DeepVulMatch: Learning and Matching Latent Vulnerability Representations for Dual-Granularity Vulnerability Detection

Michael Fu, Trung Le, *Member, IEEE,* Van Nguyen, *Member, IEEE,*
Chakkrit (Kla) Tantithamthavorn, *Senior Member, IEEE, and* Dinh Phung, *Member, IEEE*

***Abstract*— Deep learning (DL) models are widely used to detect software vulnerabilities, but identifying vulnerabilities at the line level remains challenging due to varied coding styles and the spread of vulnerabilities across multiple lines. We observe that vulnerable line embeddings tend to form clusters in the feature space, which can help models capture hidden patterns more effectively. In this paper, we propose a novel approach that leverages vector quantization (VQ) and optimal transport (OT) to exploit the clustering characteristics of vulnerable line embeddings and enhance detection performance. Specifically, we extract vulnerable line embeddings from the training data to form a vulnerability collection, which we condense into a compact vulnerability codebook using VQ and OT. Inspired by static analysis tools that rely on pattern matching, our model uses this codebook to match latent vulnerability representations during inference. Our approach also introduces dual-granularity detection, predicting both vulnerable functions and, when a function is predicted vulnerable, identifying the specific vulnerable lines within it. We evaluate our approach against 12 baselines on two large-scale datasets of real-world open-source vulnerabilities. Our method achieves the highest F1 scores at both the function and line levels. The training code and pre-trained models are available at: https://github.com/awsm-research/DeepVulMatch.**

***Index Terms*—Software Vulnerability Detection, Fine-grained Vulnerability Detection, Optimal Transport, Vector Quantization.**

CWE-787 Example | Language: C

```
1   static sk_sp<SkImage> unPremulSkImageToPremul(SkImage* input) {
2   SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),
3                                        kN32_SkColorType, kPremul_SkAlphaType);
4   RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);
5   if (!dstPixels)
6     return nullptr;
7   return newSkImageFromRaster(
8       info, std::move(dstPixels),
9       static_cast<size_t>(input->width()) * info.bytesPerPixel());  // Vulnerable
10  }
```

CWE-787 Example | Language: C

```
1   GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size)
2   {
3     if (ms)
4     {
5       int32_t nestsize = (int32_t)ms->nest_size[ms->nest_level]; // Vulnerable
6       if (nestsize == 0 && ms->nest_level == 0)
7         nestsize = ms->buffer_size_longs;
8       if (size + 2 <= nestsize) return GPMF_OK;
9     }
10    return GPMF_ERROR_BAD_STRUCTURE;
11  }
```

Fig. 1. In both functions, the CWE-787 (Out-of-Bound Write) vulnerability is triggered by an inappropriate data type assignment. In the *unPremulSkImageToPremul* [1] function above, the *size_t* type should be changed to the *unsigned* type. In the *IsValidSize* [2] function below, the *int_32t* should be changed to *uint_32t* to prevent potential buffer overflow. Despite sharing the same vulnerability type and pattern, the vulnerable lines in each function and their context are different in their written form, variable names, and positions.

I. Introduction

THE number of software vulnerabilities has been escalating rapidly in recent years. In particular, the National Vulnerability Database (NVD) [3] reported 26,447 software vulnerabilities in 2023, soaring 40% from 18,938 in 2019. The extensive use of open-source libraries, in particular, may contribute to this rise in vulnerabilities. For instance, the Apache Struts vulnerabilities [4] indicate that this poses a tangible threat to organizations. The root cause of these vulnerabilities is often insecure coding practices, making the source code exploitable by attackers who can use them to infiltrate software systems and cause considerable financial and social harm.

To mitigate security threats, security experts leverage static analysis tools that check the code against a set of known patterns of insecure or vulnerable code, such as buffer overflow vulnerabilities and other common security flaws. Deep learning (DL)-based vulnerability detection (VD) methods have demonstrated higher accuracy compared to static analysis tools [5]. Moreover, recent studies have proposed line-level VDs, that can pinpoint vulnerable lines to minimize the manual analysis burden on security analysts [6–8].

In this paper, we consider a vulnerability scope of a function as the collection of all vulnerable lines in that function. As illustrated in Figure 1, each function consists of one vulnerable line with similar vulnerability scopes. This suggests that even if two functions contain the same CWE-787 out-of-bound write vulnerability (the top-1 dangerous CWE-ID in 2023 [9]), the specific vulnerable lines can be written in different ways and located in different parts of the code. Thus, identifying vulnerabilities at the line level is challenging for DL models.

Michael Fu is with School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia.
E-mail: michael.fu@unimelb.edu.au
The remaining authors are with the Faculty of Information Technology, Monash University, Melbourne, Australia.
E-mail: {trunglm, van.nguyen1, chakkrit, dinh.phung}@monash.edu
Manuscript received Feb 24, 2025; revised N/A.

On the other hand, we observed that the embeddings of vulnerable lines form clusters in the feature space, where similar embeddings are close to each other as presented in the upper part of Figure 7. Harnessing this cluster information could empower DL models to more effectively capture the latent patterns within the feature space, which could significantly enhance the accuracy of VD approaches for both function and line-level predictions. Nevertheless, our analysis reveals that state-of-the-art VD approaches have not effectively leveraged the information presented in vulnerability scopes.

To address this gap, we propose a novel DL framework that uses vector quantization (VQ) [10] with optimal transport (OT) [11] to aggregate similar vulnerability scopes in the training data, generating a "vulnerability codebook" consisting of compressed vulnerability patterns that encapsulate similar vulnerability scopes. We then introduce a vulnerability-matching technique to leverage the learned patterns during inference. It is worth noting that in our approach, the labels of vulnerable lines are only required during the training phase to construct our codebook. Importantly, our method does not depend on vulnerable line labels during the subsequent validation and testing phases.

OT, through the Wasserstein distance [12], captures both global variations across different types of vulnerabilities (e.g., control-flow vs. memory issues) and local variations within similar vulnerabilities, allowing us to model variations across different vulnerability scopes. This enables the formation of representative centroids, effectively condensing vulnerability vectors into a compact codebook. VQ [10] complements this process by efficiently assigning each vulnerability vector to its nearest centroid based on Euclidean distance, supporting discrete pattern learning and compression. Together, OT and VQ enable us to aggregate collected vulnerability vectors into representative patterns, which are then used in our subsequent vulnerability matching process. Below, we provide an overview of our DEEPVULMATCH approach.

First, we introduce an RNN-based line embedding approach to address the input length limitations commonly encountered in Transformer models for deep learning-based vulnerability detection [6, 7]. Prior work, such as by Reimers *et al.* [13], applied Transformer-based architectures with mean or max pooling to obtain line-level embeddings. In contrast, our method employs a shallow, learnable RNN layer to aggregate subword token embeddings into line embeddings. Our evaluation also confirms that this learnable aggregation enables the model to better capture contextual relationships within each line, improving representation quality compared to static mean/max pooling methods. Moreover, our approach allows the model to process up to 3,100 tokens, substantially exceeding the 512-token limit of standard Transformer input representations.

Next, we extract a set of vulnerable line embeddings from each vulnerable function in the training data, referring to each set as a vulnerability scope. In total, we collect 6,361 vulnerability scopes from the training set. Each scope—consisting of multiple vulnerable line embeddings—is then summarized into a single flat vulnerability vector using another shallow RNN, resulting in 6,361 vulnerability vectors.

However, the numerous vulnerability vectors will require intensive computation during our vulnerability-matching inference. Thus, we leverage the clustering characteristic of vulnerability vectors and apply the principle of VQ with OT to transfer the mass distribution of vulnerability vectors into a more compact distribution of vulnerability codebook involving vulnerability patterns representing similar vulnerability vectors. We effectively quantize 6,361 vectors into 150 patterns and ensure that similar vulnerability vectors are mapped to the same pattern in the embedding space. Ultimately, the codebook encapsulates important vulnerable line information from the training data.

Inspired by the pattern-matching concept used in program analysis tools [14, 15], we further propose a vulnerability matching to leverage critical vulnerable line information learned from our VQ and OT processes. During inference, our model matches the input program against all patterns in the learned vulnerability codebook. By examining all the vulnerability patterns in the codebook, the matching process enables a thorough search for potential vulnerabilities. We name this model DEEPVULMATCH, a deep learning-based vulnerability matching approach using vector quantization (VQ) and optimal transport (OT) for function and line-level VD. To the best of our knowledge, we are the first to exploit vulnerable line information presented in training data using VQ and OT along with a vulnerability matching process, allowing us to further improve the overall capability of DL-based VD.

It is important to note that our approach introduces a dual-granularity vulnerability detection framework that detects whether a function is vulnerable and, if so, pinpoints the specific vulnerable lines. To achieve this, we jointly train the model using both function-level and line-level labels, allowing it to exploit the mutual information between the two levels for improved accuracy. To ensure consistency between predictions, line-level outputs are considered only if the function is predicted as vulnerable.

We extensively evaluated our DEEPVULMATCH using two datasets: the Big-Vul dataset [16] and the D2A dataset [17]. These datasets contain line-level vulnerability labels and are widely used by prior VD works [6–8, 18, 19]. Specifically, the Big-Vul dataset comprises over 188K C/C++ functions with diverse real-world vulnerabilities extracted from multiple large-scale open-source software projects spanning from 2002 to 2019. Through this evaluation, we aim to address the following two research questions:

- **(RQ1) What is the accuracy of our DEEPVULMATCH for predicting function-level and line-level vulnerabilities?**
  Results. Our DEEPVULMATCH approach achieves the highest F1 scores for both function and line-level vulnerability predictions on both experimental datasets. Particularly noteworthy is the line-level F1 score of 82% achieved by DEEPVULMATCH on the Big-Vul dataset, surpassing the performance of the best baseline approach by 32%.
- **(RQ2) What are the contributions of each component in our DEEPVULMATCH approach?**
  Results. The ablation study confirms the effectiveness of our RNN-based line embedding over mean or max pool-

ing methods proposed in Sentence-BERT. Furthermore, our vulnerability matching approach, employing optimal transport (OT) and vector quantization (VQ), significantly improves the F1 score from 37% to 82%. Additionally, the study validates our decision regarding the number of patterns used for the OT process. These results validate the technical proposal within our DEEPVULMATCH approach.

Novelty & Contributions. To the best of our knowledge, the contributions of this paper are as follows:

- DEEPVULMATCH, an innovative DL-based vulnerability-matching framework utilizing the optimal transport (OT) theory and vector quantization (VQ) to locate line-level vulnerabilities.
- A novel line embedding approach using recurrent neural networks (RNNs) to represent code lines efficiently.
- A thorough evaluation of our method compared to other DL-based vulnerability prediction methods on two real-world vulnerability datasets.
- A comprehensive ablation study along with an extended discussion to investigate each component in our DEEPVULMATCH approach.

Paper Organization. Section II presents background knowledge of optimal transport (OT) and vector quantization (VQ) and the motivation for using them followed by related works. Section III describes the problem statement and motivation behind our design rationale. It is followed by the technical details of our DEEPVULMATCH approach, which aims to detect line-level vulnerabilities using vulnerability matching with OT and VQ. Section IV presents the experimental setup and results. Section V-A presents the extended discussion of our DEEPVULMATCH approach. Section VI discloses the threats to validity. Section VII draws the conclusions.

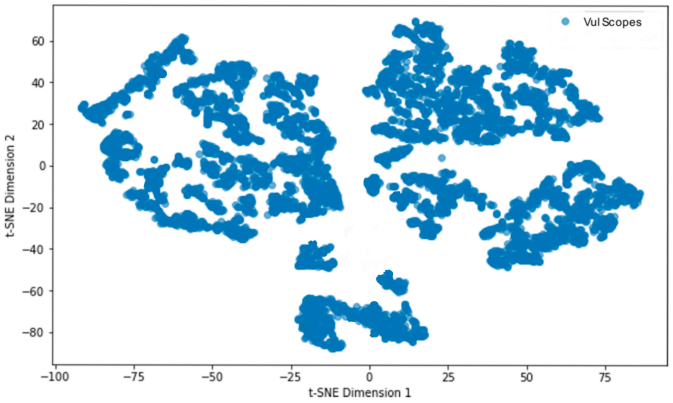

Fig. 2. The t-SNE visualization of vulnerability scopes.

II. BACKGROUND & RELATED WORK

To investigate potential structure in the representation space, we applied t-SNE to visualize the embedding space of vulnerable lines within vulnerable functions from the training data in our studied dataset. *We observed that vulnerability scopes tend to form clusters in the feature space, suggesting the presence of recognizable hidden patterns, as shown in Figure 2.* Thus, we anticipate that incorporating this clustering characteristic could substantially enhance the performance of vulnerability detection (VD) models.

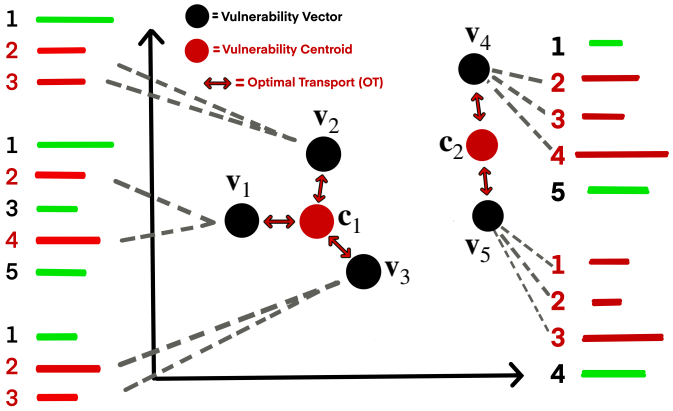

Fig. 3. The green lines are benign code lines while the red lines are vulnerable ones. We extract vulnerable lines as a vulnerability vector denoted as $\mathbf{v}$. We show that similar $\mathbf{v}$ sharing the same vulnerability pattern will stay close to each other in the feature space. Thus, they can be further grouped into vulnerability centroids denoted as $\mathbf{c}$ using optimal transport (OT).

To this end, we capture a set of distinct vulnerability scopes in the training set. In total, we curate 6,361 diverse vulnerability scopes, transforming each into a vulnerability vector $\mathbf{v}$ to compose our vulnerability collection $\mathcal{V}_v$. However, such a huge collection will introduce a computational burden during inference. To address this, we follow the principle of vector quantization (VQ) [10] and employ optimal transport (OT) [11] to quantize similar $\mathbf{v}$ instances into a vulnerability centroid $\mathbf{c}$ as depicted in Figure 3. This approach aligns with our observation that vulnerability scopes tend to cluster together, which effectively reduces the collection size. The outcome is a vulnerability codebook $\mathcal{C}$ comprising 150 vulnerability centroids $\mathbf{c}$ representing common hidden patterns. To leverage the potential of this codebook, we propose a vulnerability-matching methodology. We match a testing function with each centroid in our learned codebook to predict vulnerabilities at the function and line levels. The matching concept is inspired by static analysis tools that match human-defined patterns to identify vulnerabilities, while we match patterns learned in the feature space to detect vulnerabilities using deep learning models.

Below, we introduce background knowledge of optimal transport (OT) and vector quantization (VQ), and the motivations for using OT and VQ, followed by a discussion of related works.

A. Optimal Transport

Optimal transport theory, also known as transportation theory or the theory of mass transportation, is a mathematical framework that deals with the optimal ways to transport objects from one place to another. Originally developed by Gaspard Monge [20], OT theory has found applications in various fields, including economics, physics, computer science, and machine learning [21].

Optimal transport (OT) has been widely applied across machine learning tasks, particularly in clustering and repre-

sentation learning [22–25]. For example, Laclau *et al.* [22] used OT for co-clustering to uncover structured relationships between data instances and features. Del *et al.* [23] employed OT to robustly cluster probability distributions via Wasserstein barycenters. Yan *et al.* [24] unified subspace and spectral clustering through OT to learn geometry-aware embeddings, while Liu *et al.*[25] aligned clustering centers across domains to improve efficiency and robustness in unsupervised domain adaptation.

Inspired by these applications, we adopt OT in the vulnerability detection domain to cluster vulnerable code representations. This enables the identification of common vulnerability patterns and the condensation of high-dimensional vectors into a compact codebook of vulnerability centroids, facilitating efficient vulnerability matching. Specifically, we rely on the Wasserstein distance, a key component of optimal transport, to measure the similarity between vulnerability vectors and centroids during clustering.

Motivation. The Wasserstein distance [12] is used for clustering vulnerability vectors, as it captures both the global structure across the dataset and local differences within individual instances. This helps preserve important relationships between vectors and centroids. By minimizing this distance during clustering, we obtain representative centroids that reflect common vulnerability patterns, improving the overall effectiveness in our vulnerability matching process.

B. Vector Quantization

Vector quantization (VQ) is a technique used to reduce the dimensionality of data by representing a large set of vectors with a smaller set of prototype vectors, often referred to as centroids. Each vector in the original set is assigned to the nearest prototype vector, effectively quantizing the original data into a compressed representation.

In particular, prior works leverage VQ for efficient data compression and discrete representation learning. For example, Lu *et al.* [26] proposed VQNet, a VQ network integrated into a deep encoder-decoder for image compression. Chen *et al.* [27] introduced a learning VQ method that compresses knowledge into reference vectors to handle incremental few-shot learning while reducing forgetting. Van *et al.* [10] presented VQ-VAE, which uses VQ to learn discrete latent codes, enabling high-quality generative modeling.

Drawing on VQ's knowledge compression capability, we apply VQ to the domain of vulnerability detection, aiming to compress large sets of vulnerability feature vectors into a compact set of representative centroids. This approach facilitates efficient grouping and matching of similar vulnerability patterns, reducing complexity while preserving essential information. Below, we detail our method for integrating vector quantization into the vulnerability detection pipeline. Below, we introduce the motivation for why VQ is required in our framework.

Motivation. While the optimal transport learns to transfer vulnerability vectors to centroids, we still need to determine how we assign each vulnerability vector to its corresponding centroid during training. Inspired by the VQ-VAE approach in the computer vision domain [10], we leverage the VQ principle to address the centroid assignment for each vulnerability vector. In particular, each vulnerability vector will be assigned to its representative centroid based on the Euclidean distance, allowing us to effectively group similar vulnerability vectors during training.

C. Related Work

1) Deep Learning-based Vulnerability Detection: Prior works proposed various deep learning-based vulnerability detections (VDs) such as convolutional neural networks (CNNs) [28], recurrent neural networks (RNNs) [29–32], graph neural networks (GNNs) [6, 8, 18, 33–35], and pre-trained transformers [7, 8, 36, 37]. RNN-based methods [28, 29, 38] have been shown more accurate than program analysis tools such as Checkmarx [14] and RATS [39] to predict function-level vulnerabilities.

However, RNNs face difficulty in capturing long-term dependencies in long sequences as the model's sequential nature may result in the loss of earlier sequence information. Furthermore, function-level predictions lack the required granularity to accurately locate vulnerable lines. Thus, prior works proposed transformer-based methods that predict line-level vulnerabilities and capture long-term dependencies [7, 8].

Ding *et al.*[8] propose an ensemble approach that uses a transformer model to capture global features and a GNN to capture local features while Fu *et al.*[7] leverage a pre-trained transformer model and interpret its attention scores as line-level predictions. In addition, Nguyen *et al.*[40] leverages bidirectional RNNs with information theory to detect line-level vulnerabilities.

On the other hand, Zhou *et al.*[33] embed the abstract syntax tree (AST), control flow graph (CFG), and data flow graph (DFG) for a code function and learn the graph representations for function-level predictions. Nguyen *et al.*[35] proposed constructing a code graph as a flat sequence for function-level predictions. Hin *et al.*[6] constructed program dependency graphs (PDGs) for functions and predicted line-level vulnerabilities.

Recent works have also explored deep learning-based vulnerability detection in domain-specific contexts. For instance, Zhang *et al.* [41] focused on injection vulnerabilities in Java web applications by integrating interprocedural program analysis with a BERT-BiLSTM-CRF model, framing vulnerability detection as a sequence labeling task. In the domain of smart contracts, Huang *et al.* [42] proposed an interpretable detection framework based on a Graph Isomorphism Network (GIN), enhanced with domain-specific features and subgraph-level explanations. Similarly, Chu *et al.* [43] introduced DeepFusion, which combines program slicing and AST-based structural features with a BiLSTM+Attention model for vulnerability detection in smart contracts.

While previous studies focus on varying model architectures such as RNNs, GNNs, and transformers, this paper emphasizes a distinct avenue. We noticed that vulnerability scope embeddings tend to cluster within the feature space. Despite the potential enhancement this characteristic could offer to VD models, prior works have overlooked this critical

aspect. In response to this gap, we employ the principle of vector quantization (VQ) with optimal transport (OT) to exploit the clustering characteristics and hidden patterns inherent in vulnerability scope embeddings. Thus, this paper represents a pioneering step in learning and matching vulnerability patterns using OT and VQ.

2) Large Language Models for Vulnerability Detection: Recent works have explored the use of large language models (LLMs) for vulnerability detection (VD) [44, 45]. Fu *et al.*[45] evaluated the performance of zero-shot ChatGPT for tasks such as vulnerability detection, classification, and repair. Steenhoek *et al.*[44] fine-tuned LLMs such as UniXcoder [46] for VD. Shu *et al.*[47] evaluated the performance of LLMs with instruction tuning for VD under a multilingual scenario. It is worth noting that LLMs like ChatGPT [48], BARD [49], and CodeX [50] have demonstrated their capabilities in source code-related tasks, such as code generation. However, these LLMs were not originally designed for software security applications. Additionally, the architectures of ChatGPT, BARD, and CodeX are not fully open-sourced. While our proposed framework could theoretically be applied to these LLMs, the lack of access to modify the model architecture makes it challenging for us to implement our framework effectively for these advanced LLMs.

Our study distinguishes itself from prior works that primarily concentrate on model architecture or the use of large language models (LLMs). Instead, this paper aims to bridge the gap between the clustering characteristics of vulnerability scope embeddings observed in the training data and vulnerability detection (VD) models. Since large language models (LLMs), typically consisting of billions of parameters, are not the primary focus of this paper, we deliberately selected base-size language models with 125M-225M parameters. This decision was made to enhance the accessibility of our approach and research, as well as to facilitate future reproduction. By opting for base-size language models, we aim to reduce resource consumption and lower the barrier of computational requirements.

III. Approach

In this section, we first present the formal problem statement, followed by the technical details of our DeepVulMatch approach. For clarity and ease of reference, Table I summarizes all notations used in this section. Additionally, Figure 4 provides a high-level overview of our approach during both training and inference.

A. Problem Statement

Let us consider a dataset of N functions in the form of source code. The data set includes both vulnerable and benign functions, where the function-level and line-level ground truths have been labeled by security experts. We denote a function as a set of code lines, $X_i = [\mathbf{x}_1, ..., \mathbf{x}_n]$, where n is the max number of lines we consider in a function. Let a sample of data be $\{(X_i, y_i, \mathbf{z}_i) : X_i \in \mathcal{X}, y_i \in \mathcal{Y}, \mathbf{z}_i \in \mathcal{Z}, i \in \{1, 2, ..., N\}\}$, where $\mathcal{X}$ denotes a set of code functions, $\mathcal{Y} = \{0, 1\}$ with 1 represents vulnerable function and 0 otherwise, and $\mathcal{Z} = \{0, 1\}^n$ denotes a set of binary vectors with 1 represents vulnerable code line and 0 otherwise. Our objective is to identify the vulnerability on both *function and line levels*. It should be emphasized that our experimental datasets label the information of vulnerable code lines, $\mathcal{Z}$. However, it is crucial to note that we only rely on this information during supervised training, while *such information is not required by our method during the validation and testing phases.*

TABLE I
Notations used in this paper.

Notation	Description
N	Total number of samples in our dataset
n	Max number of lines in a function
r	Max number of tokens in a line
q	Max number of vulnerable lines in a function
a	Total number of vulnerable functions in training set
X	Code function
X^{vul}	Vulnerable lines in a function
$\mathbf{x}$	Code line
$\mathbf{t}$	Code token
y	Function label
$\mathbf{z}$	Line label
$\mathbf{S}$	Code line embedding
$\mathbf{P}$	Vulnerable line embedding
$\mathbf{P}_{benign}$	Special benign line embedding
$\mathcal{V}_v$	Vulnerability collection
$\mathcal{V}$	Vulnerability collection with special benign vectors
$\mathbf{v}$	Vulnerability vector
$\mathbf{v}_b$	Special benign vector
$\mathcal{C}$	Vulnerability codebook
$\mathbf{c}$	Vulnerability centroid
$\mathbf{c}^*_{\mathbf{v}}$	Closest centroid to a vulnerability vector
E	Token embedding layer
RNN_{line}	Line embedding RNN
RNN_{vul}	Vulnerability summarization RNN
RNN_{func}	RNN used to summarize the output of encoders
$\mathcal{F}$	Transformer encoders
$\mathbf{H}$	Input to transformer encoders
W^I	Linear projection for line-level prediction
W^J	Linear projection for function-level prediction
$\hat{y}$	Function prediction
$\hat{\mathbf{z}}$	Line prediction
$\mathcal{L}_f$	Function cross-entropy loss
$\mathcal{L}_s$	Line cross-entropy loss
W_d	Wasserstein distance

We formulate function-level vulnerability detection (VD) as a binary classification and line-level VD as a multi-label classification problem. Given X, we use an RNN denoted as RNN_{line} to obtain $\mathbf{S} \in \mathbb{R}^{n \times d}$, the d-dimensional line embeddings for X. Let us denote X^{vul} as a set of all vulnerable lines in a vulnerable function. We extract X^{vul} from a vulnerable function and embed those vulnerable lines as $\mathbf{P} \in \mathbb{R}^{q \times d}$, where q is the maximum number of vulnerable lines. We use the same RNN_{line} to embed X^{vul}.

Note that for a benign function with no vulnerable lines, we use a special learnable embedding denoted as $\mathbf{P}_{benign} \in \mathbb{R}^{q \times d}$. We transform $\mathbf{P}$ into a flat vulnerability vector (denoted as $\mathbf{v} \in \mathbb{R}^d$) using an RNN denoted as RNN_{vul}. Additionally, we transform $\mathbf{P}_{benign}$ into a flat vector denoted as $\mathbf{v}_b$ for a benign function. We collect all $\mathbf{v}$ to form a vulnerability collection $\mathcal{V}_v = [\mathbf{v}_1, ..., \mathbf{v}_a]$ where a is the total number of vulnerable functions in our training set. We obtain around 6,361

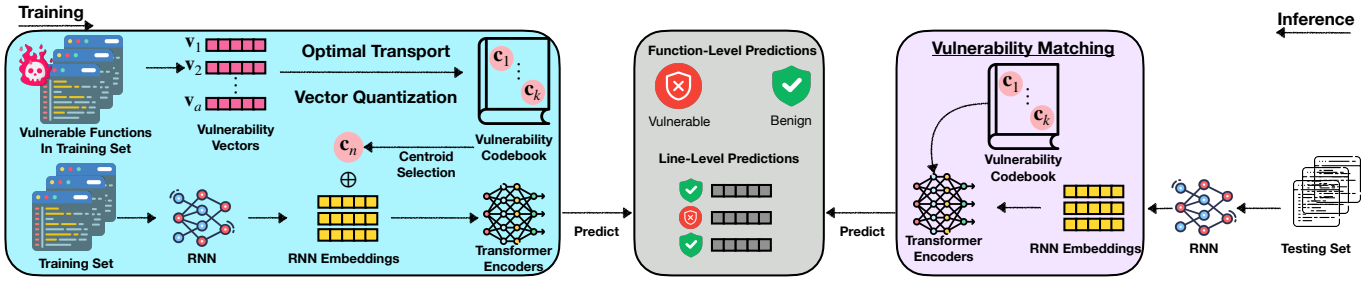

Fig. 4. Overview of our DEEPVULMATCH approach. During training, vulnerability vectors from vulnerable functions are aggregated into centroids using optimal transport and vector quantization, forming a vulnerability codebook. One centroid is selected for each input and combined with RNN embeddings and fed into the model. During inference, RNN embeddings and the codebook are used for vulnerability matching, enabling both function-level and line-level vulnerability detection.

vulnerability vectors from our training set. Handling such an extensive collection size ($\mathcal{V}_v$) will require significant computing resources during our vulnerability matching inference, as each testing sample needs to undergo 6,361 matches. Thus, we condense $\mathcal{V}_v$ using optimal transport (OT) to cluster vulnerability centroids, denoted as $\mathbf{c}$. Each $\mathbf{c}$ represents a set of similar $\mathbf{v}$. Subsequently, we construct our vulnerability codebook $\mathcal{C} = [\mathbf{c_1}, ...\mathbf{c_k}]$, with k representing the number of vulnerability centroids.

Let us denote a stack of transformer encoders as $\mathcal{F}$. In the warm-up training, we input $\mathbf{H} = \mathbf{S} \oplus \mathbf{v} \in \mathbb{R}^{(n+1)\times d}$ into $\mathcal{F}$. We then make function and line-level predictions based on the output of $\mathcal{F}$.

Below, we introduce each component in our DEEPVULMATCH approach. To support both function- and line-level vulnerability detection, we begin by encoding each function using an RNN-based line embedding module. Unlike conventional token-level embeddings, this component captures contextual relationships across lines of code. We describe this module below.

B. Line Embedding Using RNN

Prior research has used subword tokenization techniques to segment code functions [6, 7, 36, 37]. In these studies, the approach involved concatenating individual code lines within a function and then transforming them into a sequence of subword tokens using the byte pair encoding (BPE) algorithm [51]. However, many base-size models, such as CodeBERT-base, face a limitation in processing only up to 512 token embeddings. Consequently, this limitation has the potential to result in information loss, particularly for longer functions that exceed the 512-token threshold.

To address this limitation, our embedding transforms each code line into a sequence of subword tokens, without concatenating the lines. Each code line is represented as a set of token embeddings. Subsequently, we utilize a shared RNN to summarize these token embeddings within each line, thus creating a line embedding and representing each code line as a vector.

Given $X = [\mathbf{x}_1, ..., \mathbf{x}_n]$, we use BPE to tokenize $\mathbf{x}$ to a list of tokens, $[\mathbf{t}_1, ..., \mathbf{t}_r]$, where r is the number of tokens we consider in a code line. We embed each $\mathbf{t}$ using a token embedding layer $E \in \mathbb{R}^{v\times d}$ where v is the vocab size. This results in a d-dimensional token embedding, denoted as $\bar{\mathbf{S}}_\mathbf{i} \in \mathbb{R}^{n\times r\times d}$. We then input token embeddings into an RNN and get line embeddings $\mathbf{S}$, which can be summarized as $RNN_{line}(E(\mathbf{x}_1), ..., E(\mathbf{x}_n))$ where $E(\mathbf{x}_i) \in \mathbb{R}^{r\times d}$. We use the identical line embedding method to embed X^{vul} as $\mathbf{P}$.

It is worth noting that our line embedding methods can process up to 3,100 tokens per input function which is six times more than 512 tokens that can be accepted by common source code language models such as CodeBERT-base [36]. Table II shows that our line embedding method results in more than 30% enhancements in the performance of the CodeBERT and CodeGPT models for line-level predictions.

Our approach includes a warm-up phase (Figure 5) followed by a main training phase (Figure 6), both of which use the same RNN embedding module to encode functions. In the warm-up phase, we collect vulnerability vectors $\mathbf{v}$ to build a vulnerability collection $\mathcal{V}_v$ and fine-tune the model for better line-level representations. The main training phase introduces our core idea—quantizing $\mathbf{v}$ into vulnerability centroids $\mathbf{c}$, as illustrated in Figure 3. Below, we describe the technical details of the warm-up phase.

C. Training of Warm-Up Phase

For a vulnerable function, we concatenate line embeddings $\mathbf{S}$ with a vulnerability vector $\mathbf{v}$ as $\mathbf{H}$ and input it to $\mathcal{F}$. For a benign function, we input $\mathbf{H} = \mathbf{S} \oplus \mathbf{v}_b$ to $\mathcal{F}$. This input consists of n code line embeddings with an additional vulnerability vector. We select the n code line embeddings as $\mathbf{H}[1:n]$. We predict line-level vulnerabilities with $\mathbf{H}[1:n]W^I$, where $W^I \in \mathbb{R}^{d\times 1}$. We apply an RNN denoted as RNN_{func} to summarize line embeddings into a vector to predict function-level vulnerabilities with $RNN_{func}(\mathbf{H}[1:n])W^J$, where $W^J \in \mathbb{R}^{d\times 2}$. We minimize the training objective as follows:

$$\frac{1}{N}\sum_{i=1}^{N}\Big[\mathcal{L}_f\Big(RNN\big(\mathcal{F}(\mathbf{S},\mathbf{v})\big), y\Big) + \mathcal{L}_s\Big(\mathcal{F}(\mathbf{S},\mathbf{v}), \mathbf{z}\Big)\Big] \quad (1)$$

where $\mathcal{L}_f$ and $\mathcal{L}_s$ are cross-entropy over function and line labels respectively.

Having completed warm-up training with function and line-level supervision, we now move to the main training phase, initializing from the warmed-up model. This phase incorporates optimal transport (OT) and vector quantization (VQ) to enhance vulnerability pattern matching. It consists

of three key components: (1) collecting vulnerability vectors from training functions, (2) clustering them into representative centroids using OT and VQ to form a vulnerability codebook that captures latent vulnerability patterns, and (3) jointly updating the model parameters for both the OT/VQ modules and the vulnerability detection task, using supervision from function and line-level labels. The learned codebook serves as a condensed set of latent vulnerability patterns, enabling the model to match and detect vulnerabilities across diverse code functions.

D. Quantizing Vulnerability Vectors: Optimal Transport and Subsequent Main Training Phase

In this section, we introduce how we construct vulnerability collection $\mathcal{V}_v$. Then we describe how we transform $\mathcal{V}_v$ into a vulnerability codebook $\mathcal{C}$ to reduce the collection size and facilitate more efficient vulnerability matching. Finally, we introduce how we use $\mathcal{C}$ in our main training phase.

1) Collect Vulnerability Vectors from Vulnerable Functions: We extract vulnerable lines X^{vul} in the training set and use RNN_{line} to embed X^{vul} as $\mathbf{P}$. We summarize each $\mathbf{P}$ into a vulnerability vector $\mathbf{v}$ using RNN_{vul}. We collect a total of 6,361 $\mathbf{v}$ from all of vulnerable X in our training set to form our vulnerability collection $\mathcal{V}_v \in \mathbb{R}^{6,361\times h}$. We reduce the d-dimensional $\mathbf{v}$ to h-dimensional to facilitate the upcoming clustering. The large collection size of $\mathcal{V}_v$ will demand substantial computing resources during inference because we need to match each function with 6,361 vulnerability vectors. Therefore, we use optimal transport (OT) to quantize similar vulnerability vectors that share the same vulnerability patterns into vulnerability centroids. In what follows, we outline the process of OT that effectively reduces 6,361 vulnerability vectors to 150 centroids.

2) Learn to Transport Vulnerability Vectors to Vulnerability Centroids: As depicted in the left part of Figure 6, our goal is to quantize $\mathcal{V}_v$ to a vulnerability codebook, $\mathcal{C} = [\mathbf{c}_1, ..., \mathbf{c}_k]$. This codebook consists of a more compact set of vulnerability patterns. To ensure that each $\mathbf{c}$ can represent a group of similar $\mathbf{v}$, we leverage the optimal transport (OT) theory to transfer sets of $\mathbf{v}$ to their corresponding $\mathbf{c}$.

We randomly initialize the embedding space of our vulnerability codebook as $\mathcal{C} = [\mathbf{c}_1, ..., \mathbf{c}_k]$ with the k number of clusters. We minimize the Wasserstein distance [12] using the Sinkhorn approximation [52] between $\mathcal{V}_v$ and $\mathcal{C}$. Consequently, $\mathbf{v}$ and their respective $\mathbf{c}$ will converge toward each other as shown in Figure 3. Our codebook will ultimately comprise vulnerability centroids acting as representative patterns that symbolize different sets of vulnerability vectors. This allows us to aggregate similar vulnerability scopes into patterns based on the Euclidean distance. We summarize the process as follows:

$$min_{\mathcal{C}}\ W_d := W_d(\mathbb{P}_{\mathcal{V}_v}, \mathbb{P}_{\mathcal{C}}) \tag{2}$$

where $d(\mathbf{v}, \mathbf{c}) = \|\mathbf{v} - \mathbf{c}\|_2$ represents Euclidean distance, W_d is the Wasserstein distance [12], $\mathbb{P}_{\mathcal{V}_v} = \frac{1}{a}\sum_{i=1}^{a}\delta_{\mathbf{v}_i}$, $\mathbb{P}_{\mathcal{C}} = \frac{1}{a}\sum_{j=1}^{a}$, and $\delta_{\mathbf{c}_j}$ represents the Dirac delta distribution. According to the clustering view of optimal transport [53, 54], when minimizing $min_C\ W_d(\mathbb{P}_{\mathcal{V}_v}, \mathbb{P}_{\mathcal{C}})$, $\mathcal{C}$ will become the centroids of the clusters formed by $\mathcal{V}_v$. This clustering approach ensures that similar vulnerability scopes sharing the same vulnerability pattern are grouped, leading to a quantized vulnerability codebook involving common vulnerability patterns. Next, we introduce how we utilize our vulnerability codebook in the main training phase.

Algorithm 1 Training Process of DEEPVULMATCH

Input: $\mathcal{X}$, $\tilde{\mathcal{Y}}$, $\tilde{\mathcal{Z}}$, is_warm_up
Sample mini-batch of $\tilde{\mathcal{X}} = \tilde{\mathcal{X}}_b \cup \tilde{\mathcal{X}}_v$
Compute $\tilde{\mathcal{V}} = \tilde{\mathcal{V}}_b \cup \tilde{\mathcal{V}}_v$ and $\tilde{\mathcal{S}}$, $\tilde{\mathcal{S}}$ is a mini-batch of $\mathbf{S}$
if is_warm_up **then**
  Minimize objective function in (1) w.r.t. $\tilde{\mathcal{S}}$ and $\tilde{\mathcal{V}}$
else
  $W_d = \text{Sinkhorn}(\tilde{\mathcal{V}}_v, \mathcal{C})$
  for $\mathbf{v} \in \tilde{\mathcal{V}}_v$ **do**
    $\mathbf{c}^*_{\mathbf{v}} = \text{argmin}_{\mathbf{c}\in\mathcal{C}}\|\mathbf{v} - \mathbf{c}\|_2$
    Input $= \mathbf{S} \oplus \mathbf{c}^*_{\mathbf{v}}$ to $\mathcal{F}$
  end for
  for $\mathbf{v} \in \tilde{\mathcal{V}}_b$ **do**
    $\mathbf{c}^*_{\mathbf{v}} = \mathbf{v}_b$
    Input $= \mathbf{S} \oplus \mathbf{c}^*_{\mathbf{v}}$ to $\mathcal{F}$
  end for
  Minimize objective function in (4)
end if

3) Main Training Phase: The right part of Figure 6 summarizes our main training phase. We load the model parameters from the warm-up phase. We obtain the line embeddings $\mathbf{S}$ and a vulnerability vector $\mathbf{v}$ for the input function X as introduced in Section III-B. We employ a cluster selection process inspired by VQ-VAE [10], utilizing the Euclidean distance to map $\mathbf{v}$ to its most similar centroid denoted as $\mathbf{c}^*_{\mathbf{v}}$ selected from our codebook $\mathcal{C}$:

$$\mathbf{c}^*_{\mathbf{v}} = \text{argmin}_{\mathbf{c}\in\mathcal{C}}\|\mathbf{v} - \mathbf{c}\|_2 \tag{3}$$

Note that for a benign function, we directly assign $\mathbf{c}^*_{\mathbf{v}}$ as $\mathbf{v}_b$. We then concatenate $\mathbf{S}$ with $\mathbf{c}^*_{\mathbf{v}}$ and input $\mathbf{H} = \mathbf{S} \oplus \mathbf{c}^*_{\mathbf{v}}$ to $\mathcal{F}$. The subsequent forward passes are the same as our warm-up phase. We minimize the loss function as follows:

$$\frac{1}{N}\sum_{i=1}^{N}\Big[W_d + \mathcal{L}_f\Big(RNN\big(\mathcal{F}(\mathbf{S}, \mathbf{c}^*_{\mathbf{v}})\big), y\Big) + \mathcal{L}_s\Big(\mathcal{F}(\mathbf{S}, \mathbf{c}^*_{\mathbf{v}}), \mathbf{z}\Big)\Big] \tag{4}$$

Note that no real gradient is defined for $\mathbf{v}$ once we map it to a $\mathbf{c}^*_{\mathbf{v}}$ via the argmin operation in Equation 3. To let the RNNs that embed and summarize vulnerable lines be trainable via backpropagation, we follow the idea in VQ-VAE [10] which was shown effective for vector quantization. We copy gradients from $\mathbf{v}$ to $\mathbf{c}^*_{\mathbf{v}}$. Below, we present how to use our learned codebook for vulnerability matching during inference.

E. Vulnerability Detection Through Explicit Vulnerability Patterns Matching

Similar to static analysis tools that identify vulnerabilities by matching predefined patterns, we match our pre-learned vulnerability codebook to detect vulnerabilities using deep learning models.

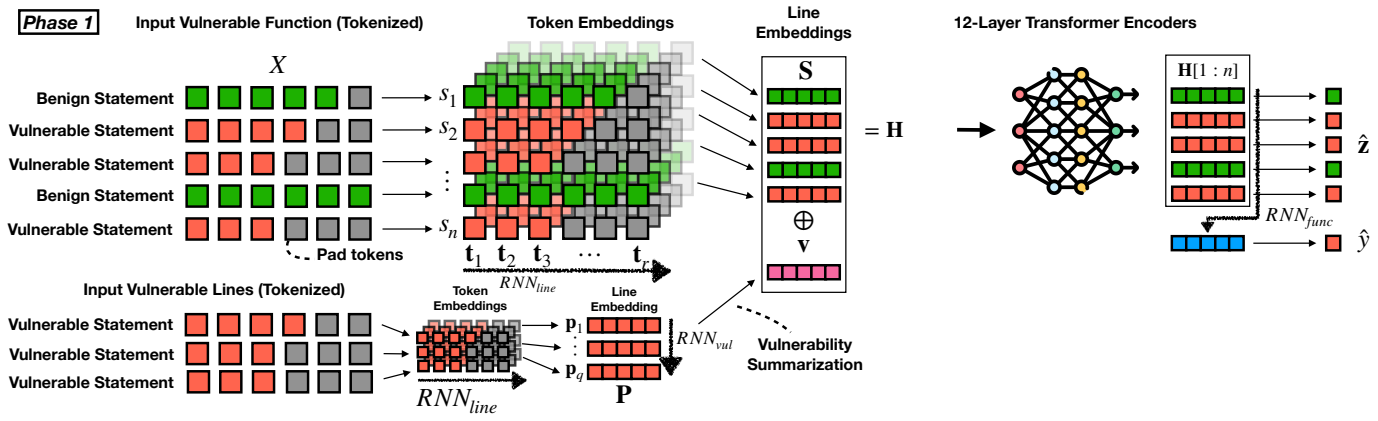

Fig. 5. The overview of the warm-up phase in our approach. We tokenize each line in a vulnerable function (i.e., X) followed by an embedding layer to map each token into a vector. We use RNN_{line} to summarize the token embeddings and get the line embedding ($\mathbf{S}$, $\mathbf{P}$). For benign functions, $\mathbf{P}$ is replaced by a special learnable embedding, $\mathbf{P}_{benign}$. We use RNN_{vul} to summarize the embeddings of vulnerable lines $\mathbf{P}$ in a vector $\mathbf{v}$ that represents the vulnerability scope. We concatenate $\mathbf{S}$ and $\mathbf{v}$ as the input to transformer encoders to consider vulnerability scopes that arise in the function and align with our vulnerability matching process. We select the line embeddings output from the last encoder, i.e., $\mathbf{H}[1:n]$. Each line embedding vector is mapped to a probability as line-level predictions, the function-level prediction is obtained by summarising $\mathbf{H}[1:n]$ to a vector using an RNN_{func} and mapping it to a probability.

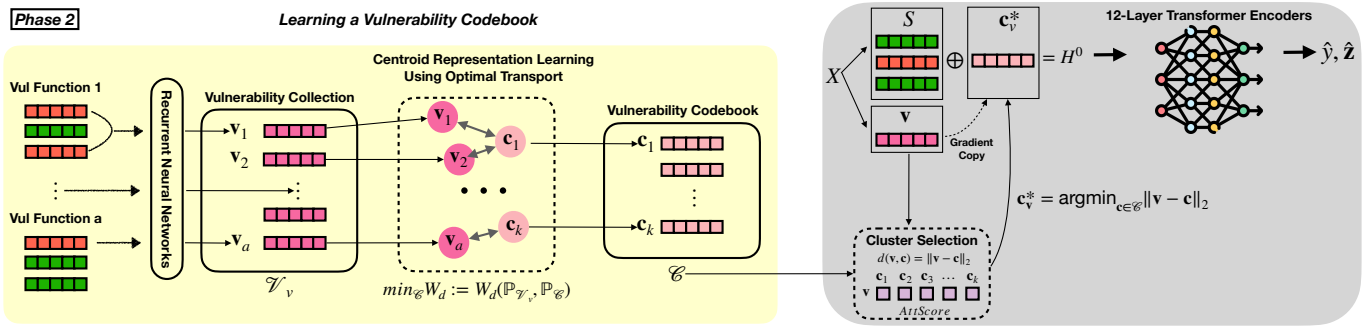

Fig. 6. The overview of the main training phase in our approach. We introduce how to learn our vulnerability codebook on the left. We first collect a set of vulnerable line embeddings from our training data. We then use RNN_{vul} to pool a set of line embeddings from each vulnerable function, forming a vulnerability vector $\mathbf{v}$. The set of these scopes forms our vulnerability collection $\mathcal{V}_v = \{\mathbf{v}_1, \ldots, \mathbf{v}_a\}$. Next, we learn vulnerability centroids $\mathbf{c}$ using the Wasserstein distance metric to create a more compact vulnerability codebook $\mathcal{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_k\}$, where each centroid represents a group of $\mathbf{v}$. During training, we minimize the Wasserstein distance between each $\mathbf{v}$ and its corresponding centroid $\mathbf{c}^*_\mathbf{v}$. As shown on the right, we input $\mathbf{H} = \mathbf{S} \oplus \mathbf{c}^*_\mathbf{v}$ to $\mathcal{F}$. To overcome the non-differentiability of the argmin operation in the networks, we copy the gradients from $\mathbf{v}$ to $\mathbf{c}^*_\mathbf{v}$ to learn RNN_{line} and RNN_{vul}.

For each testing sample X, we obtain d-dimensional line embeddings $\mathbf{S}$. We input $\mathbf{H} = \mathbf{S} \oplus \mathbf{c}$ to $\mathcal{F}$. We select the n code line embeddings as $\mathbf{H}[1:n]$. We obtain function and line-level predictions based on the output of $\mathcal{F}$. We apply RNN_{func} to summarize line embeddings into a vector to predict function-level vulnerabilities with $RNN_{func}(\mathbf{H}[1:n])W^J$, where $W^J \in \mathbb{R}^{d\times 2}$. We predict line-level vulnerabilities with $\mathbf{H}[1:n]W^I$, where $W^I \in \mathbb{R}^{d\times 1}$. This process is iterated for $k = 150$ times to match all of the vulnerability centroids $\mathbf{c}$ in our codebook $\mathcal{C}$.

To aggregate k number of predictions produced by our matching process, apply max and mean pooling. We use max pooling to select the most prominent matching result and then use argmax to obtain the function-level prediction $\hat{y}$. If X is predicted as a benign function, we directly output a zero vector as the line-level prediction. Otherwise, we use mean pooling to aggregate the line-level predictions into a flat probability vector. We apply a probability threshold of 0.5 to transform this probability vector into a binary vector as the final line-level prediction $\hat{z}$.

IV. Experimental Design and Results

A. Research Questions

One of the key goals of this paper is to evaluate our DeepVulMatch thoroughly by comparing it with other baseline approaches that focus on the function-level and line-level vulnerability detection tasks. We also formulate an ablation study to support the design decision of our DeepVulMatch approach. Below, we present the motivation for our two research questions:

(RQ1) What is the accuracy of our DeepVulMatch for predicting function-level and line-level vulnerabilities? Recent studies have leveraged pre-trained language models with transformer architectures to perform vulnerability detection effectively [7, 8, 44]. We found that different vulnerable functions may form similar vulnerability patterns, as demonstrated in Figure 1. This discovery suggests valuable information that could enhance the performance of deep learning (DL) models for locating line-level vulnerabilities. However, the use of vulnerability patterns in DL models remains unexplored. To

address this gap, we propose DEEPVULMATCH, an innovative framework that harnesses optimal transport (OT) and vector quantization (VQ) techniques to learn and match vulnerability patterns. This approach could facilitate the precise identification of vulnerabilities at both function and line levels. Consequently, we formulate this RQ to evaluate the accuracy of our proposed framework in function and line-level vulnerability detection.

(RQ2) What are the contributions of each component in our DEEPVULMATCH approach? Our DEEPVULMATCH framework incorporates two crucial components: (i) code line embeddings using RNNs, and (ii) optimal transport (OT) and vector quantization (VQ). However, the specific contributions of these components remain unexplored. We thus formulate this research question as an ablation study to assess the components in our DEEPVULMATCH framework and empirically validate our design decisions.

B. Baseline Approaches

We compare our DEEPVULMATCH approach with language models pre-trained on source code data, state-of-the-art transformer-based, GNN-based, RNN-based, and CNN-based vulnerability detection (VD) approaches proposed by prior studies. For all of the pre-trained transformer baselines, we use the standard size model (e.g., CodeT5-base, CodeBERT-base, etc.) to ensure a fair comparison where each transformer baseline has a similar number of parameters to our approach. We reproduce twelve baselines in total, where each baseline is fine-tuned on our studied dataset based on the code and hyperparameters provided by the original authors.

Below, we describe each selected baseline along with the justification for its inclusion.

1) Language Models for code: : We include five language model baselines commonly adopted for source code-related tasks: **CodeT5P-220m** [55], **CodeT5-base** [56], **CodeBERT-base** [36], **CodeGPT** [57], and **GraphCodeBERT-base** [37]. Our DEEPVULMATCH is built on CodeT5-base, a model with approximately 220M parameters. To ensure a fair comparison, we selected language model baselines that are pre-trained specifically for source code and have a comparable model size, typically ranging from 120M to 220M parameters. Models of similar size that are not pre-trained on code are excluded, as our vulnerability detection task requires strong code understanding capabilities.

2) Transformer-based VD: : We include two transformer-based baselines as follows:

- **LineVul** [7] is designed to perform function-level prediction by leveraging a pre-trained transformer model. Although it can also provide line-level predictions by interpreting and ranking the attention scores of the transformer, this approach is not suitable for the line-level classification setting. To ensure a fair comparison, we only evaluate our approach against LineVul on the function level.
- **VELVET** [8] is an ensemble method that leverages a vanilla transformer with GNNs.

Transformer-based VD models have shown superior performance compared to RNN-based approaches such as Russell *et al.* [28], SySeVR [38], and VulDeePecker [29] on the same dataset used in our study, Big-Vul [7]. Thus, we exclude RNN-based models from our baseline comparison in favor of more advanced transformer-based methods.

3) GNN-based VD: : We include three graph-based baselines as follows (note that ReGVD and Devign only predict function-level vulnerabilities):

- **LineVD** [6] leverages pre-trained CodeBERT embeddings with GNNs to detect line-level vulnerabilities.
- **ReGVD** [35] represents a code function as a sequential graph and uses GNNs to detect function-level vulnerabilities.
- **Devign** [33] leverages code property graph (CPG) [58] with GNNs to detect function-level vulnerabilities.

We include three graph-based baselines to represent a distinct class of vulnerability detection approaches that leverage structural code representations, which are fundamentally different from RNN- and transformer-based models. These methods incorporate code properties such as abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs) to model the semantics and structure of programs. In particular, LineVD [6] aligns with our task by performing line-level detection, while ReGVD [35] and Devign [33] are widely adopted function-level baselines that capture code semantics through graph-based reasoning.

4) Information Theory-based and CNN-based VD: : We include one information theory-based and one CNN-based baseline as follows:

- **ICVH** [40] is an information theory-based approach for line-level vulnerability detection, originally designed for unsupervised learning. To enable a fair comparison in our supervised setting, we adapted the original ICVH architecture by incorporating a cross-entropy loss for training.
- **TextCNN** [59] uses convolutional layers for sentence classification tasks.

We include the information theory-based ICVH [40] as it offers a conceptually different approach to our line-level vulnerability detection by modeling the information gain of vulnerable code lines. We also include TextCNN [59] as a primitive CNN-based baseline. Originally designed for sentence classification, its architecture aligns with our line-level detection task, which has not been previously evaluated for vulnerability detection at the line level.

C. Experimental Datasets

It is important to highlight that common vulnerability datasets such as Devign [33] and DiverseVul [60] are not included in this study due to the absence of line-level vulnerability labels. To identify vulnerabilities on function and line levels, we select the Big-Vul dataset [16] and the D2A dataset [17] as they are two of the largest vulnerability data sets with line-level vulnerability labels and has been used to assess line-level vulnerability detection methods [6, 7]. Big-Vul was collected from 348 Github projects and consists of 188k C/C++ functions with 3,754 code vulnerabilities spanning 91 vulnerability types. The data distribution of Big-Vul resembles

real-world scenarios, where the proportion of vulnerable to benign functions is 1:20. In contrast, D2A comprises around 6.5k samples, with an approximate 1:1 ratio of vulnerabilities, all of which were extracted from real-world projects.

D. Parameter Settings and Model Training

We split the dataset into 80% for training, 10% for validation, and 10% for testing. For both our approach and the baselines, we consider functions with up to n = 155 lines and r = 20 tokens per line, based on descriptive statistics indicating that 95% of source code functions contain fewer than 155 lines, and 95% of lines contain fewer than 20 tokens. We initialize our Transformer encoders using CodeT5-base [56], which has an embedding dimension of 768. For both training phases 1 and 2, we use the AdamW optimizer [61] with a learning rate of 1×10^{-4} and a maximum gradient norm of 1.0. The model is trained for 20 epochs in each phase with a training batch size of 64, and we select the checkpoint that achieves the highest F1 score on line-level prediction in the validation set. All experiments were conducted on a Linux machine equipped with an AMD Ryzen 9 5950X processor, 64 GB of RAM, and an NVIDIA RTX 3090 GPU. Complete details of the hyperparameter settings for all baselines are provided in our replication package at https://github.com/awsm-research/DeepVulMatch.

(RQ1) What is the accuracy of our DEEPVULMATCH *for predicting function-level and line-level vulnerabilities?*

Approach. We conduct experiments on the Big-Vul [16] and D2A [17] datasets described in Section IV-C and compare our DEEPVULMATCH methods with 12 other baselines described in Section IV-B. For both function and line-level vulnerability prediction, we report: Precision (Pre) $= \frac{\text{True Positives}}{\text{True Positives}+\text{False Positives}}$; Recall (Re) $= \frac{\text{True Positives}}{\text{True Positives}+\text{False Negatives}}$; and $F1 = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$. These measures enable us to assess the models' performance on both positive and negative classes, regardless of the class imbalance between vulnerable and benign functions. It is important to note that the line-level metrics are computed on the line level instead of the function level to determine if each line is correctly predicted. Furthermore, we conduct an extra trial for four baselines: CodeT5P-220m, CodeT5-base, CodeBERT-base, and CodeGPT. We use our RNN line embedding method instead of their token embedding to explore the potential enhancement of their performance. Note that our line embedding is not compatible with GraphCodeBERT's data flow construction.

To ensure the robustness of our results, we repeat all experiments on the Big-Vul dataset five times using different random seeds and report the median $\pm$ standard deviation of each method's performance. We apply the Wilcoxon signed-rank test and compute Cohen's d as an effect size metric. Specifically, we compare our approach against the best-performing baseline in terms of F1 score at both the function and line levels.

Result. Table II presents the experimental results of our DEEPVULMATCH approach and 12 other baseline approaches according to the function and line-level precision, recall, and F1 score. **Our approach achieves an line-level F1 score of 82% which reveals a 32% improvement over the best baseline approach, VELVET [8].** Our approach achieves the highest performance across all metrics in the Big-Vul dataset [16]. Furthermore, it also secures the top F1 score for both function and line-level prediction in the D2A dataset [17].

Notably, our RNN line embedding method significantly enhances the line-level F1 score of CodeT5 (29% $\rightarrow$ 72%), CodeT5P(29% $\rightarrow$ 67%), CodeBERT(27% $\rightarrow$ 63%), and CodeGPT(12% $\rightarrow$ 44%) on both datasets in line-level vulnerability prediction. This suggests that our RNN line embedding approach is better suited for representing code functions. Furthermore, line embeddings learn contextual information on the line level, which may capture the relationships and dependencies between lines more accurately than token embeddings. This makes line embeddings more effective than token embeddings for tasks that require a deeper understanding of the code structure, such as detecting line-level vulnerabilities.

On the function level, we compare our DEEPVULMATCH with the best-performing baseline, CodeT5, and observe a statistically significant improvement ($p < 0.05$, Cohen's $d = 4.761$). On the line level, our DEEPVULMATCH also significantly outperforms the best baseline, VELVET, with $p < 0.05$ and a Cohen's d of 16.355. These results indicate both statistical significance and a large practical effect. These confirm the effectiveness of our proposed deep learning framework for learning and matching vulnerability patterns to predict function and line-level vulnerabilities. These findings also validate our intuition that the line embeddings learned by our proposed method can capture contextual information more effectively than token embeddings, leading to more accurate identification of lines associated with vulnerabilities.

Last but not least, Table III presents the training times of our DEEPVULMATCH approach alongside several baseline methods. Compared to most baselines, DEEPVULMATCH's training time is comparable to common transformer models such as CodeT5, CodeT5P, and GraphCodeBERT. Other methods, such as TextCNN, VELVET, ReGVD, and ICVH, require substantially less training time. Nevertheless, our approach demonstrates substantial performance improvement in line-level vulnerability detection while maintaining comparable training time with transformer-based models. This training efficiency and effectiveness highlight the practical applicability of DEEPVULMATCH in real-world scenarios, where both accuracy and reasonable training duration are critical.

(RQ2) What are the contributions of each component in our DEEPVULMATCH *approach?*

Approach. Our DEEPVULMATCH approach consists of two key components: (i) code line embedding using RNN, introduced in Section III-B, and (ii) optimal transport (OT) and vector quantization (VQ), introduced in Section III-D. We conduct an ablation study to assess the contribution of these proposed components. Firstly, we examine the effect of our RNN line embedding approach by comparing it with commonly used mean and max pooling line embedding methods proposed in Sentence-BERT [13]. Secondly, we investigate the impact of our main components, OT and VQ, on learning

TABLE II
(RQ1 RESULTS) COMPARISON OF DEEPVULMATCH AGAINST 12 BASELINE METHODS ACROSS BIG-VUL AND D2A DATASETS. RESULTS IN PERCENTAGE.

Methods	Function Level			Line Level			Function Level			Line Level		
	Pre	Re	F1	Pre	Re	F1	Pre	Re	F1	Pre	Re	F1
Dataset	*Big-Vul Dataset*						*D2A Dataset*					
DEEPVULMATCH (Ours)	**97.75 ± 0.65**	88.52 ± 1.26	**92.77 ± 0.60**	**86.85 ± 0.64**	**77.64 ± 2.53**	**81.64 ± 1.40**	54.98	**73.89**	**63.04**	**23.46**	31.53	**26.90**
CodeT5P+Our RNN Embedding	93.61 ± 1.09	**89.35 ± 0.83**	91.37 ± 0.49	60.33 ± 3.54	76.29 ± 0.76	67.31 ± 1.54	49.24	51.27	50.23	7.78	38.22	12.93
CodeT5P	90.50 ± 1.97	84.70 ± 1.19	87.91 ± 0.67	18.79 ± 1.61	66.90 ± 2.63	29.45 ± 1.87	58.18	61.15	59.63	1.74	56.37	3.38
CodeT5+Our RNN Embedding	95.91 ± 0.37	89.11 ± 0.45	92.72 ± 0.28	74.01 ± 5.82	69.51 ± 3.20	71.94 ± 3.53	52.38	49.04	50.66	18.79	26.75	22.08
CodeT5	94.30 ± 0.49	84.57 ± 0.66	89.30 ± 0.35	18.15 ± 0.53	68.30 ± 1.31	28.70 ± 0.46	53.68	62.74	57.86	1.42	**57.01**	2.89
CodeBERT+Our RNN Embedding	92.15 ± 0.54	82.89 ± 0.70	87.17 ± 0.52	59.39 ± 0.79	67.84 ± 0.66	63.26 ± 0.50	60.07	55.10	57.48	3.89	45.54	7.18
CodeBERT	92.77 ± 0.68	77.03 ± 0.38	84.03 ± 0.34	17.48 ± 0.76	59.10 ± 1.82	27.05 ± 1.08	2.70	30.89	4.96			
CodeGPT+Our RNN Embedding	91.25 ± 0.50	84.90 ± 0.48	87.91 ± 0.39	32.54 ± 0.72	67.34 ± 0.59	43.88 ± 0.35	62.13	33.44	43.48	3.32	31.21	6.01
CodeGPT	57.59 ± 2.37	17.05 ± 1.77	26.20 ± 2.06	14.36 ± 1.11	10.11 ± 1.45	11.63 ± 0.93	64.37	17.83	27.93	2.40	11.46	3.96
GraphCodeBERT	50.11 ± 4.29	27.03 ± 3.10	35.12 ± 3.81	10.59 ± 3.13	26.34 ± 2.94	15.08 ± 3.58	66.36	45.22	53.79	1.66	53.15	3.21
LineVul	89.00 ± 0.24	78.47 ± 0.34	83.15 ± 0.13	-	-	-	**67.86**	30.25	41.85	-	-	-
VELVET	92.10 ± 1.38	81.60 ± 1.54	86.51 ± 0.30	38.19 ± 1.31	73.50 ± 1.63	50.26 ± 0.75	51.06	46.84	48.86	3.31	38.71	6.10
LineVD	-	-	-	27.04 ± 2.59	53.30 ± 3.23	36.00 ± 3.06	-	-	-	3.24	37.07	5.95
ReGVD	76.82 ± 0.03	50.36 ± 0.03	61.09 ± 0.01	-	-	-	59.87	58.92	59.39	-	-	-
Devign	72.29 ± 0.03	50.24 ± 0.16	59.28 ± 0.16	-	-	-	50.95	68.47	58.42	-	-	-
ICVH	77.44 ± 5.58	30.62 ± 2.95	43.50 ± 3.46	21.69 ± 1.54	41.86 ± 4.37	28.64 ± 2.09	56.99	50.64	53.63	2.32	50.64	4.44
TextCNN	86.50 ± 11.96	58.25 ± 19.38	69.62 ± 19.08	14.12 ± 4.03	53.67 ± 13.74	22.70 ± 1.67	57.29	52.55	54.82	2.00	50.96	3.84

TABLE III
TRAINING TIME OF OUR DEEPVULMATCH AND BASELINE METHODS.

Methods	Training Time
DeepVulMatch	17 hours 15 minutes
LineVul	9 hours 53 minutes
VELVET	3 hours 27 minutes
LineVD	3 hours 4 minutes
ReGVD	5 hours 4 minutes
Devign	9 hours 4 minutes
ICVH	0 hour 33 minutes
TextCNN	4 hours 39 minutes
CodeT5	17 hours 5 minutes
CodeT5P	17 hours 10 minutes
CodeBERT	16 hours 43 minutes
CodeGPT	16 hours 10 minutes
GraphCodeBERT	20 hours 30 minutes

TABLE IV
(RQ2 RESULTS) WE COMPARE OUR PROPOSED METHOD TO OTHER VARIANTS TO INVESTIGATE THE IMPACT OF THE INDIVIDUAL COMPONENTS. THE METRICS ARE REPORTED AS PERCENTAGES.

Methods	Function Level			Line Level		
	Pre	Re	F1	Pre	Re	F1
Dataset	*Big-Vul Dataset*					
DEEPVULMATCH (Ours)	97.66	89.83	93.58	86.80	77.96	**82.14**
w/o RNN emb (mean pooling)	98.49	86.00	91.83	**90.40**	67.89	77.54
w/o RNN emb (max pooling)	96.53	89.95	93.13	79.70	76.40	78.02
w/o codebook & matching	45.91	86.60	60.00	28.77	51.57	36.94
wt 50 centroids	23.95	**98.21**	38.51	16.92	**86.13**	28.28
wt 100 centroids	98.13	87.92	92.74	88.14	74.98	81.03
wt 150 centroids (ours)	97.66	89.83	93.58	86.80	77.96	**82.14**
wt 200 centroids	96.69	90.91	**93.71**	83.44	80.02	81.69
wt 400 centroids	**99.05**	62.32	76.51	81.91	70.47	75.76

TABLE V
(RQ2 RESULTS) THE LINE-LEVEL PERFORMANCE COMPARISON BETWEEN THE TOKEN EMBEDDING AND OUR RNN LINE EMBEDDING.

Methods	Line Level					
	<= 512 Tokens			>512 Tokens		
	Pre	Re	F1	Pre	Re	F1
Dataset	*Big-Vul Dataset*					
CodeT5P+Our RNN Embedding	**69.62**	70.75	**70.18**	**66.03**	**78.33**	**71.66**
CodeT5P	24.42	**83.63**	37.80	14.36	54.52	22.73
CodeT5+Our RNN Embedding	**79.72**	69.17	**74.07**	**77.67**	**63.77**	**70.04**
CodeT5	22.82	**83.38**	35.83	14.33	55.72	22.79
CodeBERT+Our RNN Embedding	**64.26**	70.89	**67.41**	**55.41**	**65.18**	**59.90**
CodeBERT	24.53	**78.71**	37.41	14.94	50.32	23.04
CodeGPT+Our RNN Embedding	**54.45**	**62.24**	**58.09**	**24.96**	**71.78**	**37.03**
CodeGPT	16.21	18.14	17.12	8.25	2.36	3.66

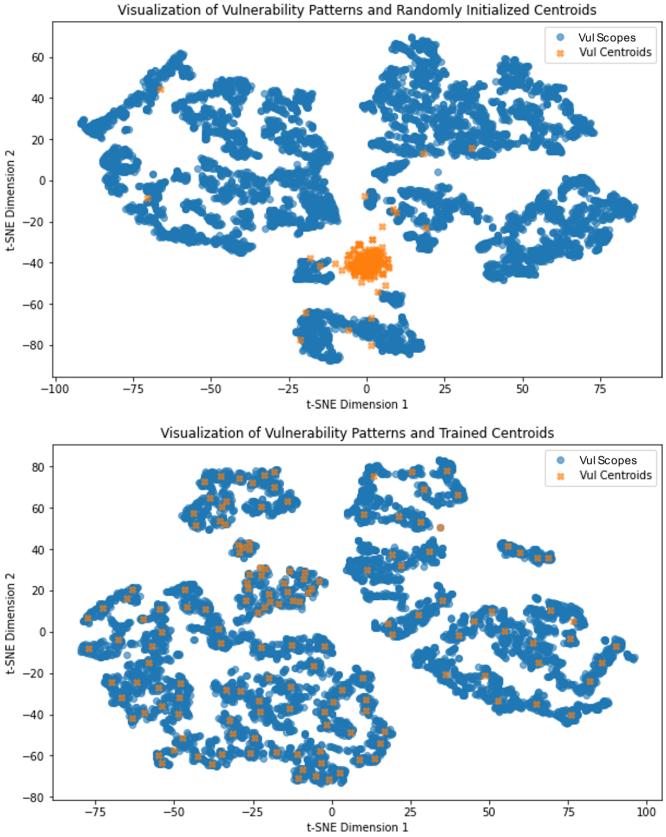

Fig. 7. The t-SNE visualization of vulnerability vectors and centroids.

and matching vulnerability patterns. We compare our approach with an identical variant that does not utilize OT and VQ. Thirdly, our OT process requires us to define k (i.e., the number of vulnerability centroids) to initialize the centroid representations before aggregating similar vulnerability vectors into these centroids. Thus, we analyze the effect of k on the performance of our approach. We compare our approach ($k = 150$) with other variants where k takes different values, specifically $k = [50, 100, 200, 400]$.

In Section III-B, we noted that our embedding method supports inputs up to 3,100 tokens—far beyond the standard 512. In theory, this should improve performance on long sequences.

To evaluate this, we compare the line-level performance of four language models (CodeT5 [56], CodeT5P [55], CodeBERT [36], CodeGPT [57]) using standard 512-token embeddings with our line embeddings, reporting results separately for inputs shorter than 512 tokens and those 512 tokens or longer.

In DEEPVULMATCH, we collected over 6,000 vulnerability vectors from the training data. Matching all patterns at inference is computationally impractical, so we use optimal transport (OT) to aggregate similar vectors—those close in Euclidean space—into representative centroids. In theory, each centroid should represent a group of similar vulnerability vectors. To verify this, we use t-SNE to visualize whether each centroid corresponds to a coherent cluster of vectors. The goal is for vectors and centroids to converge (Figure 3), minimizing the Wasserstein distance [12] as defined in Equation 2.

Result. The results of our ablation study are presented in Table IV. **Our RNN line embedding approach further improves the performance of mean and max pooling by 0.45%-1.75% on the function-level F1 score and 4.12%-4.6% on the line-level F1 score.** The max pooling would lead to information loss since it considers the maximum token embedding for each line, discarding all other token embeddings in the sequence. While the mean pooling considers all token embeddings, it treats all the token embeddings equally regardless of their importance or relevance to the line they belong where the prominent token features could be disregarded.

In contrast, our RNN line embedding approach offers several advantages. Firstly, it learns token features at each time step, allowing for a more nuanced understanding of the code line. Additionally, unlike mean and max pooling, our RNN-based approach retains more information from the entire sequence, thereby capturing a richer representation of the code line. Lastly, by considering the sequential nature of the input tokens, our RNN model can better capture contextual dependencies and relationships within the line. The results confirm the effectiveness of our RNN line embedding method, indicating that it is more effective in summarizing token embeddings.

Our main components, OT and VQ for learning and matching vulnerability patterns, significantly improve the variant, "w/o codebook & matching", by 33.58% on the function-level F1 score and 45.2% on the line-level F1 score. This underscores the importance of these components in achieving high-performance levels. The results suggest that OT and VQ play crucial roles in learning our proposed vulnerability codebook, which is responsible for retaining and leveraging the vulnerability patterns information present in vulnerable functions. This information is then utilized to identify vulnerable lines effectively during the vulnerability-matching inference. The results confirm our design decision of leveraging OT and VQ to effectively aggregate vulnerability vectors into patterns and match those patterns during inference.

The lower section of Table IV illustrates the impact of the number of vulnerability centroids (k) on our approach. The results demonstrate that our approach attains favorable line-level F1 scores for $k \in [100, 150, 200]$. Thus, in our DEEPVULMATCH approach, we empirically set $k = 150$ as it produces the optimal line-level F1 score. Notably, k represents a crucial factor, where a small value of k (e.g., 50) may result in unsatisfactory performance due to the grouping of too many vulnerability vectors together, resulting in an inadequate representation of each pattern. Conversely, a large value of k (e.g., 400) leads to a substantial embedding space of our codebook, making it challenging to update during the backward process. The results confirm the effectiveness of selecting $k = 150$ as the optimal value for the number of vulnerability centroids (k) in our approach.

As shown in Table V, **our line embedding helps the four LMs achieve the best overall performance for both short and long functions**. In particular, our approach substantially enhances the line-level F1 scores, increasing from 23% to 72% for CodeT5P, from 23% to 70% for CodeT5, from 23% to 60% for CodeBERT, and from 4% to 37% for CodeGPT. The results validate the effectiveness of our RNN line embedding approach in accurately identifying vulnerable lines within long functions.

Figure 7 depicts the t-SNE visualization of our vulnerability vectors (depicted as blue dots) and vulnerability centroids (depicted as orange dots), both before and after our primary training phase outlined in Section III-D3. The upper section illustrates the vulnerability vectors and randomly initialized centroids before the optimal transport (OT) process. In contrast, the lower section showcases the learned vulnerability vectors and their associated centroids following the OT process. **This visualization validates that our proposed OT process consolidates closely related vulnerability vectors into representative centroids within the vector space.**

TABLE VI
(DISCUSSION) THE COMPARISON BETWEEN OUR APPROACH WITH RANDOMLY INITIALIZED VULNERABILITY CENTROIDS AND OUR APPROACH WITH OPTICS CLUSTERING ALGORITHM TO DETERMINE THE NUMBER OF VULNERABILITY CENTROIDS.

Methods	Function Level			Line Level		
	Pre	Re	F1	Pre	Re	F1
Dataset	*Big-Vul Dataset*					
DEEPVULMATCH	**97.66**	89.83	**93.58**	**86.80**	**77.96**	**82.14**
DEEPVULMATCH+ OPTICS Clustering	95.80	**89.95**	92.78	82.13	76.47	79.20

V. DISCUSSION

A. Automatic Centroid Selection Using OPTICS Clustering

In Section IV, we evaluated the performance of DEEPVULMATCH and conducted an ablation study to support our design choices. A key limitation is the manual tuning required to determine the optimal number of centroids in our vulnerability codebook, which were empirically set to 150 and randomly initialized before applying optimal transport (OT) as described in Section III-D2. This process can be computationally expensive, especially for larger datasets. To address this, we explore using OPTICS (Ordering Points To Identify the Clustering Structure) [62]—a density-based clustering algorithm that adapts to varying cluster shapes and sizes without requiring a predefined number of clusters. In this section, we evaluate the feasibility of using OPTICS to automatically determine the number of clusters and initialize centroids, replacing the fixed set of 150.

As shown in Table VI, our approach achieves an F1 score of 94% and 82% at the function and line level predictions, respectively. Notably, DEEPVULMATCH+OPTICS Clustering also achieves comparable results, with F1 scores of 93% and 79% at the function and line levels, respectively. While the performance of DEEPVULMATCH+OPTICS Clustering is slightly lower than that of DEEPVULMATCH with randomly initialized centroids, it offers the advantage of automatically determining the ideal number of clusters without the need for empirical tuning. This can save computational resources, particularly for large-scale datasets. The OPTICS clustering algorithm identified a total of 115 clusters, consistent with the findings of our ablation study, suggesting that 100-200 centroids are optimal for our studied dataset. These results validate the effectiveness of our DEEPVULMATCH approach when combined with the OPTICS clustering algorithm, which automatically identifies vulnerability centroids for the subsequent OT process.

TABLE VII
(DISCUSSION) THE PERFORMANCE COMPARISON BETWEEN OUR DEEPVULMATCH APPROACH AND GPT-4.1.

Methods	Function Level			Line Level		
	Pre	Re	F1	Pre	Re	F1
Dataset	*Big-Vul Dataset*					
DEEPVULMATCH (Ours)	**97.66**	**89.83**	**93.58**	**86.80**	**77.96**	**82.14**
CoT + GPT-4.1	7.14	60.00	12.77	3.85	16.67	6.25

B. Comparison with Prompting an LLM

Recent advances in large language models (LLMs) have demonstrated strong performance on various code-related tasks. In particular, OpenAI's GPT-4.1 achieves 54.6% on the SWE-bench Verified benchmark [63], making it one of the leading models for software engineering tasks, outperforming GPT-4o and GPT-4.5 [64]. However, it remains unclear whether such general-purpose LLMs can effectively handle our dual-granularity vulnerability detection task, which requires both function- and line-level reasoning.

To investigate this, we compared our DEEPVULMATCH approach with GPT-4.1 using over 18,000 test samples from the Big-Vul [16] dataset. We adopted the chain-of-thought (CoT) prompting strategy [65] to prompt GPT-4.1 and perform the dual-granularity vulnerability detection task. As shown in Table VII, GPT-4.1 achieved only 12.77% and 6.25% F1 at the function and line levels, respectively. These results align with prior studies on ChatGPT' vulnerability prediction limitations [45], suggesting that fine-tuning remains necessary for such domain-specific, security-critical tasks.

While our method could, in principle, be combined with an LLM like GPT-4.1, doing so would require substantial computational resources, as such models typically contain hundreds of billions of parameters. In contrast, our approach uses only 0.012% of the parameters of GPT-4, making it over 8,000× more parameter-efficient, while still delivering strong performance across both granularities. This underscores its practical value as a lightweight and efficient alternative to fine-tuning LLMs for dual-granularity vulnerability detection.

VI. THREATS TO VALIDITY

Threats to the construct validity relate to the data quality and dataset selection. Our DEEPVULMATCH approach, similar to other data-driven tasks, relies on data quality, and the presence of noisy labels can impact the model's performance. Croft *et al.*[66] conducted a systematic study to assess data quality, evaluating state-of-the-art vulnerability datasets such as Big-Vul [16], D2A [17], and Devign [33]. They analyzed the datasets manually to assess the accuracy of vulnerability labels. Among the three datasets, Devign achieved the highest label correctness. However, Devign only provides function-level labels, which are not compatible with our study's focus on line-level vulnerability detection. It is worth noting that other common vulnerability datasets, such as Reveal [34] and DiverseVul [60], also only consist of function-level labels, which are not suitable for our study.

The Big-Vul dataset is one of the largest vulnerability datasets consisting of line-level vulnerability labels. It has been utilized in recent studies focusing on line-level vulnerability prediction [6, 7, 18]. The line-level labels in Big-Vul were derived from parsing code changes before and after addressing a vulnerability. However, the presence of noisy labels within the dataset may introduce bias and compromise the generalizability of our DEEPVULMATCH approach. To mitigate this threat, we leveraged an additional experimental dataset, the D2A dataset [17], which also provides line-level labels. By including the D2A dataset, which has been used in previous line-level vulnerability detection studies [8], we enhance the robustness of our approach against potential biases introduced by noisy labels in the Big-Vul dataset.

Threats to the internal validity relate to hyperparameter settings in our DEEPVULMATCH approach. In particular, determining the number of vulnerability centroids, k, before the optimal transport (OT) process is crucial for our DEEPVULMATCH approach. We empirically initialize 150 centroids for the OT process, as shown in Table IV. However, this parameter k significantly impacts the performance of our approach, as discussed in our ablation study (RQ2). Empirically determining k could be impractical and computationally intensive for future studies with large-scale datasets. To address this threat, we explore using a clustering algorithm to automatically determine an optimal k in our extended discussion in Section V-A. The results validate that our DEEPVULMATCH approach, combined with the OPTICS clustering algorithm [62], can determine k while maintaining comparable performance, outperforming all other baselines in Table II.

Threats to the external validity relate to the generalizability of our DEEPVULMATCH approach. We conduct our experiment using Big-Vul [16] and D2A [17] datasets consisting of a large amount of C/C++ functions parsed from real-world software projects. However, our DEEPVULMATCH method is not necessarily to generalize to other datasets. To mitigate this threat, we open-source our experimental dataset and model training script in our public replication package available at https://github.com/awsm-research/DeepVulMatch. Nevertheless, other vulnerability datasets can be explored in future work.

VII. Conclusion

This paper presents a novel vulnerability-matching method for function and line-level vulnerability detection (VD). Our approach capitalizes on the vulnerability patterns present in vulnerable programs, which are typically overlooked in deep learning-based VD. Specifically, we collect vulnerability patterns from the training data and learn a more compact vulnerability codebook from the pattern collection using optimal transport (OT) and vector quantization. During inference, the codebook matches all learned patterns and detects potential vulnerabilities within a given program. The evaluation results demonstrate that our method surpasses other competitive baseline methods, while our ablation study confirms the soundness of our approach. While our method demonstrates strong performance, one notable limitation is the manual tuning required to determine the optimal number of centroids in the vulnerability codebook. This parameter is highly dependent on the characteristics of the dataset and the properties of the pre-trained embeddings used. Although our experiments indicate that using OPTICS (Ordering Points To Identify the Clustering Structure) can yield performance comparable to manual tuning, this remains a semi-automated workaround. Future research could explore fully automated techniques to determine the number of centroids, potentially integrating optimal transport with adaptive clustering mechanisms.

Acknowledgment

C. Tantithamthavorn was partially supported by the ARC's DECRA Fellowship (DE200100941).

References

[1] Kyle, "Example c++ vulnerable function - unpremulskimagetopremul," https://github.com/kwyatt/webrtc_src_third_party/commit/a87bf9b1dbdc6e0fb80091a901ab7d5a05d64ecd, 2016.

[2] GoPro, "Example c vulnerable function - isvalidsize," https://github.com/gopro/gpmf-parser/commit/341f12cd5b97ab419e53853ca00176457c9f1681, 2019.

[3] H. Booth, D. Rike, and G. Witte, "The national vulnerability database (nvd): Overview," 2013-12-18 2013. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=915172

[4] NIST, "Apache struts vulnerability (cve-2021-31805)," https://nvd.nist.gov/vuln/detail/CVE-2021-31805, 2022.

[5] R. Croft, D. Newlands, Z. Chen, and M. A. Babar, "An empirical study of rule-based and learning-based approaches for static application security testing," in *Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, 2021, pp. 1–12.

[6] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 596–607.

[7] M. Fu and C. Tantithamthavorn, "Linevul: a transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 608–620.

[8] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, "Velvet: a novel ensemble learning approach to automatically locate vulnerable statements," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 959–970.

[9] CWE, "2023 cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.

[10] A. Van Den Oord, O. Vinyals *et al.*, "Neural discrete representation learning," *Advances in neural information processing systems (NeurIPS)*, vol. 30, 2017.

[11] J. Feydy, T. Séjourné, F.-X. Vialard, S.-i. Amari, A. Trouvé, and G. Peyré, "Interpolating between optimal transport and mmd using sinkhorn divergences," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 2681–2690.

[12] C. Villani, "Optimal transport: Old and new," 2008.

[13] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[14] Checkmarx, "Checkmarx static application security testing," https://checkmarx.com/, 2023.

[15] D. A. Wheeler, "Flawfinder," 2023. [Online]. Available: https://dwheeler.com/flawfinder/

[16] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020, pp. 508–512.

[17] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2a: A dataset built for ai-based vulnerability detection methods using differential analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-Szheng2021d2aEIP)*. IEEE, 2021, pp. 111–120.

[18] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 292–303.

[19] S. Pujar, Y. Zheng, L. Buratti, B. Lewis, Y. Chen, J. Laredo, A. Morari, E. Epstein, T. Lin, B. Yang *et al.*, "Analyzing source code vulnerabilities in the d2a dataset with ml ensembles and c-bert," *Empirical Software Engineering*, vol. 29, no. 2, p. 48, 2024.

[20] G. Monge, "Mémoire sur la théorie des déblais et des remblais," *Mem. Math. Phys. Acad. Royale Sci.*, pp. 666–704, 1781.

[21] M. Thorpe, "Introduction to optimal transport," *Lecture Notes*, vol. 3, 2019.

[22] C. Laclau, I. Redko, B. Matei, Y. Bennani, and V. Brault, "Co-clustering through optimal transport," in *International conference on machine learning*. PMLR, 2017, pp. 1955–1964.

[23] E. Del Barrio, J. A. Cuesta-Albertos, C. Matrán, and A. Mayo-Íscar, "Robust clustering tools based on optimal transportation," *Statistics and Computing*, vol. 29, pp. 139–160, 2019.

[24] Y. Yan, Z. Xu, C. Yang, J. Zhang, R. Cai, and M. K.-P. Ng, "An optimal transport view for subspace clustering and spectral clustering," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 15, 2024, pp. 16 281–16 289.

[25] Y. Liu, Z. Zhou, and B. Sun, "Cot: Unsupervised domain adaptation with clustering and optimal transport," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2023, pp. 19 998–20 007.

[26] X. Lu, H. Wang, W. Dong, F. Wu, Z. Zheng, and G. Shi, "Learning a deep vector quantization network for image compression," *IEEE Access*, vol. 7, pp. 118 815–118 825, 2019.

[27] K. Chen and C.-G. Lee, "Incremental few-shot learning via vector quantization in deep embedded space," in *International conference on learning representations*, 2021.

[28] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[29] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[30] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung, "Deep domain adaptation for vulnerable code function identification," in *The International Joint Conference on Neural Networks (IJCNN)*, 2019.

[31] V. Nguyen, T. Le, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection," *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2020.

[32] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: vulnerability detection with deep learning on a natural codebase for python," *Information and Software Technology*, vol. 144, p. 106809,

2022.

[33] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems (NeurIPS)*, vol. 32, 2019.

[34] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[35] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, "Regvd: Revisiting graph neural networks for vulnerability detection," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 178–182.

[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[37] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[38] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[39] RATS, "Rough auditing tool for security," https://code.google.com/archive/p/rough-auditing-tool-for-security/, 2023.

[40] V. Nguyen, T. Le, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Information-theoretic source code vulnerability highlighting," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[41] B. Zhang, X. Zhi, M. Wang, R. Ren, and J. Dong, "Enhancing java web application security: Injection vulnerability detection via interprocedural analysis and deep learning," *IEEE Transactions on Reliability*, 2025.

[42] Q. Huang, Y. He, Z. Xing, M. Yu, X. Xu, and Q. Lu, "Enhancing fine-grained smart contract vulnerability detection through domain features and transparent interpretation," *IEEE Transactions on Reliability*, 2025.

[43] H. Chu, P. Zhang, H. Dong, Y. Xiao, and S. Ji, "Deepfusion: Smart contract vulnerability detection via deep learning and data fusion," *IEEE Transactions on Reliability*, 2024.

[44] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 166–178.

[45] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" *arXiv preprint arXiv:2310.09810*, 2023.

[46] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[47] H. Shu, M. Fu, J. Yu, D. Wang, C. Tantithamthavorn, J. Chen, and Y. Kamei, "Large language models for multilingual vulnerability detection: How far are we?" *arXiv preprint arXiv:2506.07503*, 2025.

[48] OpenAI, "Chatgpt," https://openai.com/blog/chatgpt, 2022.

[49] Google, "Bard," https://bard.google.com/, 2023.

[50] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[51] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 1715–1725.

[52] M. Cuturi, "Sinkhorn distances: Lightspeed computation of optimal transport," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013.

[53] T. Nguyen, T. Le, N. Dam, Q. H. Tran, T. Nguyen, and D. Phung, "Tidot: A teacher imitation learning approach for domain adaptation with optimal transport," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2021, pp. 2862–2868, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2021/394

[54] N. Ho, X. Nguyen, M. Yurochkin, H. H. Bui, V. Huynh, and D. Phung, "Multilevel clustering via wasserstein means," in *International conference on machine learning*. PMLR, 2017, pp. 1501–1509.

[55] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[56] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *the Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 8696–8708.

[57] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[58] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[59] Y. Chen, "Convolutional neural network for sentence classification," Master's thesis, University of Waterloo, 2015.

[60] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 654–668.

[61] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2018.

[62] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," *ACM Sigmod record*, vol. 28, no. 2, pp. 49–60, 1999.

[63] OpenAI, "Introducing swe-bench verified," https://openai.com/index/introducing-swe-bench-verified/, 2024.

[64] ——, "Introducing gpt-4.1 in the api," https://openai.com/index/gpt-4-1/, 2025.

[65] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[66] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 121–133.