Just-in-Time Detection of Silent Security Patches

XUNZHU TANG, University of Luxembourg, Luxembourg
KISUB KIM, DGIST, Repulic of Korea
SAAD EZZINI, King Fahd University of Petroleum & Minerals , Saudi Arabia
YEWEI SONG, University of Luxembourg, Luxembourg
HAOYE TIAN*, University of Melbourne, Australia
JACQUES KLEIN, University of Luxembourg, Luxembourg
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg

Open-source code is pervasive. In this setting, embedded vulnerabilities are spreading to downstream software at an alarming rate. Although such vulnerabilities are generally identified and addressed rapidly, inconsistent maintenance policies can cause security patches to go unnoticed. Indeed, security patches can be *silent*, i.e., they do not always come with comprehensive advisories such as CVEs. This lack of transparency leaves users oblivious to available security updates, providing ample opportunity for attackers to exploit unpatched vulnerabilities. Consequently, identifying silent security patches just in time when they are released is essential for preventing n-day attacks and for ensuring robust and secure maintenance practices. With LLMDA we propose to (1) leverage large language models (LLMs) to augment patch information with generated code change explanations, (2) design a representation learning approach that explores code-text alignment methodologies for feature combination, (3) implement a label-wise training with labeled instructions for guiding the embedding based on security relevance, and (4) rely on a probabilistic batch contrastive learning mechanism for building a high-precision identifier of security patches. We evaluate LLMDA on the PatchDB and SPI-DB literature datasets and show that our approach substantially improves over the state-of-the-art, notably GraphSPD by 20% in terms of F-Measure on the SPI-DB benchmark.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: security patch detection, in-context learning, self-instruct

ACM Reference Format:
Xunzhu Tang, Kisub Kim, Saad Ezzini, Yewei Song, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. 2025. Just-in-Time Detection of Silent Security Patches. 1, 1 (August 2025), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

According to a recent market report[1], 96% of applications have at least one open-source component, while open-source code comprises about 80% of a given modern application. These impressive

*Corresponding author.
[1]https://gitnux.org/open-source-software-statistics/

Authors' addresses: Xunzhu Tang, xunzhu.tang@uni.lu, University of Luxembourg, Luxembourg; Kisub Kim, falconlk00@gmail.com, DGIST, Repulic of Korea; Saad Ezzini, s.ezzini@lancaster.ac.uk, King Fahd University of Petroleum & Minerals , Saudi Arabia; Yewei Song, yewei.song@uni.lu, University of Luxembourg, Luxembourg; Haoye Tian, haoye.tian@unimelb.edu.au, University of Melbourne, Australia; Jacques Klein, jacques.klein@uni.lu, University of Luxembourg, Luxembourg; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg.

ACM XXXX-XXXX/2025/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

statistics indicate that open-source software (OSS) is a key element whose engineering should be closely monitored: vulnerabilities in OSS will spread to a broad range of downstream software systems. Once discovered, they enable attackers to perform "n-day" attacks against unpatched software systems.

Timely software patching remains the first defense against attacks exploiting OSS vulnerabilities [19, 36]. Unfortunately, security patches can go unnoticed. On the one hand, the ever-increasing number of submitted patches and security advisories can overwhelm reviewers and system administrators. On the other hand, the complexity of patch management processes and the inconsistency of OSS maintenance policies can lead to the release of *silent security patches*. Such patches are submitted to the OSS repository but no specific notice is provided for maintainers of downstream software systems. Silent security patches lead to unfortunate delays in software updates [9].

Consider the example from the Linux kernel (shown in Listing 1), which illustrates the severity of silent security patches. In this patch, a boundary check was added to mitigate a buffer overflow vulnerability in TCP options handling. However, the patch lacked proper documentation or advisories, leading to the possibility that downstream maintainers might overlook it. Such omissions exemplify the need for automated tools capable of detecting silent security patches, as relying solely on manual reviews is insufficient for large-scale open-source ecosystems. In this example, GraphSPD would focus only on the immediate code changes, such as the addition of a boundary check, without considering how this update interacts with other functions or modules. By contrast, LLMDA augments the patch with LLM-generated explanations, such as "This patch prevents buffer overflow by validating input lengths," which provide missing semantic context. Additionally, PT-Former integrates this semantic information with the code structure, ensuring a comprehensive understanding of how the patch mitigates vulnerabilities across the broader system.

Detecting silent security patches is a timely research challenge that has gained traction in the literature. The prior approaches in the literature attempt to analyze code changes and commit logs within patches in order to derive security relevance. However, on the one hand, the semantics of code changes are challenging to precisely extract statically. Patches are more often non-atomic, meaning that beyond a security-relevant code change, other cosmetic or non-security changes are often involved. On the other hand, commit messages, which are supposed to describe precisely the intention of the code changes, are often missing, mostly lacking sufficient information, and sometimes misleading.

Recent literature has largely leveraged machine learning to improve the performance of security patch detection systems. In general, the proposed approaches [43, 45, 48] rely on syntactic features, while other methods [47, 60] have explored deep neural networks by treating patches as sequential data. Wang *et al.* [44] recently introduced GraphSPD, which models semantics using graph-based representations of source code. GraphSPD significantly advances the state of the art by capturing localized context within patches, thereby outperforming prior syntactic and sequential approaches. However, GraphSPD primarily focuses on local code segments and direct dependencies, limiting its ability to capture global interactions between functions or modules. Unlike GraphSPD, LLMDA does not aim to explicitly model inter-modular interactions. Instead, it focuses on enriching the semantic understanding of patches through multi-modal inputs, including LLM-generated explanations, commit descriptions, and task-specific instructions. These components enable LLMDA to capture nuanced semantic and contextual information within patches, particularly when documentation is limited or misleading. While LLMDA excels at integrating and aligning multi-modal representations to improve classification accuracy, its design does not extend to modeling the broader context of how functions or modules interact.

To address these limitations, LLMDA leverages its multi-modal design to capture both local and global contexts. First, LLM-generated explanations bridge the gap between code and intent, providing semantic insights that contextualize local changes within their broader functional and modular implications. Second, PT-Former aligns and fuses multi-modal inputs—including code, descriptions, and explanations—into a unified embedding, enabling the model to capture relationships beyond local code structures. Finally, the stochastic batch contrastive learning (SBCL) mechanism refines decision boundaries by leveraging both local and global contexts, improving the model's robustness in distinguishing complex security patches.

To cope with the aforementioned challenges, our intuition is threefold: ❶ First, the security relevance of a patch could be better identified if a proper and detailed *explanation of code changes* can be obtained. To that end, we look towards the current wave of Large Language Models (LLMs), where various studies [21, 34, 35] have demonstrated their capabilities in effectively capturing the essential context and tokens within source code for a variety of tasks. ❷ Second, the patch representation must effectively learn to *combine and align features* from the code changes with features of the change descriptions to maximally capture the relevant details for security relevance identification. ❸ Third, a *language-centric approach* where natural language *instructions* are used within the inputs to guide the learning could help exploit the power of existing general models as shown in recent papers for various tasks [5, 7, 51].

This paper. We propose and implement LLMDA (read λ), a novel framework for detecting silent security patches in open-source software. LLMDA addresses the limitations of existing methods, such as GraphSPD, by introducing the following components:

1). **LLM-Generated Explanations**: These explanations address the lack of detailed commit messages by enriching patches with semantic context, enabling the model to understand the intent behind code changes and their broader implications. 2). **PT-Former Module**: This module aligns and fuses multi-modal inputs, resolving the problem of disjoint embeddings for code and text modalities. PT-Former ensures that both local and global contexts are effectively captured. 3). **Stochastic Batch Contrastive Learning (SBCL)**: By refining classification boundaries through batch-wise comparisons, SBCL enhances the model's ability to distinguish between security-relevant and non-relevant patches, particularly in edge cases. LLMDA leverages multi-modal inputs, including code changes, developer-provided descriptions, and large language model (LLM)-generated explanations, to enrich the context and improve detection accuracy. By augmenting incomplete or vague commit messages with natural language explanations generated by LLMs, LLMDA bridges the gap between code semantics and human-readable context. Furthermore, task-specific instructions are incorporated to guide the model's learning and ensure alignment with the security classification objective. We conducted extensive ablation studies to demonstrate the individual contributions of LLM-generated explanations, PT-Former, and SBCL. These studies show how each component enhances LLMDA 's capability to detect silent security patches with higher precision and robustness compared to existing methods.

To integrate these diverse inputs, LLMDA introduces PT-Former, a specialized module for aligning and fusing embeddings from code and text modalities. PT-Former employs self-attention and cross-attention mechanisms to create a unified representation that captures both local and global contexts. Once the embeddings are generated, a stochastic batch contrastive learning (SBCL) mechanism is applied to refine the model's decision boundaries by emphasizing inter-class distinctions and intra-class consistency. This systematic combination of components allows LLMDA to achieve state-of-the-art performance in detecting silent security patches across diverse datasets, addressing the critical limitations of existing methods.

Our contributions are as follows:

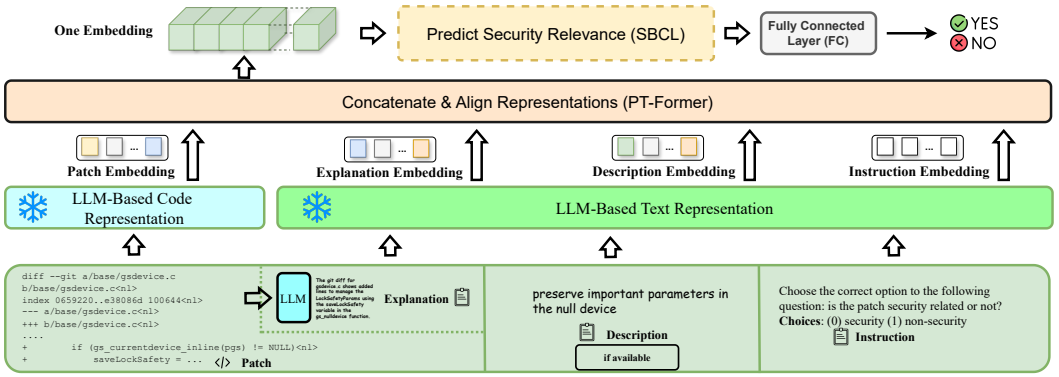


Fig. 1. Overview of LLMDA

- We introduce LLMDA, a framework that overcomes the limitations of existing methods, such as GraphSPD, by capturing both local and global contexts in patch analysis.
- We leverage LLM-generated explanations to provide semantic insights and augment patches with missing context, bridging the gap between code and intent.
- We propose PT-Former, a novel alignment module that integrates multi-modal inputs, enabling the model to capture inter-functional and inter-modular relationships effectively.
- We employ stochastic batch contrastive learning (SBCL) to refine classification boundaries, improving robustness and precision in distinguishing complex patches.
- We achieve state-of-the-art performance on PatchDB and SPI-DB datasets, with up to 42% improvement in F1-score over the incumbent GraphSPD model.

2 MOTIVATION

Open-source software (OSS) has become a cornerstone of modern software development, accounting for a significant portion of both commercial and non-commercial applications. However, the increasing reliance on OSS has exposed new challenges in maintaining its security, particularly in the timely identification and application of security patches. The motivation for this work stems from three major aspects: the risks posed by silent security patches, the limitations of existing solutions, and the transformative potential of large language models (LLMs).

2.1 Silent Security Patches: A Critical Threat

Security vulnerabilities in OSS propagate rapidly to downstream software, potentially affecting a vast number of applications. While vulnerabilities are often addressed through patches, a significant portion of these patches are released silently, without clear advisories or documentation, such as CVEs. These silent security patches leave users unaware of critical updates, creating opportunities for attackers to exploit known vulnerabilities, known as n-day attacks.

Consider the following example:

```
diff --git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
index c34fa2a..e4fb7a5 100644
--- a/net/ipv4/tcp_input.c
+++ b/net/ipv4/tcp_input.c
@@ -2342,7 +2342,8 @@ void tcp_ack(struct sock *sk, struct sk_buff *skb, int flag)

     /* Fix potential buffer overflow in TCP options */
-    memcpy(opt, skb->data + tcp_hdrlen, optlen);
+    if (unlikely(optlen > TCP_MAX_OPT_LENGTH))
+        return;
+    memcpy(opt, skb->data + tcp_hdrlen, optlen);
```

```
12 }
```

Listing 1. A silent security patch from the Linux kernel codebase.

In this example, the patch fixes a potential buffer overflow vulnerability in TCP options handling by adding a safety check for the option length (‘optlen‘). Without explicit documentation or advisory, this patch could be missed by downstream maintainers, leaving many systems exposed to exploitation.

2.2 Limitations of Existing Approaches

Previous approaches to identifying security patches have significant limitations:

- **Static Analysis and Rule-Based Tools**: These methods rely on predefined patterns or static analysis to detect security patches. However, they fail to capture the semantic intent of code changes, resulting in high false-positive rates.
- **Token and Graph-Based Models**: Machine learning models, such as token-based neural networks and graph-based representations like GraphSPD, show promise but often fail to generalize across diverse projects and cannot effectively utilize broader context, such as commit messages or developer intent.
- **Commit Message Gaps**: Commit messages often lack sufficient detail or are misleading, further compounding the challenge of identifying security patches.

2.3 Opportunities with Large Language Models

LLMs have demonstrated exceptional capabilities in understanding and generating both natural language and code. This work seeks to leverage these capabilities to:

(1) **Bridge the Gap Between Code and Intent**: By generating explanations for code changes, LLMs can provide human-readable insights that align with the patch's purpose.
(2) **Enhance Multi-Modal Understanding**: Combining code, commit messages, and natural language instructions allows for a richer representation of patches.
(3) **Enable Scalability**: LLMs can process large-scale datasets and adapt to various OSS ecosystems, ensuring a scalable and generalizable solution.

2.4 Research Objective

The overarching objective of this work is to develop a robust, scalable, and interpretable framework for detecting silent security patches in OSS. By addressing the shortcomings of existing methods and harnessing the potential of LLMs, this approach aims to:

- Identifying security patches promptly to minimize exposure to n-day attacks.
- Reducing false positives and improving detection accuracy.
- Generalizing across diverse OSS ecosystems and programming languages.

This motivation serves as the foundation for the design and implementation of `LLMDA`, a novel framework that leverages LLMs, multi-modal learning, and contrastive techniques to redefine security patch detection.

3 APPROACH

Figure 1 shows an overview of the different steps of our approach LLMDA. First, representations of multi-modal inputs (code and texts) are obtained using LLMs. Then, the obtained representations are aligned within a unique embedding space and fused into a single comprehensive representation by the **PT-Former** module. Finally, a stochastic batch contrastive learning mechanism (**SBCL**) is implemented to predict whether a given patch is a security patch or not.

3.1 Data augmentation with LLMs

The intention behind code changes is supposed to be provided in the patch description. This information is then expected to be essential for the detection of security patches. Unfortunately, commit messages, which are meant to convey patch descriptions, are often missing, mostly not sufficiently detailed, and even sometimes misleading. In LLMDA, we explore the power of LLMs, which have demonstrated remarkable capabilities on a broad spectrum of tasks [41], to explain patches. As illustrated in Figure 1, each patch is used to prompt ChatGPT (gpt-3.5-turbo-16k-0613), to produce a natural language explanation based on the following prompt instruction: "*Could you provide a concise summary of the specified patch?*"[2].

In transformer-based models, a classification token such as [CLS] is typically used to generate a general representation for downstream tasks. However, in LLMDA, we incorporate task-specific instruction as an additional input embedding to guide the model's attention towards security-related features. The instructions, as validated by recent work [6, 51], serve as a task alignment mechanism that improves the contextual understanding of the model by explicitly framing the classification objective. This is particularly crucial in security patch detection, where the subtleties of code changes and textual descriptions require precise alignment between data modalities.

The specialized instruction "*Choose the correct option to the following question: is the patch security related or not? Choices: (0) security (1) non-security*," contextualizes the model's representation learning by explicitly associating each input with its classification objective. This process enables LLMDA to consistently improve its focus on meaningful relationships between the code and its natural language description, enhancing robustness and performance across diverse datasets.

3.2 Generation of Bimodal Input Embeddings

LLMDA operates with bimodal inputs: code in the form of program patches, and text in the form of natural language descriptions of code changes and the instructions for label-wise training. We generate embeddings for each input using an adapted deep representation learning model.

Patch Embeddings: We build on CodeT5+ to infer the representation of patches. This pre-trained model is known to be one of the best-performing models for code representation learning[3]. Given a code snippet, which is a sequence of tokens $C = \{c_1, c_2, \ldots, c_n\}$, $P \in \mathcal{R}^{n \times d_{\text{in}}}$ is the associated matrix representation, where each row corresponds to the embedding of a token in C, and d_{in} is the input embedding dimension of the model.

We then apply a linear transformation followed by a non-linear activation to yield the patch embedding E_p. Specifically, the transformation function $f_{\text{CodeT5+}}$ is defined as:

$$E_p = f_{\text{CodeT5+}}(P) = \mathcal{F}(P \cdot W_p + b_p), \tag{1}$$

where $W_p \in \mathcal{R}^{d_{\text{in}} \times d_{\text{out}}}$ is the weight matrix, $b_p \in \mathcal{R}^{d_{\text{out}}}$ is a bias vector, d_{out} is the output embedding dimension, and $\mathcal{F}$ denotes a non-linear activation function (e.g., ReLU). The bias vector b_p is broadcast across all rows of the matrix $P \cdot W_p$ to ensure dimensional compatibility.

Text Embeddings: We leverage LLaMa-7b for representing text input. This pre-trained large language model (LLM) demonstrates robust generalization capabilities across diverse domains. Similarly to the embedding process for patches, for a sequence of textual tokens, we represent the input as $T \in \mathcal{R}^{m \times d_{\text{in}}}$, where m is the number of tokens in the sequence, and d_{in} is the embedding dimension.

The transformation function f_{LLaMa} is applied to generate the text embedding E_t:

[2]We have experimented with a variety of variations for this prompt and obtained similar outputs.
[3]https://huggingface.co/Salesforce/codet5p-220

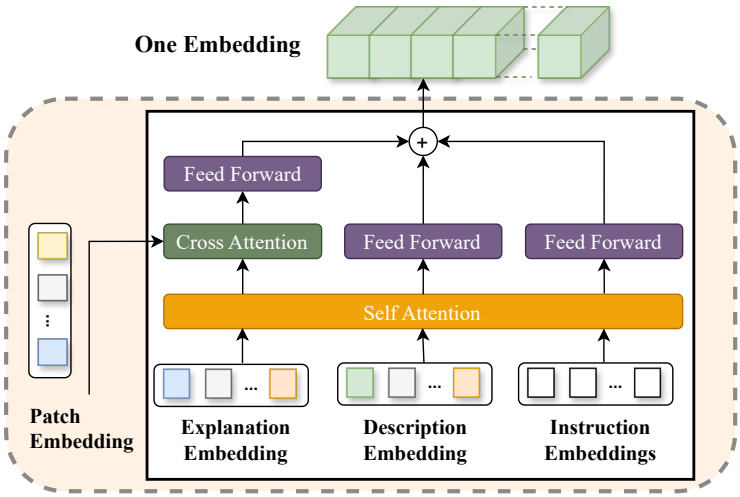


Fig. 2. Architecture of PT-Former.

$$E_t = f_{\text{LLaMa}}(T) = \mathcal{G}(T \cdot W_t + b_t), \tag{2}$$

where $W_t \in \mathcal{R}^{d_{\text{in}} \times d_{\text{out}}}$ is the weight matrix, $b_t \in \mathcal{R}^{d_{\text{out}}}$ is the bias vector, and $\mathcal{G}$ is a non-linear activation function. Similar to the patch embeddings, the bias vector b_t is broadcast along the rows of $T \cdot W_t$.

LLMDA is fed with three text inputs: generated code change explanations, developer-provided patch descriptions, and the instruction. Using the aforementioned process, we produce embeddings E_t^{ex}, E_t^{desc}, and E_t^{inst} respectively for each input.

Dimensional Consistency: In Formulas (1) and (2), the dimensions of W_p and W_t are defined as $d_{\text{in}} \times d_{\text{out}}$, where d_{in} is the input embedding dimension (e.g., 768 for CodeT5+ or LLaMa-7b), and d_{out} is the output embedding dimension. The bias vectors b_p and b_t are 1D vectors of size d_{out}, ensuring that the addition operation is valid by broadcasting b_p and b_t across all rows of their respective matrices.

3.3 PT-Former: Embeddings alignment and Concatenation

As the given two embeddings E_p and E_t represent two different modalities, a patch and a text, their feature spaces differ. In order to leverage pre-trained unimodal models for silent security patch detection, it is key to facilitate cross-modal alignment. In this regard, existing methods (e.g. BLIP2 [20], InstructBLIP [7]) resort to an image-text alignment, which we show is insufficient to bridge the modality gap. There is thus a need to align the embedding spaces before concatenating the relevant embeddings to produce a comprehensive representation of the input for the training of the classification model.

Figure 2 overviews *PT-Former*, a new architecture that we have designed for aligning embedding spaces and fusing the embeddings of LLMDA's bi-modal inputs. With *PT-Former*, we employ a self-attention mechanism to update all embeddings for a generated explanation, the human patch description, and the devised instruction. We leverage a cross-attention module between the patch embedding and the updated explanation module. Feed-forward layers are then used to align the matrix size of all hidden states before concatenating all three embeddings into a single output embedding.

Self-Attention Mechanism (SA). The self-attention mechanism is a fundamental component of the transformer architecture, designed to model interactions between elements in a sequence, enhancing the representation of each element by aggregating information from all other elements [8]. Because attention allows for a dynamic weighting of the importance of inputs' contribution to the representation of others, exploiting it in LLMDA will enable it to understand contextual relationships within the input data. In PT-Former, we implement a multi-head attention mechanism with h heads to capture various aspects of these interactions, initializing each head's query (Q), key (K), and value (V) matrices with values drawn from a standard normal distribution:

$$W_{Q_i}, W_{K_i}, W_{V_i} \sim \mathcal{N}(0,1), \quad i = 1, ..., h \tag{3}$$

where Q, K, and V are respectively the query, key, and value for each embedding to be calculated inside the self-attention.

Consider, for example, the weight matrix of the explanation E_t^{ex}. Our self-attention mechanism over E_t^{ex} (simply noted E_{ex}) is computed as:

$$\hat{E}_{ex} = \mathcal{SA}(E_{ex}) = \text{Softmax}\left(\frac{E_{ex}W_{Q_i}(E_{ex}W_{K_i})^T}{\sqrt{dim}}\right)E_{ex}W_{V_i} \tag{4}$$

where *dim* represents the dimensionality of the embeddings. Similarly, the two other text embeddings (i.e., E_t^{desc} and E_t^{inst}) are passed through the $\mathcal{SA}$ operation to obtain their updated embeddings, we will obtain updated embeddings, $\hat{E}_t^{desc}$ and $\hat{E}_t^{inst}$ respectively.

Cross-Attention for Alignment (CA). Cross-attention mechanisms have proven to be very effective in linking the semantic spaces between different types of data [49][52]. We employ CA to align the embedding spaces of code changes (E_p), yielded by CodeT5+, and explanations (E_t^{ex}), yielded by LLaMa-7b. We focus on *explanation*, since it is the main text input that we associate with the patch: description can be missing while instruction is always the same. However, it should be noted that all text inputs are embedded with LLaMa-7b and are thus in the same embedding space as an explanation. The key feature of cross-attention is its ability to selectively focus on and integrate relevant information from both code and natural language explanations. This helps in achieving a better understanding of the relationship between the syntactical structure of code and its interpretation in natural language. The cross-attention computation therefore explicates the interaction between code changes (E_p) and their explanations (E_t^{ex}). CA starts by transforming E_p and $\hat{E}_t^{expl}$ into query (Q_{pa}), key (K_{ex}), and value (V_{ex}) matrices using learnable weights. The attention mechanism then calculates how much focus each part of the code changes should give to different parts of the explanations. This is done by computing attention scores, which determine the output, effectively linking code changes to their explanations. The process is summarized as follows:

$$\begin{gathered} Q_{pa} = E_{pa}W^Q, \quad K_{ex} = E_{ex}W^K, \quad V_{ex} = E_{ex}W^V, \\ E_{pa-ex} = \text{softmax}\left(\frac{Q_{pa}K_{ex}^T}{\sqrt{dim}}\right)V_{ex} \end{gathered} \tag{5}$$

where $E_{ex} = \hat{E}_t^{expl}$ the updated embedding of explanation input through Self-Attention, W^Q, W^K, and W^V are the weight matrices to be learned. E_{pa-ex} is the fused embedding of E_{pa} and E_{ex}.

Embedding Fusion and Non-linear Transformation. We then pass the updated embeddings to feedforward layers. Each feedforward process involves two dense layers with ReLU activation. We represent the feedforward process by the function $FF(\dots)$. Then, E_{pa-ex}, $\hat{E}_{desc}$, $\hat{E}_{inst}$ can be updated as $E_{pa-ex} = FF(E_{pa-ex})$, $E_{desc} = FF(\hat{E}_{desc})$, and $E_{inst} = FF(\hat{E}_{inst})$.

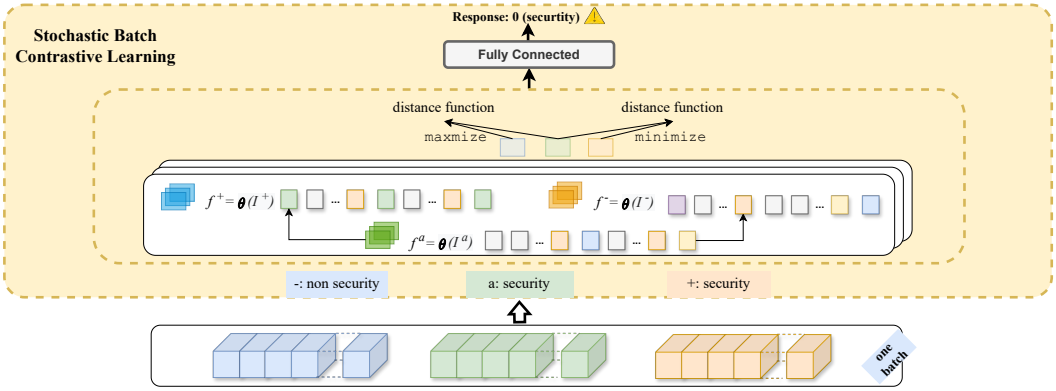


Fig. 3. Overview of our SBCL layer.

After obtaining attention outputs from all heads, we concatenate them to generate one embedding:

$$E = E_{pa-ex} \oplus E_{desc} \oplus E_{inst} \tag{6}$$

where $\oplus$ is the concatenation operation.

Label-wise Attention with Instruction. Inspired by the results of InstructionBLIP [7], we postulate that an instruction that combines a question with explicit labels can provide two advantages in our security detection task: First, it can provide guidance to train models in the direction of answering the security question. Second, since it can provide the opportunity to build a relationship between inputs and the instruction labels through the calculation of their high-dimensional embeddings, leading the model to leverage instructions in a label-wise manner. In conclusion, the design of the instruction and its embedding within will help guide the model to focus on particular aspects of the data, thereby improving the representational efficiency for our targeted downstream task.

3.4 Stochastic Batch Contrastive Learning (SBCL)

Once *PT-Former* outputs a single embedding for each sample to be assessed, we must learn to predict whether it is a security patch or not. At this point, the patch is represented along with its LLM-generated explanation, developer description, and the labeled instruction in *PT-Former*. LLMDA must therefore feed it into a binary classifier to predict security relevance (cf. Figure 1).

To enhance the learning process by effectively leveraging the intrinsic patterns within the dataset, we design a Stochastic Batch Contrastive Learning (SBCL) (see in Figure 3) mechanism for security patch identification. *SBCL* is designed to operate on batches of data comprising fused embeddings of security-related and non-security-related inputs (i.e., E in Eq. 6)

Given a batch of data $\mathcal{B}$ containing embeddings $E = \{E_{pa-ex}, E_{de}, E_{in}\}$ for each data point, we employ a stochastic batch contrastive learning mechanism to discern between security and non-security data points. For each batch, we randomly select an anchor data point related to security. We then identify positive samples within the batch that are also related to security and negative samples that are not. This forms a triplet for each anchor comprising the anchor, positive, and negative samples.

Batch Sampling and Triplet Formation. In the context of SBCL, each batch $\mathcal{B}$ is carefully constructed to include a balanced mix of security-related (*security*) and non-security-related (*non-security*) examples. From each batch, we systematically form triplets for training. A triplet consists of an anchor (a), a positive example (p), and a negative example (n). The anchor and positive

examples are drawn from the *security* category, ensuring they share underlying security-relevant features, whereas the negative example is selected from the *non-security* category.

Batch Mining of Positive and Negative Pairs. In the SBCL framework, a systematic approach is used to select positive and negative pairs within each batch. This process utilizes embeddings generated by *PT-Former* for all examples in a batch. The selection criterion for a positive example is its reduced similarity to the anchor, aimed at maximizing intra-class variability. Conversely, a negative example is deemed challenging on the basis of its increased similarity to the anchor, designed to augment the model's precision in distinguishing between closely associated examples of different classes.

The selection of informative positive and negative pairs is facilitated by measuring the distances between embeddings in the batch. The Euclidean distance formula is applied to determine the distance $d(E_a, E_b)$ between two embeddings E_a and E_b:

$$d(E_a, E_b) = \sqrt{\sum_{i=1}^{dim} (E_a^{(i)} - E_b^{(i)})^2} \tag{7}$$

This methodological approach ensures the identification and use of the most relevant examples to enhance the discriminative capacity of the model.

Stochastic Batch Contrastive Loss. We design the stochastic batch contrastive loss to optimize the embedding space in order to distinguish between security-related and non-security-related examples effectively. This objective is achieved by minimizing the distance between embeddings of anchor and positive pairs and maximizing the distance between the embeddings of anchor and negative pairs within each batch. The loss for a given triplet (a, p, n) is mathematically defined as:

$$L(a, p, n) = \max(0, d(E_a, E_p) - d(E_a, E_n) + \text{margin}) \tag{8}$$

where $d(E_x, E_y)$ calculates the distance between two embeddings E_x and E_y, and margin is a predefined margin that enforces a minimum distance between the anchor-positive and anchor-negative pairs and its value is set as 0.1 here.

The batch loss is computed as the mean of the losses for all triplets within the batch:

$$L_{\text{SBCL}} = \frac{1}{|\mathcal{T}|} \sum_{(a,p,n) \in \mathcal{T}} L(a, p, n) \tag{9}$$

where $\mathcal{T}$ denotes the set of all triplets in the batch. This formulation ensures the development of an embedding space that accurately represents the distinctions between security and non-security instances, facilitating effective classification.

3.5 Prediction and Training Layer for Security Patch Detection

The final component of LLMDA is a Training and Prediction Layer, specifically designed for security patch detection. This layer is responsible for interpreting the fused embeddings produced by *PT-Former* and making accurate predictions regarding the security relevance of each patch.

Training Procedure. Training the model to accurately predict security patches involves minimizing a loss function that measures the discrepancy between the predicted probabilities and the ground-truth labels. A commonly used loss function for binary classification tasks is the binary cross-entropy loss, given by:

$$L_{BCE} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(P_i) + (1 - y_i) \log(1 - P_i)] \tag{10}$$

where N is the number of examples in the training set, y_i is the ground-truth label for the i-th example (1 for security-related and 0 for non-security-related), and P_i is the computed probability for the i-th example to be security-related.

In an end-to-end training regime, both the contrastive loss from the previous sections and the BCE loss are combined: $L = L_{BCE} + L_{SBCL}$. At the end of the training, a learned weight matrix is available to drive inference.

Prediction Step. The prediction mechanism utilizes a fully connected (FC) neural network layer that takes as input the fused embedding from *PT-Former*, representing the unified view of the patch, its generated explanation, the developer description, and LLMDA instruction. The FC layer is defined as follows:

$$P = \sigma(W_p \cdot E + b_p) \tag{11}$$

where E denotes the single fused embedding input, W_p is the learned weight matrix of the FC layer, b_p is the bias term, and σ represents the activation function, typically a sigmoid function for binary classification tasks such as security patch detection. The output P signifies the probability that a given patch is security-relevant.

4 EXPERIMENTAL SETUP

We discuss the research questions that we investigate before presenting the baselines and datasets as well as the evaluation metrics.

4.1 Research Questions

- **RQ-1** *How effective is* LLMDA *in identifying security patches?* We assess LLMDA against well-known literature benchmarks and compare the achieved performance against some strong baselines.
- **RQ-2** *How do key design decisions in* LLMDA *contribute to its performance?* We perform an ablation study where we investigate the added value of label-wise training, the generated explanations, PT-Former and contrastive learning.
- **RQ-3** *To what extent the distribution of patch representations in* LLMDA *improves over the state of the art?* We visualize the learned representations from LLMDA and GraphSPD to observe the differences in their potential discriminative power. Based on case studies, we also qualitatively assess how LLMDA representation assigns scores to key tokens.
- **RQ-4** *Does the trained* LLMDA *model generalize beyond our study dataset?* We evaluate the robustness of LLMDA by applying the model trained on a given dataset to samples from a different dataset.

4.2 Datasets

We consider two datasets from the recent literature :

- **PatchDB** [46] is an extensive set of patches of C/C++ programs. It includes about 12K security-relevant and about 24K non-security-relevant patches. The dataset was constructed by considering patches referenced in the National Vulnerability Database (NVD) as well as patches extracted from GitHub commits of 311 open-source projects (e.g., Linux kernel, MySQL, OpenSSL, etc.).
- **SPI-DB** [60] is another large dataset for security patch identification. The public version includes patches from FFmpeg and QEMU, amounting to about 25k patches (10k security-relevant and 15k non-security-relevant).

We selected the aforementioned datasets because they collectively provide a significant variety in the vulnerabilities as well as a spectrum of patches (with different styles, syntax and semantic implementations). Thus, they are suitable for intra-project and cross-project assessment.

4.3 Evaluation Metrics

We consider common evaluation metrics from the literature:

- **+Recall** and **-Recall**. These metrics are borrowed from the field of patch correctness prediction [42]. In this study, +Recall measures a model's proficiency in predicting security patches, whereas -Recall evaluates its capability to exclude non-security ones.
- **AUC and F1-score [15]**. The overall effectiveness of LLMDA is gauged using the AUC (Area Under Curve) and F1-score metrics.

4.4 Baseline Methods

- **GraphSPD**: We consider the most recently published state-of-the-art GraphSPD [44], which, after demonstrating that prior token-based approaches do not capture sufficient semantics, deploys a cutting-edge graph neural network method for security patch detection. Indeed, it represents a significant advancement by using graph representations of patches, allowing for richer semantics compared to previous deep neural network methods relying on token sequences.
- **TwinRNN**: In our study, we opt for RNN-based solutions [47][60], which leverage a twin RNN architecture to assess the security relevance of a given patch. This approach involves employing two RNN modules, each equipped with shared weights, to analyze the code sequences before and after the patch application.
- **GPT**: We consider LLMs as a relevant baseline given that we employ them as part of our pipeline (to generate patch explanations). We opt for GPT (v3.5) [4], which is accessible. We prompt it with the following instructions: *"Given the following code change, determine if it is related to a security vulnerability or not. Please respond with either 'security' or 'non-security' and you must provide an answer. [*`Patch information`*]"*
- **CodeT5**: Similarly to GPT, because the CodeT5 [50] encoder-decoder model is a core component that is used as an initial embedder of patches in LLMDA, we consider it as a baseline approach for classifying patches.
- **VulFixMiner** [57] builds on the CodeBERT transformer-based approach for representing patches to train the security patch identification classifier. We reproduce it as a baseline.

Beyond these baselines, the literature in software engineering has recently proposed **CoLeFunDa** [56]. However, we do not directly compare against it in our work because it is closed-source and is not readily reproducible.[4]

With CoLeFunDa, the authors propose to use the GumTree differencing tool to extract the description of changes that are made. It considers syntactic descriptions of change operations (e.g., UPDATE invocation at IF) while our approach generates descriptions that provide step by step reasoning. It should be noted that its major benefit is visible in terms of effort-based metrics. In the original publication, the authors show that it improves over VulFixMiner by 1% in terms of AUC.

4.5 Implementation

LLMDA is implemented using the PyTorch library (version 11), and experiments are conducted on two NVIDIA V100 GPUs (32GB) with CUDA 11. The datasets are partitioned with 80% of the samples used for training and 20% for testing, ensuring a balanced evaluation setup. To ensure statistical reliability, all experiments are repeated three times with different random seeds, and the results are averaged. The AdamW optimizer [24] is employed for weight optimization, with a learning rate of 1×10^{-5} and a decay rate of 0.01, ensuring convergence and regularization over 20

[4]We have requested access to the code. However, the authors have replied that they are not authorized to share it by their employer.

epochs. Batch sizes are set to 16 for training and 64 for testing to balance computational efficiency and performance.

Key hyperparameters, including temperature and dropout, are configured to 0.1 and 0.5, respectively. The **temperature** parameter is crucial in controlling the randomness of text generation when using large language models like GPT-3.5 or GPT-4. Specifically, temperature, denoted as T, scales the logits l_i of the model outputs, modifying the probability distribution of generated tokens as follows:

$$p_i = \frac{\exp(l_i/T)}{\sum_j \exp(l_j/T)}. \tag{12}$$

A lower temperature ($T < 1$) reduces randomness by flattening the probability distribution, emphasizing high-probability tokens and generating more deterministic outputs. Conversely, higher temperatures ($T > 1$) encourage diversity by sharpening the distribution.

For this work, we set $T = 0.1$ to prioritize accuracy and precision in the generated explanations, minimizing randomness and hallucinations. This setting is particularly advantageous for tasks such as technical code explanations and systematic predictions, where deterministic and reliable outputs are essential. While reducing randomness limits creativity, it ensures consistency and relevance in outputs, aligning with the high-stakes nature of security patch detection. Dropout is set to 0.5 to balance regularization and generalization, further stabilizing model performance across training iterations.

5 EXPERIMENT RESULTS

5.1 Overall performance of LLMDA

In this section, we evaluate the performance of LLMDA and compare against the selected baselines across the PatchDB and SPI-DB datasets. Table 1 reports the performance measurements on different metrics.

Table 1. Performance metrics (%) on security patch detection

Method	Dataset	AUC	Accuracy	F1	+Recall	-Recall
TwinRNN [47]	PatchDB	66.50	62.10	45.12	46.35	54.37
	SPI-DB	55.10	57.40	47.25	48.00	52.10
GraphSPD [44]	PatchDB	78.29	71.55	54.73	75.17	79.67
	SPI-DB	63.04	64.20	48.42	60.29	65.33
GPT (v3.5)	PatchDB	50.01	50.20	52.97	49.28	50.67
	SPI-DB	49.83	48.15	42.19	44.70	55.20
Vulfixminer [57]	PatchDB	71.39	69.40	64.55	55.72	77.03
	SPI-DB	68.04	66.55	54.42	68.14	62.04
CodeT5 [50]	PatchDB	71.00	68.85	63.73	54.98	76.18
	SPI-DB	72.88	69.25	56.77	65.45	68.75
Fine-Tuned CodeT5+ [49]	PatchDB	81.23	77.90	75.14	77.21	84.90
	SPI-DB	66.45	68.05	55.78	68.10	75.12
Fine-Tuned LLaMa-7b [5]	PatchDB	83.15	79.30	76.98	78.90	86.23
	SPI-DB	68.20	70.10	57.02	70.34	78.21
LLMDA	PatchDB	84.49 (± 0.51)	80.75 (± 0.45)	78.19 (± 0.37)	80.22 (± 0.21)	87.33 (± 0.24)
	SPI-DB	68.98 (± 0.27)	71.25 (± 0.30)	58.13 (± 0.33)	70.94 (± 0.13)	80.62 (± 0.22)

LLMDA demonstrates consistent and robust performance in detecting security patches and distinguishing non-security patches. On the PatchDB dataset, LLMDA achieves +Recall and -Recall of 80% and 87%, respectively, reflecting its balanced ability to detect security patches and exclude non-security ones. Although the performance on SPI-DB is lower, it remains consistent across both classes, underscoring the model's generalizability to different datasets.

The superior performance of LLMDA compared to baseline methods (cf. Table 1) can be attributed to its ability to capture semantic context through multi-modal inputs. For example, on the PatchDB dataset, LLMDA significantly outperforms token-driven neural network approaches such as VulfixMiner, TwinRNN, and GPT 3.5 across all metrics, with improvements ranging from ∼18% to ∼24% in AUC. These gains highlight the effectiveness of leveraging LLM-generated explanations and PT-Former for integrating code and textual inputs, enabling LLMDA to better capture the intent and context of patches compared to token- or sequence-based methods that rely solely on syntactic features.

Compared to the state-of-the-art GraphSPD, LLMDA achieves notable improvements on the PatchDB dataset: 6, 9.2, 23, 5, and 8 percentage points in AUC, Accuracy, F1, +Recall, and -Recall, respectively. These improvements stem from LLMDA 's ability to address GraphSPD's limitations, such as its reliance on local code segment analysis. By incorporating LLM-generated explanations, LLMDA bridges the gap between code changes and their broader semantic context, while PT-Former aligns multi-modal embeddings to capture inter-functional and inter-modular dependencies. The SBCL mechanism further enhances the model's robustness by refining classification boundaries, particularly for challenging edge cases.

However, certain metrics, such as AUC on SPI-DB, are slightly lower compared to baselines like CodeT5. This discrepancy can be explained by the characteristics of SPI-DB, which contains fewer descriptive commit messages and a higher prevalence of imbalanced features. CodeT5's reliance on token-level patterns provides a marginal advantage in such scenarios. Despite this, LLMDA demonstrates substantial gains in other critical metrics such as F1 (+10%), +Recall (+10%), and -Recall (+15%) on SPI-DB, indicating its superior ability to accurately identify security patches while minimizing false positives and negatives. These trade-offs suggest that LLMDA prioritizes real-world security concerns, where reducing false negatives is often more critical than optimizing AUC alone.

To ensure a fair and robust evaluation, we compare LLMDA with fine-tuned versions of CodeT5+ and LLaMa-7b, as these models represent stronger baselines with comparable deployment costs. Both models are fine-tuned on the PatchDB and SPI-DB datasets using the same training configuration as LLMDA, including the AdamW optimizer, learning rate, and batch sizes.

The results indicate that LLMDA consistently outperforms both fine-tuned CodeT5+ and LLaMa-7b across all metrics. The improvements are particularly evident in the F1 score and +Recall, showcasing LLMDA's superior ability to identify security patches. This can be attributed to LLMDA's unique design, which integrates multi-modal inputs (code, descriptions, and explanations) and employs the PT-Former module for precise alignment and fusion of embeddings.

While fine-tuned LLaMa-7b demonstrates competitive performance, its reliance on a single modality limits its ability to capture the nuanced interactions between code and textual descriptions. Similarly, fine-tuned CodeT5+ struggles to generalize across datasets due to its less comprehensive representation of multi-modal inputs.

[RQ-1] ☛ LLMDA *is effective in detecting security patches. With an F1 score at 78.19%,* LLMDA *demonstrates a well-balanced performance: our model can concurrently attain high precision and high recall. Specifically, we achieved a new state-of-the-art performance in identifying both security patches (+Recall) and recognizing non-security patches (-Recall). Comparison experiments further confirm that* LLMDA *is superior to the baselines and is consistently high-performing across the datasets and across the metrics.*

5.2 Contributions of key design decisions

In this section we investigate the impact of key design choices on the overall performance of LLMDA. To that end, we perform an ablation study on :

- *Inputs*: Compared to prior works, LLMDA innovates by considering two additional inputs, namely an LLM-generated explanation of the code changes as well as an instruction. What performance gain do we achieve thanks to these inputs?
- *Representations:* A major contribution of the LLMDA design is the *PT-Former* module, which enables to align and concatenate bimodal input representations belonging to different embedding spaces. What performance gap is filled by PT-Former?
- *Classifiers:* LLMDA relies on stochastic batch contrastive learning to enhance its discriminative power, in particular for samples that are close to the decision boundaries of security relevance. To what extent does SBCL maximize LLMDA's performance?

To answer the aforementioned sub-questions, we build variants of LLMDA where different components are removed. We then compute the performance metrics of each variant and compare them against the original LLMDA.

5.2.1 ***Impact of LLM-generated explanations.*** We build a variant LLMDA_{EX-} by discarding the explanation part like the following example, "PATCH [CLS] ~~EXPLANATION~~ [CLS] DESCRIPTION [CLS] INSTRUCTION". It allows us to investigate the impact of the explanation we generate by an LLM and if the explanation encodes a unique or crucial context that could support the approach. Table 2 reports the performance results achieved in this ablation study.

Table 2. Performance (%) of LLMDA_{EX-} (without the LLM-generated explanations)

Model	Dataset	AUC	Accuracy	F1	+Recall	-Recall
LLMDA_{EX-}	PatchDB	83.24 (↓ 1.25)*	79.80 (↓ 0.95)*	76.73 (↓ 1.46)*	79.01	86.09
	SPI-DB	68.27 (↓ 0.71)*	70.55 (↓ 0.70)*	57.57 (↓ 0.56)*	70.23	80.07

* (↓ *x.xx*) indicates the performance drop compared to LLMDA.

It is noticeable that the performance of LLMDA_{EX-} is consistently lower across the various metrics and across the datasets. These findings highlight the significance of LLM-generated explanations in enhancing the model's predictive capabilities.

5.2.2 ***Impact of instruction.*** We conduct an ablation study based on a variant, LLMDA_{IN-} by removing the instruction part from the original sequence like the following example, "PATCH [CLS] EXPLANATION [CLS] DESCRIPTION [CLS] ~~INSTRUCTION~~". By doing so, we expect to confirm the impact of our instruction. The performance of this variant on the PatchDB and SPI-DB datasets is reported in Table 3.

Table 3. Performance (%) of LLMDA_{IN-} (without the designed instruction)

Model	Dataset	AUC	Accuracy	F1	+Recall	-Recall
LLMDA_{IN-}	PatchDB	82.51 (↓ 1.98)	79.25 (↓ 1.50)	76.14 (↓ 2.05)	78.55	85.64
	SPI-DB	67.93 (↓ 1.05)	70.30 (↓ 0.95)	57.25 (↓ 0.88)	69.90	79.62

Again, we note that the performance drops compared to LLMDA. It even appears that, without the instruction, the performance drop is slightly more important than when the model does not include the LLM-generated explanations. These findings underscore the importance of the label-wise design

decision based on explicitly adding an instruction among the inputs for embedding to enhance the model's performance for security patch detection.

5.2.3 ***Impact of PT-Former.*** To evaluate the significance of the *PT-Former* module, we designed a variant, LLMDA$_{PT-}$, in which the *PT-Former* space alignment and representation combination module is removed. In this variant, a simpler approach is used to generate a unified embedding space for all inputs. Specifically, we concatenate all input embeddings directly as follows:

$$E = E_p \oplus E_{expl} \oplus E_{desc} \oplus E_{inst} \tag{13}$$

Here, E_p, E_{expl}, E_{desc}, and E_{inst} represent the embeddings of the patch, explanation, description, and instruction, respectively, and $\oplus$ denotes the concatenation operation.

Table 4. Performance (%) of LLMDA$_{PT-}$ (without the *PT-Former* module)

Model	Dataset	AUC	Accuracy	F1	+Recall	-Recall
LLMDA$_{PT-}$	PatchDB	81.45 (↓ 3.04)	77.60 (↓ 3.15)	73.56 (↓ 4.63)	75.12	84.34
	SPI-DB	64.89 (↓ 4.09)	68.20 (↓ 3.05)	56.34 (↓ 1.79)	68.45	74.89

The performance results for LLMDA$_{PT-}$ are presented in Table 4. Compared to the original LLMDA, LLMDA$_{PT-}$ demonstrates a consistent decline across all metrics. On PatchDB, the removal of PT-Former results in a 3.04% drop in AUC and a 4.63% decrease in F1-score. While +Recall and -Recall remain relatively stable, their marginal reductions indicate a diminished ability to correctly classify both security-relevant and non-relevant patches. On SPI-DB, the impact of PT-Former's removal is more pronounced in AUC and F1, with a decline of 4.09% and 1.79%, respectively.

These results highlight the critical role of PT-Former in effectively aligning and integrating multi-modal embeddings. By relying solely on concatenation, LLMDA$_{PT-}$ fails to capture the nuanced relationships between different input modalities, such as inter-modular and inter-functional dependencies. PT-Former's advanced attention mechanisms are essential for achieving state-of-the-art performance, as they enhance the interaction between inputs and preserve crucial semantic information. This experiment underscores the importance of PT-Former's design in enabling LLMDA to outperform its variants and existing baselines.

5.2.4 ***Impact of SBCL.*** In LLMDA we designed SBCL to optimize the model's ability to discern different patterns between positive (security patches) and negative (non-security patches) examples more effectively. Figure 4 illustrates the ambition: after PT-Former learns the representations, the embedding subspaces of security and non-security patches will certainly intersect on some "difficult" samples. SBCL is designed to find the optimum decision boundary. To assess the importance of SBCL, we design a variant, LLMDA$_{SBCL-}$, where we directly feed the embeddings processed by *PT-Former* into the fully connected layer (i.e., without the stochastic batch contrastive learning step).

Table 5 presents the performance results of LLMDA$_{SBCL-}$. Compared to the original LLMDA, the performance drop is noticeable. Despite the relatively small proportion of the semantic space at the intersection between the subspaces of security and non-security patches, SBCL enables achieving 1-3 percentage points improvement on the different metrics.

[RQ-2] ☛ *The ablation study results reveal that each of the key design decisions contributes noticeably to the performance of* LLMDA. *In particular, without the* PT-Former *module* LLMDA *would lose about 8 percentage points in F1.*

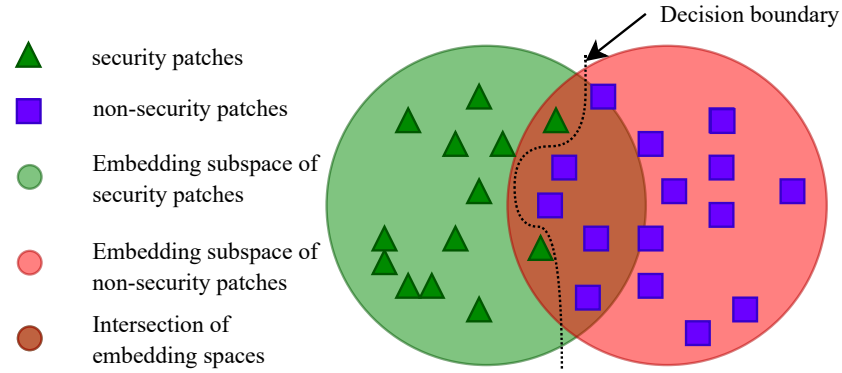


Fig. 4. Illustration of embedding subspaces of security/non-security patches for contrastive learning

Table 5. Performance (%) of LLMDA$_{SBCL-}$ (without contrastive learning)

Model	Dataset	AUC	Accuracy	F1	+Recall	-Recall
LLMDA$_{SBCL-}$	PatchDB	82.93 (↓ 1.56)	79.85 (↓ 0.90)	76.45 (↓ 1.74)	78.72	85.81
	SPI-DB	67.43 (↓ 1.55)	70.80 (↓ 0.45)	56.61 (↓ 1.52)	69.45	79.10

5.3 Discriminative power of LLMDA representations

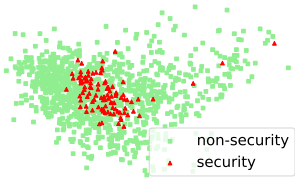
In this section, we investigate to what extent the representations obtained with LLMDA are indeed enabling a good separation of security and non-security patches in the embedding space. To that end, we consider two separate evaluations: the first attempts to visualize the embedding space of LLMDA and compares it against baselines, including GraphSPD (i.e., the state of the art), TwinRNN, Vulfixminer, and CodeT5; the second qualitatively assesses two case studies.

5.3.1 ***Visualization of embedding spaces.*** We consider 1,000 random patches from our PatchDB dataset. We then collect their associated embeddings from LLMDA and baselines (ie., GraphSPD, TwinRNN, Vulfixminer, and codeT5) and apply principal component analysis (PCA) [33]. Given the imbalance of the dataset, the drawn samples are largely non-security patches, while security patches are fewer. Figure 5 presents the PCA visualizations of the representations.

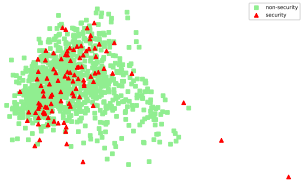
In addition, we also provide virtualization (as shown in Fig. 6) of CodeT5 and LLMDA on the test set to indicate the difference of these two models.

We observe from the distribution of data points that LLMDA can effectively separate the two categories (i.e., security and non-security patches), in contrast to the incumbent state-of-the-art, GraphSPD and other baselines (ie., TwinRNN, Vulfixminer, and codeT5 as shown in Figure 5). This finding suggests that the representations of LLMDA are highly relevant for the task of security patch detection.

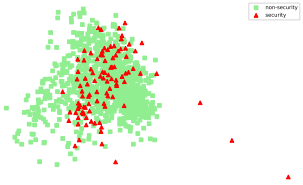
5.3.2 Case studies. Table 6 presents 2 examples to illustrate the difference between LLMDA and baselines (ie., GraphSPD, TwinRNN, Vulfixminer, and CodeT5) in terms of what the representations can capture, and potentially explain the reason LLMDA was successful on these cases while GraphSPD (previously the state-of-the-art work) was not. For our classification task, we have two labels: security (0) and non-security (1). For LLMDA, we can directly consider the label name in the instruction. Thus, we compute the attention map between security and the tokens in the patch, the explanation, and the description. For GraphSPD, however, since there is no real label name involved in the training and inference phases, we compute the attention score between the words in the patch and the number "0" or "1". To simplify the analysis, we only highlight, in Table 6, tokens for which the similarity score is higher than 0.5.



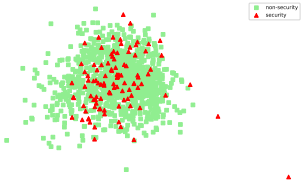
(a) Embeddings yielded by GraphSPD



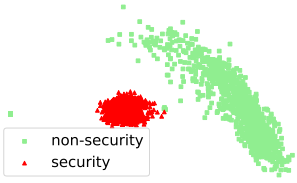
(b) Embeddings yielded by TwinRNN



(c) Embeddings yielded by Vulfixminer

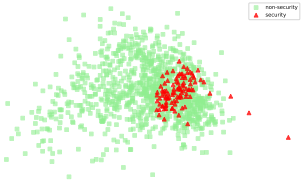


(d) Embeddings yielded by CodeT5

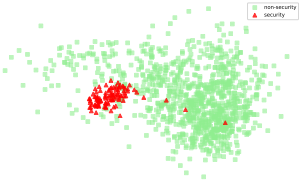


(e) Embeddings yielded by LLMDA

Fig. 5. PCA visualizations of security and non-security patch embeddings by baselines and LLMDA.



(a) Embeddings yielded by CodeT5



(b) Embeddings yielded by LLMDA

Fig. 6. PCA visualizations of security and non-security patch embeddings by CodeT5 and LLMDA on test set.

As shown in the examples, LLMDA generally assigns high similarity scores to security-related aspects, suggesting a detection capability that nuances between tokens. For example, in the sgminer.c patch, LLMDA gives high scores to *realloc* and *mutex_init*, indicating a finer sensitivity to potential security implications within these code parts. Similarly, in the krb5/auth_context.c patch, the use of *memset* for initializing authenticator memory is scored high in LLMDA, reflecting its more acute recognition of security practices.

In contrast, since GraphSPD is graph-based, it focuses on the patch itself. For the same patch cases, GraphSPD can even give very high attention scores for non-security labels. For example, *krb5_auth_con_init* is given a 0.6 score for "non-security" and *ALLOC* is given a 0.67 attention score towards non-security as well. These scores may justify many failures of GraphSPD in the security patch detection task.

Apart from GraphSPD, we also compute the attention maps for TwinRNN, Vulfixminer, and CodeT5. As shown in Table 6, Vulfixminer and CodeT5 exhibit similar behaviors, highlighting tokens such as 'mutex_init(&pool->pool_lock)', 'krb5_auth_con_init', and 'ALLOC'. TwinRNN,

Table 6. Attention scores for security and non-security labels by GraphSPD and LLMDA on two sample patches

GraphSPD (failed cases)		LLMDA (successful cases)		
Patch (*non-formatted token sequence*)	Developer Description	Patch (*non-formatted token sequence*)	Explanation	Description
diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@@ -518,7 +518,7 @@ struct pool *add_pool(void)*(non-security score = 0.65)*sprintf(buf, "Pool %d", pool->_no);pool->poolname = strdup(buf);-\tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+\tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&pool->pool_lock); if(unlikely(pthread_cond_init(&pool->cr_cond, NULL)))	Fixed missing realloc removed by mistake	diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@@ -518,7 +518,7 @@ struct pool *add_pool(void)sprintf(buf, "Pool %d", pool->_no);pool->poolname = strdup(buf);-\tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+\tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2)); *(security core = 0.57)* pools[total_pools++] = pool;mutex_init*(security core = 0.50)*(&pool->pool_lock); if(unlikely(pthread_cond_init(&pool->cr_cond, NULL)))	Modified sgminer.c, adjusted realloc usage*(security core = 0.67)* with explicit type casting*(security core = 0.63)*	Fixed missing realloc*(security core = 0.63)* removed by mistake
diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0edea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init*(non-security score = 0.60)*(krb5_context context, ALLOC*(non-security score = 0.67)*(p->authenticator, 1); if (!p->authenticator) return ENOMEM; + memset (p->authenticator, 0, sizeof(*p->authenticator)); p->flags = KRB5_AUTH_CONTEXT_DO_TIME;	zero authenticator	diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0edea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,7 +53,7 @@ krb5_auth_con_init(krb5_context context, ALLOC(p->authenticator, 1); if (!p->authenticator) return ENOMEM; + memset (p->authenticator, 0, sizeof(*p->authenticator)); *(security core = 0.59)* p->flags = KRB5_AUTH_CONTEXT_DO_TIME;	Added memset to initialize 'authenticator' memory in krb5_auth_con_init function*(security core = 0.84)*.	zero authenticator *(security core = 0.77)*

TwinRNN (failed cases)	Vulfixminer (failed cases)	CodeT5 (failed cases)
Patch (*non-formatted token sequence*)	Patch (*non-formatted token sequence*)	Patch (*non-formatted token sequence*)
diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool->_no);pool->poolname = strdup(buf);-\tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2))*(non-security score = 0.66)*;+\tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&pool->pool_lock) *(non-security score = 0.52)*; if(unlikely(pthread_cond_init(&pool->cr_cond, NULL)))	diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool->_no);pool->poolname = strdup(buf);-\tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+\tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&pool->pool_lock) *(non-security score = 0.59)*; if(unlikely(pthread_cond_init(&pool->cr_cond, NULL)))	diff -git a/sgminer.c b/sgminer.cindex a7dd3ab3..08697cd0 100644-- a/sgminer.c+++ b/sgminer.c@@ -518,7 +518,7 @@ struct pool *add_pool(void) sprintf(buf, "Pool %d", pool->_no);pool->poolname = strdup(buf);-\tpools = realloc(pools, sizeof(struct pool *) (total_pools + 2));+\tpools = (struct **pool **)realloc(pools, sizeof(struct pool *) * (total_pools + 2));pools[total_pools++] = pool;mutex_init(&pool->pool_lock) *(non-security score = 0.52)*; if(unlikely(pthread_cond_init(&pool->cr_cond, NULL)))
diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0edea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init*(non-security score = 0.55)*(krb5_context context, ALLOC*(non-security score = 0.63)*(p->authenticator, 1); if (!p->authenticator) return ENOMEM; + memset (p->authenticator, 0, sizeof(*p->authenticator)); p->flags = KRB5_AUTH_CONTEXT_DO_TIME;	diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0edea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init*(non-security score = 0.73)*(krb5_context context, ALLOC*(non-security score = 0.56)*(p->authenticator, 1); if (!p->authenticator) return ENOMEM; + memset (p->authenticator, 0, sizeof(*p->authenticator)); p->flags = KRB5_AUTH_CONTEXT_DO_TIME;	diff -git a/lib/krb5/auth_context.c b/lib/krb5/auth_context.c index 0edea5418..3cba484e1 100644 -- a/lib/krb5/auth_context.c +++ b/lib/krb5/auth_context.c @@ -53,6 +53,7 @@ krb5_auth_con_init*(non-security score = 0.64)*(krb5_context context, ALLOC(p->authenticator, 1)*(non-security score = 0.68)*; if (!p->authenticator) return ENOMEM; + memset (p->authenticator, 0, sizeof(*p->authenticator)); p->flags = KRB5_AUTH_CONTEXT_DO_TIME;

on the other hand, demonstrates attention patterns more aligned with GraphSPD. These models fail due to several reasons. TwinRNN, like GraphSPD, emphasizes structural relationships in the patch but lacks semantic understanding, leading to poor differentiation of security-critical tokens. For instance, it assigns high scores to 'ALLOC' in the 'krb5_auth_con_init' patch but overlooks 'memset', which has a clear security implication in ensuring memory safety. Similarly, Vulfixminer and CodeT5 focus on surface-level patterns and overgeneralize their relevance, treating tokens like 'mutex_init' as equally important in both security and non-security contexts. This results in a failure to prioritize tokens with actual security implications, such as explicit casting in 'realloc' usage in the 'sgminer.c' patch or memory initialization with 'memset' in the 'krb5_auth_con_init' patch. Moreover, none of these models effectively integrates domain-specific knowledge to capture

the subtle semantics of secure coding practices, which limits their ability to identify security-critical patches. In contrast, LLMDA excels in capturing these nuances by leveraging contextual and domain-aware information, as reflected in its superior attention scores in Table 6.

[RQ-3] ☛ *The design of* LLMDA *leads to patch representations that enable enhanced ability over GraphSPD and other baselines (TwinRNN, Vulfixminer, CodeT5) in effectively differentiating between security and non-security patches on the embedding spaces. Our analysis of sample cases shows that* LLMDA *assigns high attention scores to tokens associated with security-related aspects, making it effective for accurately identifying security patches.*

5.4 Robustness of LLMDA

A model is accepted as robust if it performs strongly on datasets that differ from the training data. For our study task, robustness should ensure reliable predictions on unseen patches. We assess the robustness of LLMDA and baselines (ie., GraphSPD, TwinRNN, Vulfixminer, and CodeT5) by training them against the PatchDB and testing against the samples from the FFmpeg dataset used to construct the benchmark for Devign [59] vulnerability detector. This test data includes 13,962 data points, consisting of 8,000 security-related and 5,962 non-security-related patches. The selection of the FFmpeg dataset is motivated by its coverage of a wide range of vulnerabilities.

Table 7. Performance (%) of GraphSPD, LLMDA, and baselines on unseen patches from FFmpeg dataset [59]. Numbers in parentheses (↓X) denote performance drops relative to PatchDB cross-validation.

Method	Accuracy	Precision	AUC	Recall	+Recall	-Recall	F1
TwinRNN	40.23	45.10	41.50	34.75	34.75 (↓42.85)	49.40	38.12 (↓13.50)
Vulfixminer	41.85	47.00	43.12	35.90	35.90 (↓41.90)	50.50	39.10 (↓12.80)
CodeT5	45.12	50.45	46.75	39.00	39.00 (↓38.45)	53.20	43.75 (↓11.59)
GraphSPD	43.65	51.15	44.81	36.88	36.88 (↓38.29)	52.75	42.86 (↓11.74)
LLMDA	66.78	72.70	66.69	67.30	67.30 (↓12.92)	66.09	69.89 (↓8.30)

* (↓ *x.xx*) indicates performance drop compared to LLMDA's cross-validation on PatchDB (cf. Table 1).

Table 7 summarizes the performance results of LLMDA, GraphSPD, and additional baselines on the unseen dataset. Overall, LLMDA demonstrates superior robustness compared to all baselines. It achieves significantly higher scores across all metrics, with a notable advantage of about 20 percentage points in precision over GraphSPD and a greater performance gap with other baselines. The results further highlight the performance drop when testing on unseen data compared to cross-validation on PatchDB. For example, LLMDA loses approximately 13 percentage points in +Recall, while GraphSPD shows a more substantial drop of 38 percentage points.

Moreover, the token-based and sequential baselines (TwinRNN, Vulfixminer, CodeT5) exhibit the lowest robustness. These methods fail to capture cross-project and unseen patterns due to their reliance on localized or syntactic features. By contrast, LLMDA effectively leverages multi-modal inputs, semantic explanations, and alignment strategies to generalize to diverse datasets. These findings underscore LLMDA's ability to deliver reliable predictions on unseen patches, setting a new standard for robustness in security patch detection.

[RQ-4] ☛ *Experiments on unseen patches clearly demonstrate that* LLMDA *is more robust than GraphSPD and prior baselines. In terms of identifying security patches,* LLMDA*'s performance drop is about threefold smaller compared to the prior state-of-the-art GraphSPD under the same experimental settings.*

5.5 Effect of Different Models and Different Instructions

Table 8. Performance metrics (%) on security patch detection with different instructions

Model	Instruction Type	Dataset	AUC	Accuracy	F1	+Recall	-Recall
LLMDA (Gemini 2.0 Flash)	Task-Specific	PatchDB	76.45	72.00	69.23	72.45	80.34
	Detailed	PatchDB	74.67	70.50	68.00	70.12	79.12
	Concise	PatchDB	73.12	69.00	66.34	69.00	77.89
	Direct Explanation	PatchDB	75.78	71.50	68.56	71.23	79.67
	Task-Specific	SPI-DB	64.56	66.50	55.67	65.34	72.12
	Detailed	SPI-DB	63.12	65.00	54.12	64.00	71.00
	Concise	SPI-DB	62.45	64.30	53.67	63.23	70.12
	Direct Explanation	SPI-DB	63.89	65.70	54.67	64.45	71.67
LLMDA (Claude 3.5 Sonnet)	Task-Specific	PatchDB	81.12	77.20	74.45	78.34	84.23
	Detailed	PatchDB	79.56	75.80	72.67	76.45	82.67
	Concise	PatchDB	77.89	74.10	71.12	75.00	81.12
	Direct Explanation	PatchDB	80.12	76.50	73.45	77.12	83.56
	Task-Specific	SPI-DB	67.34	69.80	57.34	68.89	76.45
	Detailed	SPI-DB	66.12	68.60	56.00	67.12	75.12
	Concise	SPI-DB	65.34	67.90	55.34	66.00	74.00
	Direct Explanation	SPI-DB	66.89	69.10	56.34	67.67	75.89
LLMDA (Mixtral 8x7B)	Task-Specific	PatchDB	76.45	72.40	68.34	72.10	79.67
	Detailed	PatchDB	74.78	71.00	67.12	70.45	78.34
	Concise	PatchDB	72.67	69.80	65.00	69.12	76.89
	Direct Explanation	PatchDB	75.23	71.80	67.89	71.34	78.90
	Task-Specific	SPI-DB	64.12	66.70	54.23	64.67	71.89
	Detailed	SPI-DB	63.34	65.90	53.45	63.89	70.78
	Concise	SPI-DB	62.00	64.50	52.67	63.00	70.00
	Direct Explanation	SPI-DB	63.45	65.40	53.89	64.12	71.34
LLMDA(gpt3.5-turbo)	Task-Specific	PatchDB	84.49 (± 0.51)	80.75 (± 0.45)	78.19 (± 0.37)	80.22 (± 0.21)	87.33 (± 0.24)
	Detailed	PatchDB	82.34 (± 0.48)	79.50 (± 0.40)	75.89 (± 0.33)	78.34 (± 0.20)	85.67 (± 0.22)
	Concise	PatchDB	80.56 (± 0.45)	77.95 (± 0.38)	73.34 (± 0.30)	76.12 (± 0.18)	84.12 (± 0.21)
	Direct Explanation	PatchDB	81.78 (± 0.47)	78.60 (± 0.39)	74.45 (± 0.32)	77.45 (± 0.19)	84.89 (± 0.22)
	Task-Specific	SPI-DB	68.98 (± 0.27)	71.25 (± 0.30)	58.13 (± 0.33)	70.94 (± 0.13)	80.62 (± 0.22)
	Detailed	SPI-DB	67.89 (± 0.26)	70.80 (± 0.29)	56.78 (± 0.30)	68.45 (± 0.12)	79.45 (± 0.21)
	Concise	SPI-DB	66.45 (± 0.25)	69.40 (± 0.27)	55.23 (± 0.28)	67.12 (± 0.11)	78.67 (± 0.20)
	Direct Explanation	SPI-DB	67.12 (± 0.26)	70.00 (± 0.28)	56.00 (± 0.29)	68.00 (± 0.12)	79.00 (± 0.21)

Table 9. Comparison of Instruction Types for Explanation Generation

Instruction Type	Example Instruction	Characteristics
Task-Specific	Choose the correct option to the following question: Is the patch security related or not? Choices: (0) security (1) non-security.	Tailored to the classification task; generates explanations that directly address the patch's relevance to security, making it highly actionable for the downstream model.
Detailed	Could you explain the purpose of this patch and describe why the change was made?	Provides a detailed explanation of both the intent and the functional impact of the code change, offering comprehensive context for the model to learn from.
Concise	Summarize the purpose of this patch in one sentence.	Generates brief and focused explanations; while less detailed, these instructions are efficient and provide essential insights into the patch.
Direct Explanation	Explain what happened inside the code changes.	Focuses on describing the specific modifications made in the patch; emphasizes technical changes without necessarily linking them to broader intent or security relevance.

Table 8 and Table 9 indicates that the quality and effectiveness of explanations generated for LLMDA are significantly influenced by both the choice of model and the type of instruction provided.

To evaluate these factors, we conducted a systematic ablation study using five different large language models (LLMs) (Gemini 2.0 Flash [28], Claude 3.5 Sonnet [3], Mixtral 8x7B [10], Gemma 7B [18], and Llama3 70B [2]) and four types of instructions (Task-Specific, Detailed, Concise, Direct Explanation). Below, we provide an intrinsic analysis of how these variations impact security patch detection.

Effect of Different Instructions. The instruction type used to generate explanations significantly affects their quality and relevance, which in turn impacts the downstream performance of LLMDA.

Task-Specific Instructions. For instance: *"Choose the correct option to the following question: Is the patch security related or not? Choices: (0) security (1) non-security."* These instructions align closely with the binary classification task, generating explanations that directly assess the security relevance of patches. The tight alignment ensures better feature extraction and improved context understanding, leading to superior metrics such as F1 and +Recall. Recent studies [5, 51] have demonstrated the effectiveness of task-specific prompts in enhancing downstream performance.

Detailed Instructions. For example: *"Could you explain the purpose of this patch and describe why the change was made?"* These prompts produce comprehensive explanations, offering both the intent and functional impact of the code change. While informative, detailed instructions can include extraneous information not directly relevant to the classification task, slightly diluting performance compared to task-specific prompts. Nevertheless, the additional context is particularly useful for datasets with sparse metadata, such as SPI-DB [60].

Concise Instructions. For instance: *"Summarize the purpose of this patch in one sentence."* Concise instructions prioritize brevity, generating short explanations that provide essential insights. While efficient, these instructions often lack the depth required to fully capture the intent and implications of the patch, resulting in lower +Recall and F1 scores [4].

Direct Explanation Instructions. For example: *"Explain what happened inside the code changes."* These prompts focus on describing the specific modifications made in the patch without linking them to broader security implications. While useful for technical analysis, the lack of semantic insights necessary for security classification results in comparatively lower performance, consistent with findings in [35].

Effect of Different Models. The choice of LLM affects the quality of generated explanations, including relevance, coherence, and semantic richness, which significantly impacts LLMDA 's classification performance.

Gemini 2.0 Flash and Claude 3.5 Sonnet. These models perform consistently well, with Claude 3.5 Sonnet achieving the strongest results when paired with task-specific or detailed instructions. Their advanced contextual understanding enables them to generate high-quality explanations that balance detail and relevance.

Mixtral 8x7B and Gemma 7B. These models exhibit moderate performance across all instructions. Their relatively smaller architectures limit their ability to generate nuanced semantic explanations, leading to lower performance on complex datasets like SPI-DB.

Llama3 70B. Despite its size, Llama3 70B shows variability depending on the instruction type. It performs well with task-specific instructions but struggles with concise prompts, likely due to over-reliance on syntactic patterns.

LLMDA. As the proposed model, LLMDA consistently outperforms all other LLMs across instructions. Task-specific instructions paired with LLMDA yield the highest performance, demonstrating its ability to leverage semantically aligned explanations and integrate them effectively into its multi-modal architecture.

Dataset-Specific Insights. **PatchDB.** The diversity of repositories in PatchDB highlights the importance of high-quality explanations. Task-specific and detailed instructions enable LLMDA to generalize effectively across various patch styles. **SPI-DB.** The sparse commit messages and imbalanced features in SPI-DB pose challenges for all configurations. However, detailed instructions mitigate these issues by providing additional context, while task-specific instructions remain the most effective overall due to their alignment with the classification objective.

Conclusion *The section discusses that both the choice of LLM and the instruction type significantly influence the performance of* LLMDA. *Task-specific instructions paired with high-capability models like Claude 3.5 Sonnet and Gemini 2.0 Flash yield superior results, highlighting the importance of aligning explanations with the classification task. Detailed instructions improve robustness in sparse datasets like SPI-DB, while concise and direct explanations underperform due to their lack of contextual depth. These findings emphasize the critical role of prompt engineering and model selection in enhancing explanation quality and downstream performance.*

5.6 *Extended Evaluation of Instruction*

We evaluate the performance by conducting experiments on the following: Instruction Positions and Importance of Instruction.

Importance of Instruction Positions. The position of the instruction within the input sequence significantly affects the model's ability to leverage task-specific guidance. As shown in Table 10, placing the instruction at the end of the sequence yields the highest performance, with an AUC of 84.49% and an F1 score of 78.19%, outperforming configurations where the instruction is placed at the beginning or middle. This result aligns with the transformer architecture's tendency to assign greater attention to later tokens, allowing the end-positioned instruction to better influence the final embedding. Conversely, positioning the instruction at the beginning or middle may dilute its significance as subsequent tokens are processed, leading to a slight decrease in performance. These findings underscore the importance of aligning input design choices with the model's architectural characteristics to maximize the utility of task-specific instructions.

Table 10. Performance (%) with Different Instruction Positions

Configuration	Dataset	AUC	Accuracy	F1	+Recall	-Recall
Instruction at Beginning	PatchDB	82.45	78.60	76.12	78.45	85.89
Instruction at Middle	PatchDB	83.23	79.30	77.00	79.23	86.34
Instruction at End (Original)	PatchDB	**84.49**	**80.75**	**78.19**	**80.22**	**87.33**

Importance of Instruction. Furthermore, as shown in Table 11, we also evaluate the impact of instruction by replacing the instruction with a sentence composed of random words generated from a random sentence Web provider [6]. Here, we take two random sentences as examples: Sentence$_A$: "He stepped gingerly onto the bridge knowing that enchantment awaited on the other side." Sentence$_B$: "When confronted with a rotary dial phone, the teenager was perplexed."

The performance results for these configurations are presented in Table 10. As shown, using random sentences as instructions leads to a noticeable performance drop compared to the original task-specific instruction used in LLMDA. Specifically, both Sentence$_A$ and Sentence$_B$ achieve lower AUC, Accuracy, F1, +Recall, and -Recall metrics than the original configuration.

The results indicate that replacing the task-specific instruction with random sentences introduces noise and misleads the training process. Unlike task-specific instruction, which provides direct

[6] https://randomwordgenerator.com/sentence.php

Table 11. Impact of Instruction

Configuration	Dataset	AUC	Accuracy	F1	+Recall	-Recall
Sentence$_A$	PatchDB	75.34	71.50	70.12	71.45	78.90
Sentence$_B$	PatchDB	76.05	72.00	71.00	72.12	79.50
LLMDA **(Original)**	PatchDB	**84.49**	**80.75**	**78.19**	**80.22**	**87.33**

guidance aligned with the classification objective (security-related or non-security-related patches), random sentences lack semantic relevance and coherence in relation to the task. This misalignment results in a reduced ability of the model to focus on meaningful features and relationships within the data, leading to degraded performance across all metrics.

The experimental results highlight the importance of using instructions in LLMDA. Instruction serves as a guiding mechanism, focusing the model's attention on the critical aspects of the problem and enhancing its ability to learn meaningful representations for classification. In contrast, random sentences introduce irrelevant information, diluting the signal and adversely affecting the model's effectiveness. This underscores the necessity of carefully designed instructions to ensure optimal training and robust model performance.

6 DISCUSSION

6.1 Threats to Validity

Internal validity. A first threat is the quality of the generated patch explanations. Since LLMs may be factually wrong in their descriptions of the code changes or, in contrast, be vastly good for our well-known study datasets, LLMDA performance evaluation may be biased. We mitigate this threat by considering a state-of-the-art LLM and by rigorously analyzing the impact of the generated LLM in an ablation study.

A second threat is the evolving performance of the hosted GPT models. It may prevent reproducibility since this evolution introduces instability, potentially affecting the consistency of results even with identical prompts or instructions.

A third threat lies in the constraint imposed by the input size limitation of 512 tokens. For long patches, LLMDA performs truncation, resulting in the loss of essential information and potentially affecting the accuracy and reliability of the model's predictions.

External validity. A threat is that we rely on PatchDB and SPI-DB datasets, which may not generalize our findings beyond their diverse samples. For example, SPI-DB contains patches from only 2 projects. We mitigate this threat by relying on 2 distinct datasets, PatchDB having samples from over 300 projects. Furthermore LLMDA is natural language-centric and thus our key design choices are programming language-independent.

Another threat stems from the fact that we rely on pre-trained models (CodeT5 and LlaMa-7b) as initial embedders of LLMDA's inputs. These models may actually not be adapted for the task at hand. To mitigate this threat our selection was based on the fact that they were demonstrated in the literature as among the best performing models for related tasks.

Construct validity. A threat is that our experiments do not try various prompts in the *Instruction* input. This may lead to an oversight in properly checking the potential contribution of instruction on the model's performance. We have mitigated this threat by performing an ablation study that shows the impact of the current design. Better prompts may positively increase this impact.

6.2 Limitation

One limitation of the current study is the use of GPT-3.5 (gpt-3.5-turbo-16k-0613) for generating explanations of code changes within the LLMDA framework. While GPT-3.5 has demonstrated strong performance in understanding and explaining code, the release of GPT-4.0 (gpt-4-0613) presents an opportunity for further enhancement.

To illustrate this, consider the following patch (non-security) from the Linux kernel, which modifies socket configurations to enable asynchronous I/O operations:

```
@@ -1950,6 +1950,7 @@ static int xs_local_finish_connecting(struct rpc_xprt *xprt,
        sk->sk_user_data = xprt;
        sk->sk_data_ready = xs_data_ready;
        sk->sk_write_space = xs_udp_write_space;
+   sock_set_flag(sk, SOCK_FASYNC);
        sk->sk_error_report = xs_error_report;
        sk->sk_allocation = GFP_NOIO;

@@ -2136,6 +2137,7 @@ static void xs_udp_finish_connecting(struct rpc_xprt *xprt, struct
     socket *sock)
        sk->sk_user_data = xprt;
        sk->sk_data_ready = xs_data_ready;
        sk->sk_write_space = xs_udp_write_space;
+   sock_set_flag(sk, SOCK_FASYNC);
        sk->sk_allocation = GFP_NOIO;

        xprt_set_connected(xprt);

@@ -2237,6 +2239,7 @@ static int xs_tcp_finish_connecting(struct rpc_xprt *xprt, struct socket
      *sock)
        sk->sk_data_ready = xs_tcp_data_ready;
        sk->sk_state_change = xs_tcp_state_change;
        sk->sk_write_space = xs_tcp_write_space;
+   sock_set_flag(sk, SOCK_FASYNC);
        sk->sk_error_report = xs_error_report;
        sk->sk_allocation = GFP_NOIO;
```

GPT-4.0 provides a more detailed and contextually rich explanation:

> The provided code snippets modify socket configurations in three different functions (xs_local_finish_connecting, xs_udp_finish_connecting, xs_tcp_finish_connecting). They uniformly set the SOCK_FASYNC flag using `sock_set_flag(sk, SOCK_FASYNC);`, enabling asynchronous I/O operations on the sockets involved. This change likely aims to enhance efficiency by allowing non-blocking handling of network events and data reception across different types of connections (local, UDP, TCP).

In contrast, GPT-3.5's explanation is more concise:

> The changes in the code snippets add the `sock_set_flag(sk, SOCK_FASYNC);` line to three different functions (xs_local_finish_connecting, xs_udp_finish_connecting, and xs_tcp_finish_connecting). This modification enables asynchronous (signal-driven) I/O on the sockets, allowing the program to receive signals when I/O events occur, thereby improving network operations' efficiency and responsiveness.

While both explanations are accurate, GPT-4.0's explanation offers a more nuanced understanding of the code changes' implications, which could be crucial for determining the security relevance of a patch. In this case, the patch is likely non-security-related, as it focuses on improving I/O efficiency rather than addressing a security vulnerability.

Future work could explore the integration of GPT-4.0 or other advanced language models to enhance the LLMDA framework's ability to generate detailed explanations and improve the prediction

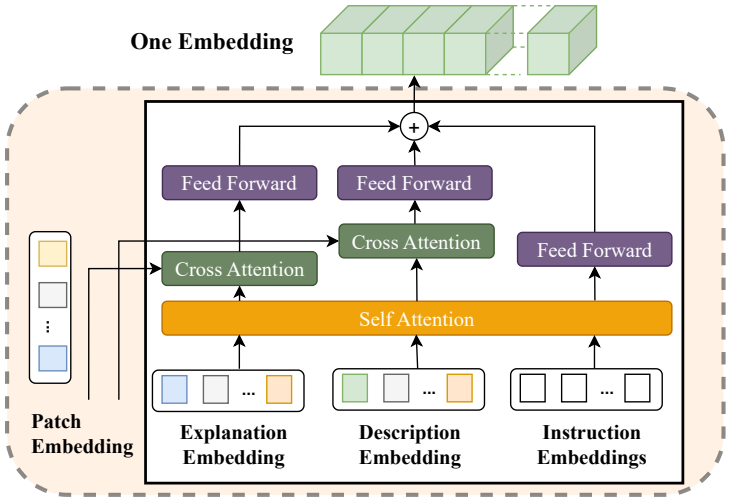


Fig. 7. Cross-Attention Between Patch and Description version of PT-Former

of security patches. This could lead to more accurate and timely detection of silent security patches, thereby enhancing the overall security posture of open-source software systems.

6.3 Leveraging Patch Descriptions for Enhanced Cross-Attention

While the original architecture of LLMDA effectively models the relationships between the code patch and the LLM-generated explanation (E_{expl}), it does not fully utilize the potential insights provided by patch descriptions (E_{desc}) when they are available. To address this limitation, we enhance the architecture by introducing an additional cross-attention mechanism between the patch embedding (E_p) and the description embedding (E_{desc}).

Figure 7 illustrates the updated architecture of the PT-Former module, which now includes cross-attention layers for both the explanation and the description. The updated PT-Former processes the input embeddings as follows: 1) Cross-Attention Between Patch and Explanation ($E_p \leftrightarrow E_{expl}$): This captures the semantic relationship between the code patch and its generated explanation. 2) Cross-Attention Between Patch and Description ($E_p \leftrightarrow E_{desc}$): This models the contextual relevance of the patch description to the code changes. 3) Aggregation: The outputs of the two cross-attention layers are concatenated and passed through feed-forward layers to generate a unified representation of the patch.

In cases where the description (E_{desc}) is missing, a masking mechanism is employed to ensure that the model seamlessly defaults to using only E_p and E_{expl}, maintaining the robustness of the system.

Table 12. Performance (%) with Enhanced Cross-Attention

Model	Dataset	AUC	F1	+Recall	-Recall
Original PT-Former	PatchDB	84.49	78.19	80.22	87.33
Enhanced PT-Former	PatchDB	**85.25**	**79.34**	**81.45**	**88.12**
Original PT-Former	SPI-DB	68.98	58.13	70.94	80.62
Enhanced PT-Former	SPI-DB	**70.12**	**59.45**	**72.18**	**81.87**

6.3.1 Experimental Results. Table 12 summarizes the performance of the enhanced PT-Former architecture compared to the original design. On the PatchDB dataset, the enhanced architecture achieves a 0.76% improvement in AUC and a 1.15% increase in F1, reflecting its ability to leverage additional context from patch descriptions. Similarly, on the SPI-DB dataset, the AUC and F1 scores improve by 1.14% and 1.32%, respectively.

The results demonstrate that incorporating cross-attention between the patch and description embeddings significantly enhances the model's ability to capture the broader context and improve its decision-making process. Importantly, the fallback mechanism ensures that the enhanced architecture remains robust even when descriptions are missing.

Conclusion☛ *These results highlight the effectiveness of leveraging patch descriptions through cross-attention. By incorporating this additional layer,* LLMDA *achieves improved performance across all metrics, demonstrating its adaptability to varying input configurations.*

6.4 Limitations of PT-Former: Dependency on Embedding Models

One notable limitation of PT-Former lies in its dependence on the quality of the underlying code and text embedding models, such as CodeT5+ for code and LLaMA for text. These embeddings form the foundation of PT-Former's input representation, and any inadequacies in the pre-trained models can propagate through the network, impacting the final performance.

Impact of Embedding Quality. The effectiveness of PT-Former in aligning and fusing multi-modal inputs heavily relies on the richness and accuracy of the embeddings generated by these models. If the embeddings fail to capture nuanced semantic or structural information, the subsequent alignment and fusion processes may be constrained. For example:

- **Code Embeddings**: If the pre-trained CodeT5+ model struggles to represent certain programming constructs or interactions, PT-Former's ability to capture inter-functional and inter-modular relationships could be limited.
- **Text Embeddings**: Similarly, LLaMA-generated embeddings might miss critical contextual details or nuances in patch descriptions, reducing the model's overall interpretability and precision.

Generalization Across Domains. Another challenge arises from the generalization capability of the pre-trained models. CodeT5+ and LLaMA are typically fine-tuned on specific datasets, which may not comprehensively cover the diversity of codebases or patch descriptions encountered in real-world scenarios. As a result:

- PT-Former's performance might degrade on unseen or cross-project datasets, where the embedding models fail to generalize effectively.
- The reliance on static embeddings limits PT-Former's ability to dynamically adapt to varying code styles, programming languages, or textual nuances across domains.

6.5 Comparison with CoLeFunDa

While both CoLeFunDa [56] and our approach leverage contrastive learning for silent vulnerability fix identification, the methodologies and contributions differ significantly.

First, CoLeFunDa emphasizes function-level code representation, focusing on CWE classification and exploitability rating alongside vulnerability fix detection. In contrast, our approach targets security patch detection by integrating multi-modal inputs, including code changes, developer-provided descriptions, and LLM-generated explanations. This integration allows LLMDA to capture both syntactic and semantic contexts, addressing limitations of function-level analysis.

Second, CoLeFunDa relies on a contrastive learning framework applied to augmented function slices, without leveraging natural language guidance. Our work innovates by introducing task-specific natural language instructions, which act as guiding mechanisms for contextualizing the data and aligning the model's focus with the security detection task.

Finally, the architectural designs are distinct. CoLeFunDa employs a function change encoder (FCBERT) for its representations, whereas LLMDA utilizes the PT-Former module. PT-Former aligns and fuses multi-modal inputs through cross-attention and self-attention mechanisms, enabling a unified embedding space that enhances generalization across diverse datasets.

These distinctions underscore the unique contributions and novelty of our work, particularly in addressing limitations in prior methods, including CoLeFunDa.

6.6 Model Selection for Initial Embeddings

We utilized CodeT5+ and LLaMa-7b as they can function as an encoder-only model while other models like CodeLlama are not inherently designed to function as an encoder-only model, as they are primarily a decoder-only transformer based on the LLaMA. Embedding models often use this configuration to generate dense vector representations of input text or code.

Furthermore, we conducted additional experiments using CodeLLaMA-13B as a unified embedding model for both code and natural language (NL) inputs. Unlike the original LLMDA, which employs CodeT5+ for code and LLaMa-7b for NL, this setup generates all embeddings from a single model, simplifying the alignment process in PT-Former.

The results, presented in Table 13, show that CodeLLaMA-13B achieves competitive performance but falls slightly short of the original LLMDA setup. Specifically, LLMDA with CodeLLaMA-13B demonstrates a small drop in AUC and F1 across both datasets. Additionally, the training process required more epochs to converge, likely due to the lack of task-specific optimization for code and NL modalities in a unified model.

These findings reinforce the advantages of using specialized models for multi-modal tasks. While CodeLLaMA-13B provides a simpler deployment setup, the combination of CodeT5+ and LLaMa-7b offers better task-specific performance and faster convergence. Future research could explore fine-tuning unified models like CodeLLaMA for enhanced performance in security patch detection.

Table 13. Performance (%) and Training Efficiency Comparison: Original LLMDA vs. CodeLLaMA-13B

Model	Dataset	AUC	F1	+Recall	-Recall	Epochs	Training Time
LLMDA *Original*	PatchDB	**84.49**	**78.19**	**80.22**	**87.33**	20	12.5h
LLMDA *CodeLLaMA−13B*	PatchDB	83.12 (↓ 1.37)	76.78 (↓ 1.41)	78.45	85.67	30	
LLMDA *Original*	SPI-DB	**68.98**	**58.13**	**70.94**	**80.62**	20	
LLMDA *CodeLLaMA−13B*	SPI-DB	67.51 (↓ 1.47)	56.45 (↓ 1.68)	69.01	78.54	30	18h

"LLMDA *Original*" means LLMDA with CodeT5+ to embed codes and LLaMa-7b to embed texts.
"LLMDA *CodeLLaMA−13B*" means LLMDA with LLMDA to embed codes and texts.

6.7 Extended Case Studies on the Impact of Explanations and Instructions

To further illustrate the importance of explanations and instructions in LLMDA, we present two additional case studies involving security patches. These cases highlight how the inclusion of natural language explanations and task-specific instructions contributes to accurate predictions, particularly in challenging scenarios.

6.7.1 Case Study 1: Complex Code Changes with Minimal Documentation. Patch:

```
diff --git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
index c34fa2a..e4fb7a5 100644
--- a/net/ipv4/tcp_input.c
```

```
+++ b/net/ipv4/tcp_input.c
@@ -2342,7 +2342,8 @@ void tcp_ack(struct sock *sk, struct sk_buff *skb, int flag)

     /* Fix potential buffer overflow in TCP options */
-    memcpy(opt, skb->data + tcp_hdrlen, optlen);
+    if (unlikely(optlen > TCP_MAX_OPT_LENGTH))
+        return;
+    memcpy(opt, skb->data + tcp_hdrlen, optlen);
 }
```

Listing 2. A complex security patch with minimal documentation.

Baseline Prediction (GraphSPD): Non-Security
LLMDA **Prediction:** Security
Reason: The patch adds a boundary check to prevent buffer overflows, a critical vulnerability. GraphSPD fails to detect the intent due to the absence of a detailed commit message. However, LLMDA uses an LLM-generated explanation (e.g., "This patch mitigates buffer overflow by adding a boundary check") to capture the semantic significance of the changes. The task-specific instruction further directs the model to prioritize security-related aspects of the patch.

6.7.2 Case Study 2: Misleading Commit Message. Patch:

```
diff --git a/src/http.c b/src/http.c
index abcd123..dcba321 100644
--- a/src/http.c
+++ b/src/http.c
@@ -567,7 +567,8 @@ int validate_http_request(request *req) {
     /* Sanitize user inputs to prevent injection */
-    strcpy(buffer, req->input);
+    if (strlen(req->input) < MAX_LEN) {
+        strncpy(buffer, req->input, MAX_LEN);
+    }
     return validate_buffer(buffer);
 }
```

Listing 3. A patch with a misleading commit message.

Commit Message: "Minor refactor of input validation code."
Baseline Prediction (CodeT5+): Non-Security
LLMDA **Prediction:** Security
Reason: The misleading commit message downplays the significance of the patch, causing CodeT5+ to miss its security implications. LLMDA relies on LLM-generated explanations to identify that the patch introduces input length checks to prevent buffer overflows, a critical security improvement. The instruction guides the model to focus on identifying security-relevant changes.

6.7.3 **Analysis of Findings.** In both cases, LLMDA successfully identifies the security relevance of the patches, while baselines fail. The key differentiators are:

- **Explanations:** LLM-generated explanations bridge the gap between code semantics and human-readable context, enabling the model to understand the intent behind code changes.
- **Instructions:** Task-specific instructions provide explicit guidance for identifying security patches, enhancing the model's decision-making process.

7 RELATED WORK

Our work is related to various research directions in the literature. We discuss three main categories in this section.

7.1 Security Patch Analysis

Patch analysis, after being addressed in the literature of empirical studies and static analysis research, has been increasingly a key application area of machine learning for software engineering [19, 29, 44, 48]. In terms of security patches, Li et al. [19] provided foundational empirical insights into the unique attributes of such patches. Rule-based approaches [16, 53] were then pivotal in demonstrating that the identification of security patches is feasible using common patterns [54, 54]. Afterwards, Wang et al. [48] proceeded to data-driven methodologies with statistical machine learning. RNN-based approaches such as PatchRNN [47] and SPI [60] then revealed that neural networks were key enablers in understanding patches. With ColeFunda [56], researchers proposed to summarize the semantics of patches using git differencing tools. Most recently, GraphSPD [44] achieved state-of-the-art performance by implementing a graph-based approach that focuses on ensuring that the semantics in the code change are effectively captured. In this work, our LLMDA approach employs Large Language Models for semantic analysis of code changes and introduces a multi-modal alignment method to improve the accuracy of security patch detection.

7.2 Deep Learning in Vulnerability Detection and Bug Repair

Deep learning has enabled software engineering research to advance in the automation of the detection of vulnerable code [13, 22, 32]. Most recently, Fu et al. advanced software vulnerability detection by proposing VulExplainer [12] for the classification of vulnerability types using Transformer-based hierarchical distillation. In another direction, Nguyen et al. contributed by identifying vulnerability-relevant code statements through deep learning and clustered contrastive learning [26] and by creating ReGVD [27], a graph neural network model for vulnerability detection. In the era of LLM, CodeAgent [37] was proposed as a multi-agent LLM system for automating code review, addressing tasks such as detecting inconsistencies, identifying vulnerabilities, validating code style, and suggesting revisions. In the area of bug detection and repair, Tang et al. [39] proposed a collaborative bug-finding approach driven by app reviews, utilizing user feedback to identify and address software defects more efficiently. Luo et al. [25] explored the integration of federated learning to ensure data privacy in LLM-based program repair. Pian et al. [30] introduced jLED, a joint learning framework that localizes and edits source code, thereby enhancing the repair process. Yang et al. [55] presented MORepair, a multi-objective fine-tuning method that trains LLMs to repair code more effectively, improving both the quality and efficiency of the repair process.

7.3 Patch representation learning

Reasoning about patches using deep neural networks has attracted significant interest in recent years [11, 23, 31, 38, 40, 50]. Although initial work directly used generic code representation models such as CodeBERT [11], CodeT5 [50], GraphCodeBERT [14] or PLBART [1]. Some recent works, such as CCRep [23], ReconPatch [17], and CCBERT [58] have explored specialized approaches to better capture the semantics of code changes. With LLMDA, our approach attempts to learn specific representations for the task of security patch detection. Our approach, LLMDA, builds on the foundation of leveraging deep neural networks for patch representation, advancing beyond generic models like CodeBERT and CodeT5 by focusing on specialized representation learning tailored specifically for detecting security patches.

8 CONCLUSION

In this work, we proposed a framework, LLMDA, for security patch detection. It implements a language-centric approach to the overall problem of learning to identify silent security patches. First, LLMDA augments patch information with LLM-generated explanations. Then, it builds an embedding

where multi-modal patch information is concatenated with a natural language instruction after the alignment of embedding spaces. Finally, using contrastive learning, it ensures that challenging cases are the decision boundaries are well discriminated. Experimental assessments of LLMDA over two literature datasets demonstrate how LLMDA achieves a new state of performance on the target task. Further ablation studies confirm the contribution of the key design choices as well as the robustness of the trained model.

Open Science: All code, data and results are publicly available in our artifact repository: https://llmda.github.io

9 ACKNOWLEDGMENTS

This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant No. 949014).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[2] Meta AI. 2023. Llama3 70B: Large Language Model for Multimodal Applications. https://ai.meta.com/llama3-70b.

[3] Anthropic. 2023. Claude 3.5 Sonnet: High-Performance Language Model for Complex Tasks. https://www.anthropic.com/claude-3-5-sonnet.

[4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[5] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416* (2022).

[6] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research* 25, 70 (2024), 1–53.

[7] Wenliang Dai, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Junqi Zhao, Weisheng Wang, Boyang Li, Pascale Fung, and Steven Hoi. 2023. InstructBLIP: Towards General-purpose Vision-Language Models with Instruction Tuning. arXiv:2305.06500 [cs.CV]

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] Nesara Dissanayake, Mansooreh Zahedi, Asangi Jayatilaka, and Muhammad Ali Babar. 2022. Why, How and Where of Delays in Software Security Patch Management: An Empirical Investigation in the Healthcare Sector. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (2022), 1–29.

[10] Hugging Face. 2023. Mixtral 8x7B: Multi-Domain Pre-trained Model for Code and Text. https://huggingface.co/mixtral-8x7b.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[12] Michael Fu, Van Nguyen, Chakkrit Kla Tantithamthavorn, Trung Le, and Dinh Phung. 2023. VulExplainer: A Transformer-based Hierarchical Distillation for Explaining Vulnerability Types. *IEEE Transactions on Software Engineering* (2023).

[13] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering* 29, 1 (2024), 4.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[15] Mohammad Hossin and Md Nasir Sulaiman. 2015. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process* 5, 2 (2015), 1.

[16] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–554.

[17] Jeeho Hyun, Sangyun Kim, Giyoung Jeon, Seung Hwan Kim, Kyunghoon Bae, and Byung Jun Kang. 2024. ReConPatch: Contrastive patch representation learning for industrial anomaly detection. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2052–2061.

[18] Gemma AI Labs. 2023. Gemma 7B: Lightweight Model for Language and Code Tasks. https://gemmalabs.ai/models/7b.

[19] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.

[20] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International conference on machine learning*. PMLR, 19730–19742.

[21] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[23] Zhongxin Liu, Zhijie Tang, Xin Xia, and Xiaohu Yang. 2023. Ccrep: Learning code change representations via pre-trained code model and query back. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 17–29.

[24] Ilya Loshchilov and Frank Hutter. 2018. Fixing weight decay regularization in adam. (2018).

[25] Wenqiang Luo, Jacky Wai Keung, Boyang Yang, He Ye, Claire Le Goues, Tegawende F. Bissyande, Haoye Tian, and Bach Le. 2024. When Fine-Tuning LLMs Meets Data Privacy: An Empirical Study of Federated Learning in LLM-Based Program Repair. arXiv:2412.01072 [cs.SE] https://arxiv.org/abs/2412.01072

[26] Van Nguyen, Trung Le, Chakkrit Tantithamthavorn, John Grundy, Hung Nguyen, Seyit Camtepe, Paul Quirk, and Dinh Phung. 2022. An Information-Theoretic and Contrastive Learning-based Approach for Identifying Code Statements Causing Software Vulnerability. *arXiv preprint arXiv:2209.10414* (2022).

[27] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 178–182.

[28] OpenAI. 2023. Gemini 2.0 Flash: Advanced Language Model for Code and Text Understanding. https://www.openai.com/gemini-2-flash.

[29] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 426–437.

[30] Weiguo Pian, Yinghua Li, Haoye Tian, Tiezhu Sun, Yewei Song, Xunzhu Tang, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2025. You Don't Have to Say Where to Edit! jLED–Joint Learning to Localize and Edit Source Code. *ACM Transactions on Software Engineering and Methodology* (2025).

[31] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2023. MetaTPTrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5239–5247.

[32] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[33] Lindsay I Smith. 2002. A tutorial on principal components analysis. (2002).

[34] Chia-Yi Su and Collin McMillan. 2023. Semantic Similarity Loss for Neural Source Code Summarization. *arXiv preprint arXiv:2308.07429* (2023).

[35] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *arXiv preprint arXiv:2305.12865* (2023).

[36] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3282–3299.

[37] Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé Bissyandé. 2024. CodeAgent: Autonomous Communicative Agents for Code Review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 11279–11313.

[38] Xunzhu Tang, Haoye Tian, Zhenghan Chen, Weiguo Pian, Saad Ezzini, Abdoul Kader Kaboré, Andrew Habib, Jacques Klein, and Tegawende F Bissyande. 2024. Learning to represent patches. In *Proceedings of the 2024 IEEE/ACM 46th*

International Conference on Software Engineering: Companion Proceedings. 396–397.

[39] Xunzhu Tang, Haoye Tian, Pingfan Kong, Saad Ezzini, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2024. App review driven collaborative bug finding. *Empirical Software Engineering* 29, 5 (2024), 124.

[40] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 981–992.

[41] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[42] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this Change the Answer to that Problem? Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. *arXiv preprint arXiv:2208.04125* (2022).

[43] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 386–396.

[44] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2409–2426.

[45] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. Detecting" 0-day" vulnerability: An empirical study of secret security patch in OSS. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 485–492.

[46] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 149–160.

[47] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 595–600.

[48] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–9.

[49] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[50] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[51] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[52] Xi Wei, Tianzhu Zhang, Yan Li, Yongdong Zhang, and Feng Wu. 2020. Multi-modality cross attention network for image and sentence matching. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10941–10950.

[53] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *The 2020 Annual Network and Distributed System Security Symposium (NDSS'20)*.

[54] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. 2020. Automatic hot patch generation for android kernels. In *29th USENIX Security Symposium (USENIX Security 20)*. 2397–2414.

[55] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawende Bissyande, Claire Le Goues, and Shunfu Jin. 2025. MORepair: Teaching LLMs to Repair Code via Multi-Objective Fine-Tuning. *ACM Transactions on Software Engineering and Methodology* (2025).

[56] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E Hassan. 2023. Colefunda: Explainable silent vulnerability fix identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2565–2577.

[57] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.

[58] Xin Zhou, Bowen Xu, DongGyun Han, Zhou Yang, Junda He, and David Lo. 2023. CCBERT: Self-Supervised Code Change Representation Learning. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 182–193.

[59] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

[60] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.