlands, 2011. <https://dx.doi.org/10.1007/978-94-007-0257-8>
- [2] M. Eceiza, J. L. Flores, and M. Iturbe, “Fuzzing the Internet of Things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems,” *IEEE Internet Things J.*, vol. 8, no. 13, pp. 10390–10411, Jul. 2021. <https://dx.doi.org/10.1109/JIOT.2021.3056179>
- [3] S. Taviss, S. H. H. Ding, M. Zulkernine, P. Charland, and S. Acharya, “Asm2Seq: Explainable assembly code functional summary generation for reverse engineering and vulnerability analysis,” *Digital Threats*, vol. 5, no. 1, Art. no. 6, pp. 1–25, Mar. 2024. <https://dx.doi.org/10.1145/3592623>
- [4] D. Liu, Y. Tang, B. Wang, W. Xie, and B. Yu, “Automated vulnerability detection in embedded devices,” in *Advances in Digital Forensics XIV*, G. Peterson and S. Shenoi, Eds. Cham: Springer International Publishing, 2018, pp. 313–329. https://dx.doi.org/10.1007/978-3-319-99277-8_17
- [5] A. Aumpansub and Z. Huang, “Learning-based vulnerability detection in binary code,” in *Proc. 14th Int. Conf. Machine Learning and Computing (ICMLC)*, 2022. <https://dx.doi.org/10.1145/3529836.3529926>
- [6] H. Hanif and S. Maffei, “VulBERTa: Simplified source code pre-training for vulnerability detection,” in *Proc. Int. Joint Conf. Neural Networks (IJCNN)*, 2022. <https://dx.doi.org/10.1109/IJCNN55064.2022.9892280>
- [7] X. Li, Y. Qu, and H. Yin, “PalmTree: Learning an assembly language model for instruction embedding,” in *Proc. 2021 ACM SIGSAC Conf. Computer and Communications Security (CCS)*, Nov. 2021, pp. 3236–3251. <https://dx.doi.org/10.1145/3460120.3484587>
- [8] A. Diwan, M. Q. Li, and B. C. M. Fung, “VDGraph2Vec: Vulnerability detection in assembly code using message passing neural networks,” in *2022 21st IEEE Int. Conf. Machine Learning and Applications (ICMLA)*, Nassau, Bahamas, Dec. 2022, pp. 1039–1046. <https://dx.doi.org/10.1109/ICMLA55696.2022.00173>
- [9] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, “Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies,” *ACM Comput. Surv.*, vol. 54, no. 2, pp. 1–42, Mar. 2022. <https://dx.doi.org/10.1145/3432893>
- [10] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude, “An empirical study of static analysis tools for secure code review,” in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, 2024.
- [11] R. Venkitaraman and G. Gupta, “Static program analysis of embedded executable assembly code,” in *Proc. 2004 Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Washington, DC, USA, Sep. 2004, pp. 157–166. <https://dx.doi.org/10.1145/1023833.1023857>
- [12] R. Bagnara, A. Bagnara, and P. M. Hill, “The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software,” unpublished, Sep. 4, 2018. <https://dx.doi.org/10.48550/arXiv.1809.00821>
- [13] S. Rawat, D. Ceara, L. Mounier, and M.-L. Potet, “Combining static and dynamic analysis for vulnerability detection,” unpublished, May 16, 2013. <https://dx.doi.org/10.48550/arXiv.1305.3883>
- [14] Z. Qian, F. Zhong, Q. Hu, Y. Jiang, J. Huang, M. Ren, and J. Yu, “Software vulnerability analysis across programming language and program representation landscapes: A survey,” unpublished, Mar. 26, 2025. <https://dx.doi.org/10.48550/arXiv.2503.20244>
- [15] Z. Shen and S. Chen, “A survey of automatic software vulnerability detection, program repair, and defect prediction techniques,” *Security and Communication Networks*, vol. 2020, pp. 1–16, Sep. 2020. <https://dx.doi.org/10.1155/2020/8858010>
- [16] S. Kim, R. Y. C. Kim, and Y. B. Park, “Software vulnerability detection methodology combined with static and dynamic analysis,” *Wireless Pers. Commun.*, vol. 89, no. 3, pp. 777–793, Aug. 2016. <https://dx.doi.org/10.1007/s11277-015-3152-1>
- [17] A. Guzman and OWASP Contributors, “Firmware security testing methodology,” OWASP. [Online]. Available: <https://scriptingxss.gitbook.io/firmware-security-testing-methodology>. [Accessed: Aug. 27, 2025].
- [18] M. Messner and P. Eckmann, “Firmware security analyzer EMBA,” presented at TROOPERS22, Heidelberg, Germany, 2022. [Online]. Available: https://troopers.de/downloads/troopers22/TR22_EMBA.pdf. [Accessed: Aug. 27, 2025].
- [19] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, “VulDeePecker: A deep learning-based system for vulnerability detection,” in *Proc. 2018 Network and Distributed System Security Symp.*, 2018. <https://dx.doi.org/10.14722/ndss.2018.23158>
- [20] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, “Automated vulnerability detection in source code using deep representation learning,” unpublished, Nov. 28, 2018. <https://dx.doi.org/10.48550/arXiv.1807.04320>
- [21] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, “Vulnerability detection with graph simplification and enhanced graph representation learning,” unpublished, Feb. 9, 2023. <https://dx.doi.org/10.48550/arXiv.2302.04675>
- [22] Y. Chen, Z. Lin, and Z. Guo, “Application of hierarchical attention network in vulnerability detection model,” in *2024 2nd Int. Conf. Big Data and Privacy Computing (BDPC)*, Macau, China, Jan. 2024, pp. 43–48. <https://dx.doi.org/10.1109/BDPC59998.2024.10649344>
- [23] “The Common Vulnerabilities and Exposures (CVE) program,” The MITRE Corporation. [Online]. Available: <https://www.cve.org/>. [Accessed: Aug. 19, 2025].
- [24] National Institute of Standards and Technology, “Juliet C/C++ 1.3 test suite,” Software Assurance Reference Dataset (SARD). [Online]. Available: <https://samate.nist.gov/SARD/test-suites/112>. [Accessed: Aug. 19, 2025].
- [25] C.-A. Brust, T. Sonnekalb, and B. Gruner, “ROMEO: A binary vulnerability detection dataset for exploring Juliet through the lens of assembly language,” *Computers & Security*, vol. 128, Art. no. 103165, May 2023. <https://dx.doi.org/10.1016/j.cose.2023.103165>
- [26] “objdump(1) — Linux manual page,” man7.org. [Online]. Available: <https://man7.org/linux/man-pages/man1/objdump.1.html>. [Accessed: Aug. 19, 2025].
- [27] Hugging Face, “RoBERTa model documentation,” Hugging Face Transformers, 2024. [Online]. Available: https://huggingface.co/docs/transformers/model_doc/roberta. [Accessed: Sep. 1, 2025].