IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 15 August 2024, accepted 1 October 2024, date of publication 7 October 2024, date of current version 22 October 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3474857

RESEARCH ARTICLE

Similarity-Based Source Code Vulnerability Detection Leveraging Transformer Architecture: Harnessing Cross-Attention for Hierarchical Analysis

SUNGMIN HAN[1], MIJU KIM[1], JAESIK KANG[2], KWANGSOO KIM[2], SEUNGWOON LEE[2], AND SANGKYUN LEE[1], (Member, IEEE)

[1]School of Cybersecurity, Korea University, Seoul 02841, Republic of Korea
[2]Cyber Warfare Research and Development Laboratory, LIG Nex1, Seongnam-si 13488, Republic of Korea

Corresponding author: Sangkyun Lee (sangkyun@korea.ac.kr)

This work was supported by Korea University Grant and LIG Nex1.

ABSTRACT The growing complexity and volume of modern software have led to an increase in source code vulnerabilities, posing significant security risks. In response, deep learning-based automated source code vulnerability detection methods, particularly those utilizing source code similarity analysis, have recently emerged as promising solutions. However, existing similarity-based source code vulnerability detection methods frequently fail to fully utilize information from the hierarchical structure of source code and are often computationally expensive, limiting their practicality in real-world scenarios. In this paper, we introduce XTransformer, a novel deep learning-based source code vulnerability detector tailored for comparing target source code against archived vulnerable codes across various levels of the source code's hierarchical structure by leveraging extra cross-attention imposed on the transformer architecture. Additionally, we propose a specialized training strategy based on supervised contrastive learning to improve XTransformer's ability to effectively learn and differentiate between vulnerable and non-vulnerable source codes. Comprehensive experiments demonstrate that XTransformer outperforms current state-of-the-art methods across different datasets and code lengths while significantly reducing the inference time compared to other similarity-based methods that utilize hierarchical information from source code.

INDEX TERMS Code similarity, contrastive learning, cross-attention, source code vulnerability detection, transformer.

I. INTRODUCTION

The rapid evolution of modern software development has dramatically increased the complexity and volume of source code. As the complexity and volume of source code grow, vulnerabilities, which are flaws or weaknesses in the underlying code that malicious attackers can exploit [1], [2], have also become increasingly complex and challenging for human experts to detect manually. Consequently, the threat posed by these vulnerabilities has escalated, leading to a higher risk of severe security breaches.

The 2021 data breach in Microsoft Exchange Servers, one of the largest breaches in Microsoft's history, illustrates the dangers of source code vulnerabilities. This breach, which affected nearly 400,000 Exchange servers, was the result of vulnerabilities within the software's source code [3]. Thus, the need to develop automated source code vulnerability detection has become urgent in order to address modern software's rising complexity and volume of vulnerabilities.

Recently, with the advancement of deep learning and the successful introduction of deep learning models in cybersecurity, research in automated source code vulnerability detection has focused on applying and developing deep learning-based detection models [4], [5], [6], [7], [8], [9],

The associate editor coordinating the review of this manuscript and approving it for publication was Jemal H. Abawajy.

[10], [11], [12], [13], [14]. Among these methods, similarity-based approaches [10], [11], [12], [13], which detect vulnerable code by comparing it against known vulnerable codes, have gained attention for their effectiveness in detecting not only known vulnerabilities that have been previously identified or listed in the Common Vulnerabilities and Exposures (CVE) database [15], but also new vulnerabilities, known as zero-day vulnerabilities, that have not yet been discovered.

However, despite the effectiveness of similarity-based approaches, we find that few methods fully utilize the information embedded in the hierarchical structure of source code, even though previous studies have recognized the importance of using information from different levels of this hierarchical structure for detecting vulnerabilities—from token-level to higher levels such as function-level [9], [13]. Additionally, even when approaches attempt to use hierarchical structure information, they often require a high computational cost to detect source-code vulnerabilities. For example, very recent work [13] incorporates hierarchical information to detect vulnerable code, but its inference time required for detection is significantly high, making it unsuitable for real-world systems that require timely detection.

In this paper, we introduce XTransformer, a novel deep learning-based source code vulnerability detector based on source code similarity analysis. Unlike existing approaches that rely on the Siamese network architecture [16], XTransformer utilizes a customized transformer architecture for a more time-efficient hierarchical comparison of source codes. By modifying the original transformer design [17] to leverage multiple cross-attention layers for comparing archived vulnerable codes (known vulnerable codes) and target source codes across various levels of their hierarchical structure, XTransformer achieves improved detection accuracy and efficiency. Additionally, we propose a specialized training strategy based on supervised contrastive learning [18], which enhances XTransformer's detection performance by enabling it to extract core features from vulnerable code, even when training data is limited, helping to distinguish between vulnerable and non-vulnerable source codes.

Our contributions can be summarized as follows: (1) We propose XTransformer, a novel similarity-based source code vulnerability detector built on a customized transformer architecture that is better suited for hierarchical similarity analysis of source code. XTransformer efficiently utilizes information from the hierarchical structure of source code by leveraging cross-attention to compare source code similarity, thereby improving detection accuracy. (2) We introduce a specialized training strategy based on supervised contrastive learning, tailored to enhance XTransformer's performance in the source code vulnerability detection scenario. (3) We demonstrate through experiments that XTransformer surpasses current state-of-the-art methods, delivering superior performance and timely detection across various datasets.

II. RELATED WORKS

We provide an overview of deep learning-based source code vulnerability detection and contrastive learning, the two key related subjects to our work.

A. SOURCE CODE VULNERABILITY DETECTION

The goal of source code vulnerability detection is to identify potential security weaknesses or flaws in software's source codes that attackers could exploit for malicious activities, such as unauthorized access and data breaches [1], [2].

There are two main ways to analyze vulnerabilities in source code: dynamic analysis and static analysis. Dynamic analysis involves analyzing a source code during its execution, while static analysis examines a source code without executing it. Since static analysis can detect vulnerabilities early in the development stage, incurs less computational cost, and is faster than dynamic analysis, it is often preferred [1], [2], [9], [13]. Our work considers the static analysis scenario.

Traditionally, rule-based methods, which identify vulnerable source code by matching predefined patterns or rules, have been used to detect vulnerabilities within the static analysis scenario. However, rule-based approaches often struggle to detect vulnerabilities that deviate from predefined patterns and rely heavily on manually crafted rules by domain experts. This manual process is costly, and the effectiveness of detection can vary depending on the security expertise of the individual who defines the rules [5], [19].

Therefore, with the success of deep learning models and their ability to automatically learn and extract patterns from data, deep learning-based methods for source code vulnerability detection have been proposed to address the challenges of rule-based methods [4], [5], [6], [7], [8], [9], [11], [12], [13]. For example, Russell et al. [5] proposed a convolutional neural network (CNN)-based vulnerability detector, which was the first method to apply deep learning directly to source code for feature learning. Their work demonstrated that deep feature representation learning on source code is a promising approach for automated vulnerability detection. Gu et al. [9] introduced a hierarchical model inspired by the structural similarity between code and documents, where source code tokens form statements and statements form code blocks—similar to how text tokens form sentences and sentences form documents. They implemented a hierarchical attention network, consisting of two levels of bidirectional gated recurrent unit (BiGRU) layers with attention mechanisms [20], to extract vulnerability patterns from the source code's token- and statement-level attributes, thereby enhancing the accuracy of vulnerability detection.

Among these deep learning-based approaches, similarity-based methods [11], [12], [13], which detect vulnerable code by comparing it against known vulnerable codes often collected from historical data, have garnered attention for their effectiveness in identifying both new and previously encountered vulnerabilities. This approach is particularly effective at detecting vulnerabilities in duplicated code,

which is crucial given that studies [21], [22], [23], [24], [25] have shown duplicated code can constitute up to 20% of modern software. This significantly increases the risk of recurring vulnerabilities if a defect is found in any part of the duplicated code [13].

Since similarity-based source code vulnerability detection is analogous to text similarity analysis, which quantifies the similarity between two pieces of text [26], [27], many vulnerability detection methods have adopted text similarity techniques. In particular, as the Siamese neural network [16] has emerged as a powerful architecture for text similarity tasks [28], [29], [30], [31], [32], [33], Siamese architecture-based methods for source code vulnerability detection have been proposed [11], [12], [13].

The Siamese architecture consists of two identical sub-networks with the same configuration, parameters, and weights, each independently processing one of the two input data. The outputs are then compared to calculate a similarity score, indicating how closely related the two inputs are. Siamese architecture-based approaches for source code vulnerability detection differ in their choice of sub-networks.

For example, Sun et al. [12] introduced VDSimilar, which employs bidirectional long short-term memory (BiLSTM) networks coupled with attention mechanisms [20] as the sub-network within the Siamese architecture for detecting vulnerabilities. Their research demonstrated that VDSimilar is highly effective at identifying vulnerable code, even with a relatively small dataset, and significantly outperforms non-similarity-based vulnerability detectors. More recently, Han et al. [28] proposed CODE-SMASH, a state-of-the-art model based on a Siamese architecture, designed to detect vulnerabilities by analyzing the similarity of hierarchical information extracted from two pieces of source code. This approach, which compares token- to function-level attributes across different code samples, has demonstrated strong performance in detecting vulnerabilities across a wide range of code lengths, underscoring the importance of considering information across various hierarchical levels of source codes.

Despite advancements in deep learning-based similarity approaches for source code vulnerability detection, these methods often fail to fully utilize information from the various levels of the source code's hierarchical structure. Even when they do, the computational time required for detecting vulnerabilities tends to be high.

B. CONTRASTIVE LEARNING

Contrastive learning is a powerful learning framework that trains models to transform data points into vector representations such that similar (or positive) pairs sharing semantics are pulled closer in a representation space while dissimilar (or negative) pairs are pushed apart. It is foundational in enabling models to discern and encode the semantic relationships between data points effectively [34]. Two main approaches within contrastive learning are distinguished by their use of label information: self-supervised contrastive and supervised contrastive learning.

Self-supervised contrastive learning empowers models to derive semantically meaningful data representations without relying on explicit label information. Instead of using predetermined labels, this approach forms positive pairs from augmented variations of the same data point and negative pairs from distinct data points [35], [36], [37]. A notable contribution to this strategy is SimCLR, introduced in [35], which presents a simplified self-supervised contrastive learning framework that eliminates the need for specialized architectures or memory banks. Their findings emphasize the critical role of multiple data augmentations in deriving effective representations, with contrastive learning benefiting more from data augmentation techniques than conventional supervised learning does.

Building upon the self-supervised contrastive learning, supervised contrastive learning incorporates label information to further refine the contrastive learning process, enhancing the model's ability to distinguish between different classes more effectively [18]. It leverages the available labels to form positive pairs not just between augmented versions of the same instance but also among different instances of the same class. It has shown to be particularly beneficial in scenarios where fine-grained distinctions between classes are crucial.

Contrastive learning has also been studied for natural language processing (NLP), as demonstrated in works like [38], [39], [40], [41], [42], [43]. Initial efforts in NLP tried to apply contrastive learning using data augmentation techniques such as word deletion, reordering, and substitution [38], [39]. However, adapting contrastive learning to NLP presents unique challenges due to the complex semantics of the natural language, where even minor changes like altering a single word can significantly shift the meaning. A notable advancement is SimCSE [40], which utilizes the inherent dropout mechanism [44] of transformer-based models as a form of natural data augmentation, effectively generating variant sentence embeddings without altering the original text input. Using dropout [44] as a data augmentation for contrastive learning, they showed that SimCSE can significantly enhance state-of-the-art sentence embeddings on semantic textual similarity tasks.

Inspired by the effectiveness of contrastive learning in NLP and considering that source code can be treated as a form of text, we employ supervised contrastive learning to train our model. This approach enhances the model's ability to extract core representations from source codes, even when the available training data is limited.

III. METHODOLOGY

We introduce XTransformer, a novel deep learning-based source code vulnerability detector based on similarity analysis. We begin by outlining our problem setup. Following this, we detail the architecture of our model, which is built upon the transformer [17], and we describe our training

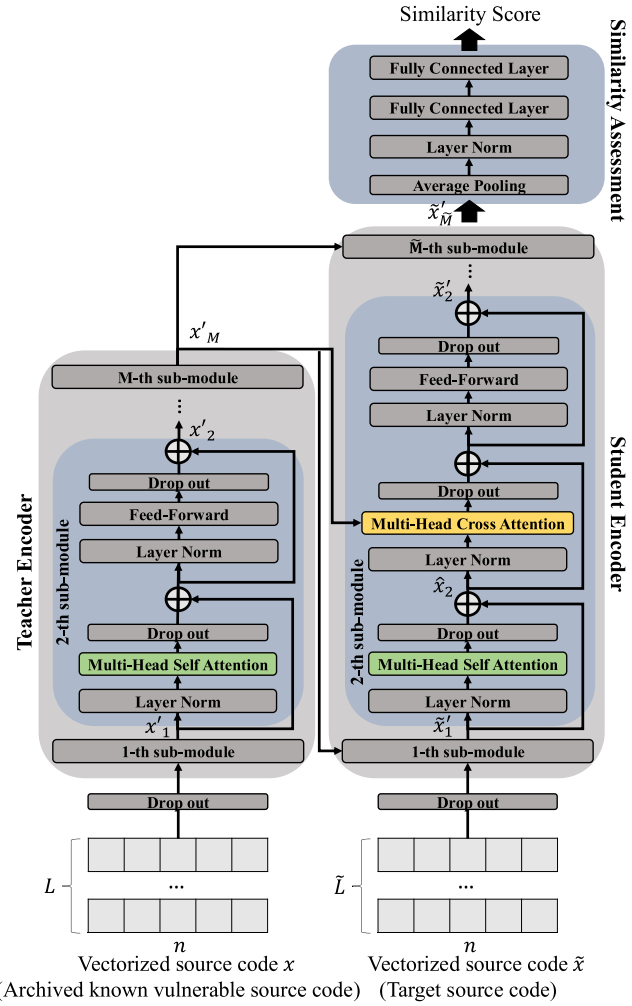

FIGURE 1. The overall architecture of our XTransformer.

strategy, which employs supervised contrastive learning [18]. Figure 1 depicts the architecture of our model, and our training strategy is illustrated in Fig. 2.

A. PROBLEM STATEMENT

We consider a detection scenario involving several archived vulnerable source codes, which is a collection of previously identified and analyzed codes containing specific security flaws. For each new source code that needs vulnerability assessment, we evaluate its similarity to the archived vulnerable codes. If a new source code closely matches an archived vulnerable code, it is classified as vulnerable.

Both archived and new codes to be evaluated are treated as sequences of tokens, which are the smallest units used to represent source code. Since deep learning models operate on numerical data, each token is transformed into a vectorized form using a method such as word2vec [45].

Formally, we denote an archived known vulnerable code as $x \in \mathbb{R}^{L\times n}$ and a new source code (target source code) as $\tilde{x} \in \mathbb{R}^{\tilde{L}\times n}$, where L and $\tilde{L}$ represent the length of each code's token sequence, respectively. Here, n is the dimension of a vectorized token. Our goal is to develop a model capable of detecting vulnerable codes by efficiently computing the similarity between a new source code $\tilde{x}$ and an archived code x, leveraging hierarchical information across various levels of their structure.

B. MODEL ARCHITECTURE

Contrary to previous studies [12], [13] that rely on the Siamese architecture [16] to evaluate the similarity between two source codes, we propose a model that utilizes the transformer architecture's cross-attention mechanism [17] to assess the similarity of a target source code by comparing them with archived vulnerable code across the hierarchical structure of the target source code. Our model comprises three main components: teacher encoder, student encoder, and similarity assessment component. We chose the names of teacher and student encoders since the former takes archived vulnerable codes, which are like correct answers provided by teachers, while the latter takes the codes to be tested for vulnerability, like answers provided by students.

1) TEACHER ENCODER

The teacher encoder is engineered to extract the core representation of a given archived vulnerable code x, capturing key attributes essential for analysis. It comprises M stacked sub-modules, all sharing the same architectural design.

Each sub-module contains two main sub-layers: a multi-head self-attention layer followed by a feed-forward layer, both of which are enhanced with layer normalization [46], dropout [44], and residual connections [47]. Unlike the original transformer architecture [17] that adopts layer normalization after each sub-layer, our model introduces layer normalization before each sub-layer. This modification is designed to enhance the stability of model training, ultimately leading to improvements in the model's performance, as studied in [48].

The multi-head self-attention layer within each sub-module of the teacher encoder consists of multiple self-attention heads. Each head computes its output by utilizing queries, keys, and values derived from the same data as follows:

$$A'_{m,h} := \text{softmax}\left(\frac{Q_{m,h}(x'_{m-1})K_{m,h}(x'_{m-1})^T}{\sqrt{s}}\right)V_{m,h}(x'_{m-1}),$$

where $A'_{m,h} \in \mathbb{R}^{L\times d}$ represents the output of the h-th head within the m-th sub-module, d is the dimension of the output for each self-attention head, and $s \in \mathbb{R}$ is a scaling factor. $x'_{m-1} \in \mathbb{R}^{L\times n}$ corresponds to the output from the $(m-1)$-th sub-module for the archived vulnerable code x, with m ranging from 1 to M. The transformation functions $Q_{m,h}(\cdot)$, $K_{m,h}(\cdot)$, and $V_{m,h}(\cdot)$ produce h-th head's query, key, and value, respectively, through dot products between x'_{m-1} and the weight matrices $W^Q_{m,h}$, $W^K_{m,h}$, and $W^V_{m,h}$, each in $\mathbb{R}^{n\times d}$. This structured approach enables the model to dynamically focus on different features of the previous layer's output x'_{m-1}. The output from each attention head, denoted as $A'_{m,h}$

for h ranging from 1 to H, is combined through concatenation, and then x'_{m-1} is added via a residual connection to form $A_m = \text{Concat}(A'_{m,1}, A'_{m,2}, \ldots, A'_{m,H}) + x'_{m-1}$, where $A_m \in \mathbb{R}^{L\times n}$.

The combined output A_m is further processed through the feed-forward network, which consists of two fully connected layers with the rectified linear unit (ReLU) activation function, yielding the final output $x'_m \in \mathbb{R}^{L\times n}$ for the m-th sub-module as follows:

$$x'_m = \text{ReLU}(A_m \cdot W_m^o + b_m^o) \cdot W_m^{o'} + b_m^{o'} + A_m,$$

where ReLU$(\cdot)$ represents the ReLU activation function applied element-wise. Here, W_m^o and $W_m^{o'}$ are the weight matrices of the fully connected layers, and b_m^o and $b_m^{o'}$ are the bias terms for adjusting the output. The addition of A_m signifies a residual connection. Note that we have omitted layer normalization and dropout in the above expressions for simplicity.

Through the sequential application of these sub-modules, the teacher encoder conducts a hierarchical analysis of input token sequences to capture various levels of semantic meaning, as studied in [49], [50], and [51]. This approach enables the teacher encoder to adeptly extract the core representation of a given archived vulnerable code, capturing both its token-level details and higher-level deep semantic meanings.

2) STUDENT ENCODER

The student encoder is designed to compare a target source code $\tilde{x}$ with the core representation of a given archived vulnerable source code x extracted by the teacher encoder. It aims to facilitate a deep, intrinsic comparison, focusing on identifying and extracting features in a code $\tilde{x}$ that closely align with the fundamental characteristics of the archived code. This approach ensures that the analysis not only recognizes direct matches but also appreciates nuanced similarities essential for accurate similarity assessments.

The student encoder's architecture is similar to that of the teacher encoder, consisting of stacked sub-modules that contain a multi-head self-attention layer and a feed-forward layer. A key distinction of the student encoder is the incorporation of an additional multi-head cross-attention layer within its sub-modules. Each sub-module in the comparison component is composed of three main layers: a multi-head self-attention layer to capture features of semantic meanings within a source code internally, a multi-head cross-attention layer to compare and align a code with the archived vulnerable code's core representation, and a feed-forward layer to refine the processed information further.

More specifically, consider the output from the multi-head self-attention layer within the $\tilde{m}$-th sub-module of the student encoder for a target code $\tilde{x} \in \mathbb{R}^{\tilde{L}\times n}$ as $\hat{x}_{\tilde{m}} \in \mathbb{R}^{\tilde{L}\times n}$, where $\tilde{m}$ spans from 1 to $\tilde{M}$, and let the output of the teacher encoder for the archived code x be denoted as $x'_M \in \mathbb{R}^{L\times n}$. In contrast to the multi-head self-attention that computes queries, keys, and values from identical data, the multi-head cross-attention layer utilizes distinct data for these elements. This layer utilizes multiple cross-attention heads where each head computes queries from the current state of the code $\hat{x}_{\tilde{m}}$ and keys and values from the core representation x'_M of the archived code x as follows:

$$\tilde{A}'_{\tilde{m},h} := \text{softmax}\left(\frac{Q_{\tilde{m},h}(\hat{x}_{\tilde{m}})K_{\tilde{m},h}(x'_M)^T}{\sqrt{\tilde{s}}}\right)V_{\tilde{m},h}(x'_M),$$

where $\tilde{A}'_{\tilde{m},h} \in \mathbb{R}^{\tilde{L}\times\tilde{d}}$ signifies the output of the h-th head within the $\tilde{m}$-th sub-module, $\tilde{d}$ represents the dimension of the output for each cross-attention head, and $\tilde{s} \in \mathbb{R}$ acting as a scaling factor. $Q_{\tilde{m},h}(\cdot)$, $K_{\tilde{m},h}(\cdot)$, and $V_{\tilde{m},h}(\cdot)$ are the transformations for producing the query, key, and value. This cross-attention layer allows the model to directly compare and contrast the current processed state $\hat{x}_{\tilde{m}}$ of the code $\tilde{x}$ by the preceding multi-head self-attention layer with the core representation x'_M of the archived vulnerable code x, adjusting its analytical focus to align and emphasize aspects of the code that are most similar to the given archived code.

Outputs from all the cross-attention heads are concatenated, and then $\hat{x}_{\tilde{m}}$, which is the output of the preceding multi-head self-attention layer, is added to the concatenated output via a residual connection: $\tilde{A}_{\tilde{m}} = \text{Concat}(\tilde{A}'_{\tilde{m},1}, \tilde{A}'_{\tilde{m},2}, \ldots, \tilde{A}'_{\tilde{m},H}) + \hat{x}_{\tilde{m}}$, resulting in $\tilde{A}_{\tilde{m}} \in \mathbb{R}^{\tilde{L}\times n}$. This composite output is further processed by the feed-forward layer:

$$\tilde{x}'_{\tilde{m}} = \text{ReLU}(\tilde{A}_{\tilde{m}} \cdot \tilde{W}_{\tilde{m}}^o + \tilde{b}_{\tilde{m}}^o) \cdot \tilde{W}_{\tilde{m}}^{o'} + \tilde{b}_{\tilde{m}}^{o'} + \tilde{A}_{\tilde{m}},$$

where $\tilde{x}'_{\tilde{m}} \in \mathbb{R}^{\tilde{L}\times n}$ represents the output for the $\tilde{m}$-th sub-module, and $\tilde{m}$ spans from 1 to $\tilde{M}$

By stacking these sub-modules, the student encoder compares various semantic levels of a code $\tilde{x}$ with the core representation x'_M of a given archived vulnerable code x, focusing on extracting features from the code that closely align with the core representation of the archived code. This approach allows the model to compare two source codes from token-level information to higher-level information, such as function-level information, providing a hierarchical processing capability. This makes the model more effective in handling not only short code sequences but also long code sequences, making it more suitable for addressing the complexity and dynamism of modern software vulnerabilities, similar to the previous study [13].

3) SIMILARITY ASSESSMENT

The similarity assessment component calculates the similarity score between a target code $\tilde{x}$ and an archived vulnerable code x by analyzing the output $\tilde{x}'_{\tilde{M}} \in \mathbb{R}^{\tilde{L}\times n}$ from the student encoder, which reflects features of the code $\tilde{x}$ that are closely aligned with the core representation x'_M of the given archived code x. This process involves averaging the features of the student encoder's output $\tilde{x}'_{\tilde{M}}$ to reduce its dimensionality to $\mathbb{R}^{\tilde{L}}$, resulting in a representation that is more manageable

for assessing similarity. The averaged data is then normalized and passed through two fully connected layers with ReLU activation function. Subsequently, the output from these layers is processed through the sigmoid function to generate a similarity score within the range of [0, 1], indicating the degree of similarity.

Finally, with a predefined threshold $\rho > 0$, we assess the presence of source code vulnerabilities. If the similarity score is greater than ρ, the target code $\tilde{x}$ is considered vulnerable code; otherwise, it is deemed benign.

C. TRAINING STRATEGY

We propose a specialized training strategy designed to enhance our model's ability to identify core features in code, thereby improving its detection performance, even when training data is limited. Initially, we focus on training the teacher encoder exclusively using supervised contrastive learning [18]. Subsequently, we train the student encoder along with the similarity assessment component.

To detail our approach, we define each minibatch sampled from a set of various source codes, which will be compared with archived known vulnerable codes, as $D := \{(\tilde{x}^{(i)}, \tilde{y}^{(i)})\}_{i=1}^{N}$, where $\tilde{x}^{(i)}$ is the i-th source code in the set D and $\tilde{y}^{(i)}$ is its corresponding label (vulnerable or not). We set the label $\tilde{y}^{(i)} = 1$ for vulnerable code and $\tilde{y}^{(i)} = 0$ for non-vulnerable code. Here, N is the total number of source codes in the set D. Additionally, each minibatch includes a set of archived vulnerable codes with P samples, defined as $G := \{(x^{(p)}, y^{(p)})\}_{p=1}^{P}$, where $x^{(p)}$ denotes the p-th archived known vulnerable code, and $y^{(p)}$ denotes its label. Since the codes in the set G consist solely of vulnerable codes, we set $y^{(p)} = 1$ for all p, where $p = 1 \ldots P$. Here, P represents the total number of samples in the set G.

1) TRAINING TEACHER ENCODER

The first step of our training strategy focuses on training the teacher encoder. This process enables the teacher encoder to identify and extract distinct representations of archived known vulnerable source codes, effectively distinguishing them from non-vulnerable codes. To achieve this, we adopt a supervised contrastive learning framework [18].

Unlike conventional contrastive learning methods [18], [35], [36], [40], where anchor points—used as reference points to compare with other data points for learning discriminative features—can be any data points, our strategy exclusively utilizes archived vulnerable source codes from the set G as anchor points. This approach enables our model to specifically learn discriminative features of vulnerable codes based on these archived vulnerable source codes. In addition to setting these archived vulnerable source codes as anchor points, we designate source codes from each minibatch as either positive or negative samples relative to these anchor points: a source code is considered a positive sample if it is vulnerable, and a negative sample if it is non-vulnerable.

Let us formalize our approach by defining the teacher encoder as a function $f_e(\cdot; w_e) : \mathbb{R}^{L\times n} \rightarrow \mathbb{R}^{L\times n}$, which transforms a source code into an encoded representation. Following previous contrastive learning approaches [18], [35], [36], we introduce a linear projection head denoted by $f_g(\cdot; w_g) : \mathbb{R}^{L\times n} \rightarrow \mathbb{R}^{L'}$, which positioned atop of a feature extractor to map the extracted features into an embedding space conducive for applying contrastive loss, thereby facilitating a more effective representation learning. By integrating the linear projection head f_g on top of the teacher encoder f_e, we establish a composite representation function $f_r(\cdot; w_e, w_g) := f_g \circ f_e(\cdot)$, where w_e and w_g are the weights of the teacher encoder and the linear projection head, respectively.

With the representation function $f_r(\cdot; w_e, w_g)$ and for every minibatch set $B := D \cup G$ including N various source codes and P archived known vulnerable source codes, we define our supervised contrastive loss for training the teacher encoder as follows:

$$\mathcal{L}_{\text{con}}(w_e, w_g) = \frac{-1}{P} \sum_{p=1}^{P} \sum_{i=1}^{N+P} \bar{y}^{(i)} \left(\log \frac{\exp((z^{(p)})^T \bar{z}^{(i)}/\tau)}{\sum_{j=1}^{N+P} \exp((z^{(p)})^T \bar{z}^{(j)}/\tau)} \right),$$

where $z^{(p)} = f_r(x^{(p)})$ indicates the output of the function f_r for the p-th archived known vulnerable source code $x^{(p)}$ (anchor point), where $p = 1 \ldots P$ from the set G. Here, $\bar{z}^{(i)}$ represents the output of the function f_r for the i-th source code from the minibatch set B and $\bar{y}^{(i)}$ is its corresponding label: 1 for vulnerable or 0 for non-vulnerable code, where $i = 1 \ldots N + P$. τ is a temperature hyperparameter.

The expression $N + P$ refers to the total number of source code samples in each minibatch, combining N samples from the set D and P vulnerable source code samples from the set of archived vulnerable codes G. The inclusion of these P vulnerable source code samples, which are originally sampled to act as anchor points, serves as a form of data augmentation. This is achieved by leveraging the dropout mechanism within our model to introduce slight variations in their representations, as discussed in [40]. Since these P samples are all known vulnerable codes, we assign them the label of vulnerable (1) and consider them as positive samples to the anchor point $x^{(p)}$.

Upon completing this training procedure, we discard the linear projection head $f_g(\cdot; w_g)$ and use only the teacher encoder $f_e(\cdot; w_e)$.

2) TRAINING STUDENT ENCODER WITH SIMILARITY ASSESSMENT

In our second training step, we train the student encoder along with the similarity assessment component. During this training procedure, the pre-trained weights of the teacher encoder, trained in the first step, are kept fixed to ensure it provides a stable and consistent representation of archived vulnerable source codes. Additionally, we initialize the weights of the multi-head self-attention layers and

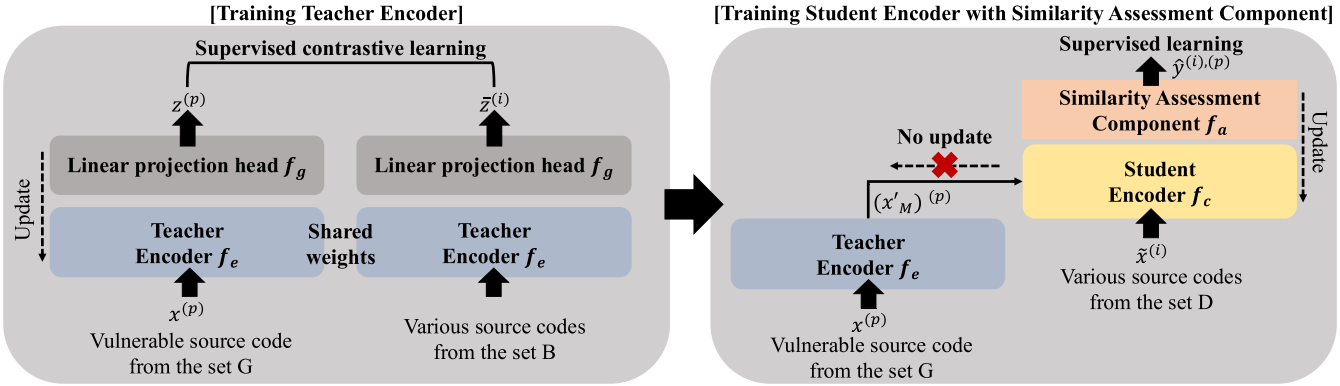

FIGURE 2. An overview of our training strategy.

feed-forward layers within the student encoder with the corresponding pre-trained weights from the teacher encoder.

To formally describe our method, we define the student encoder as a function $f_c(\cdot; w_c) : \mathbb{R}^{\tilde{L}\times n} \rightarrow \mathbb{R}^{\tilde{L}\times n}$, which processes source codes into representations that are aligned with the output from the teacher encoder. We also define the similarity assessment component as a function $f_a(\cdot; w_a) : \mathbb{R}^{\tilde{L}\times n} \rightarrow \mathbb{R}$. These functions are combined into a unified function $f_u(\cdot; w_c, w_a) := f_a \circ f_c(\cdot)$, where w_c and w_a represent the weights of the student encoder and the similarity assessment component, respectively.

With the unified function $f_u(\cdot; w_c, w_a)$, our training loss is defined as follows:

$$\mathcal{L}(w_c, w_a) = \frac{-1}{PN}\sum_{p=1}^{P}\sum_{i=1}^{N}\tilde{y}^{(i)}\log(\hat{y}^{(i),(p)}) + (1-\tilde{y}^{(i)})\log(1-\hat{y}^{(i),(p)}),$$

where $\hat{y}^{(i),(p)} = f_u(\tilde{x}^{(i)}|(x'_M)^{(p)}; w_c, w_a)$ represents the predicted similarity between the i-th source code $\tilde{x}^{(i)}$ in the set D and the p-th archived vulnerable code $x^{(p)}$ in the set G. Here, $(x'_M)^{(p)}$ refers to the output from the teacher encoder for the p-th archived vulnerable code $x^{(p)}$. This approach enables precise model training focused on the similarity assessment between source codes.

IV. EXPERIMENTS

In this section, we demonstrate the effectiveness of our proposed method, which we call XTransformer, by comparing its performance against various similarity-based source code vulnerability detection methods.

A. EXPERIMENT SETTINGS

All our experiments were conducted in a computational environment with PyTorch v.1.9.1, Numpy v.1.17.4, and scikit-learn v.0.22.2, running on an Ubuntu 18.04.3 (64-bit) system. The hardware setup included an Intel Xeon Silver 4214 CPU, 32GB RAM, and an NVIDIA GeForce RTX2080Ti GPU, supported by CUDA v.10.2.

TABLE 1. Characteristics of the Chromium, Debian, and Sun datasets used for experiments.

Dataset	Chromium	Debian	Sun
No. of vulnerable functions	701	1,056	324
No. of non-vulnerable functions	3,391	15,785	465
No. of similar pairs	245,350	557,040	2,158
No. of dissimilar pairs	2,377,091	16,668,960	2,398

1) BASELINE METHODS

We compared our XTransformer with four deep learning-based source code vulnerability detectors that are based on source code similarity: TokenCNN [5], HAN [9], VDSimilar [12], and CODE-SMASH [13]. Following the approach of the previous study [13], we adapted TokenCNN [5] and HAN [9], originally proposed for vulnerability classification, into a Siamese structure with two fully connected layers to function as similarity-based detection models.

It is noteworthy that, as studied in [13], TokenCNN and VDSimilar only consider token-level features of source code, while HAN considers both token and statement-level features, and CODE-SMASH considers token, statement, and function-level features for source code similarity analysis in detecting vulnerabilities.

2) DATASET

We tested our method on three different datasets: the Chromium and Debian datasets [52] for large-scale datasets, and the Sun dataset [12] (the name is after the first author of the paper which has introduced the dataset) for a small-scale dataset. Each dataset consists of vulnerable and non-vulnerable functions, where each function represents a sequence of code that performs specific tasks.

Specifically, the Chromium and Debian datasets, sourced from the Reveal dataset [52] and built upon real-world projects, consist of functions written in C/C++ language and annotated as either vulnerable or non-vulnerable. In our experiments, we focused on functions with fewer than 80 lines of code, categorizing them by line count to evaluate

our method's effectiveness across different code lengths, as explored in a previous study [13].

The Sun dataset also consists of vulnerable and non-vulnerable functions written in C/C++, but it is organized according to various CVEs (Common Vulnerabilities and Exposures), unlike the Chromium and Debian datasets. In our experiments, similar to the approach of the original paper [12], we considered CVEs that appear at least three vulnerable and three non-vulnerable functions so that the number of generated similar and different pairs will be sufficient for training.

We then created pairs of functions, where each pair consisted of a vulnerable function and either another vulnerable function or a non-vulnerable function. For the Chromium and Debian datasets, we built function pair datasets using all the vulnerable and non-vulnerable functions. In the case of the Sun dataset, we paired functions according to CVEs, as done in the original study [12]. We labeled these pairs as similar (1) if both functions are vulnerable and dissimilar (0) if they are not. We outline the details in Table 1.

For the Chromium and Debian datasets, we split the function pair datasets into training, validation, and test sets with a 6:2:2 ratio, following previous works [9], [13]. For the Sun dataset, due to the small amount of data, we conducted 10-fold cross-validation following the original work [12]. Specifically, we randomly divided the dataset into 10 subsets. For each fold, 9 subsets were used for training (with the training data further divided in an 8:2 ratio for training and validation) and 1 subset for testing. This process was repeated 10 times, resulting in 10 groups of results. We then reported the average of these results.

3) EVALUATION METRIC

We evaluated the performance of similarity models with four metrics: accuracy, precision, recall, and F1 score, which are elaborated as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}},$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}},$$

$$\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Here, TP (true positive) is the number of similar function pairs correctly identified as similar by a model, TN (true negative) is the number of dissimilar function pairs correctly identified as dissimilar, FP (false positive) refers to the number of dissimilar function pairs wrongly identified as similar, and FN (false negative) refers to the number of similar function pairs wrongly identified as dissimilar.

The higher the values of these metrics, the better the model's performance in identifying and differentiating between similar and dissimilar function pairs. The F1 score, in particular, is preferable as it provides a balanced measure of the model's precision and recall. This is especially valuable in situations where there is an imbalance between the classes, such as in our case, where the number of similar pairs and dissimilar pairs are imbalanced.

TABLE 2. Detection performance on the Chromium, Debian, and Sun datasets. The best values are boldfaced and the second-best are underlined.

	Model	Accuracy	Precision	Recall	F1
Chromium	TokenCNN	0.9170	0.9140	0.9480	0.9218
	VDSimilar	<u>0.9344</u>	<u>0.9229</u>	0.9446	0.9318
	HAN	0.9145	0.9160	0.9416	0.9229
	CODE-SMASH	0.9233	0.9192	**0.9711**	<u>0.9355</u>
	XTransformer	**0.9549**	**0.9464**	<u>0.9691</u>	**0.9545**
Debian	TokenCNN	0.9576	0.9728	<u>0.9194</u>	0.9452
	VDSimilar	0.7860	0.6700	0.9127	0.7726
	HAN	0.9575	0.9733	0.9188	0.9450
	CODE-SMASH	<u>0.9596</u>	**0.9940**	0.9043	<u>0.9469</u>
	XTransformer	**0.9627**	<u>0.9864</u>	**0.9195**	**0.9515**
Sun	TokenCNN	0.7385	0.5584	0.6475	0.5698
	VDSimilar	0.7062	0.4876	0.5453	0.5083
	HAN	0.5827	0.3366	0.5029	0.3529
	CODE-SMASH	<u>0.9285</u>	<u>0.8797</u>	<u>0.8633</u>	<u>0.8705</u>
	XTransformer	**0.9470**	**0.9098**	**0.9004**	**0.9040**

4) IMPLEMENTATION DETAILS

We followed the previous study [13] for tokenizing source codes and employed word2vec [45] for source code vectorization with an embedding size of 64. We trained all models with a batch size of 50 for 120 epochs at a learning rate of 10^{-3}, using the Adam optimizer [53].

For our model, we configured both the teacher encoder and the student encoder with four sub-modules, applied a dropout rate of 0.2, and set the number of attention heads to four. Based on the best results from a previous study [40], we set the temperature hyperparameter τ to 0.05 in our experiments. For the baseline models, we adhered to the optimal settings from their respective studies.

We adjusted each model's detection threshold ρ, which converts the model's output score (similarity score) into binary decisions to determine whether the source codes are similar or dissimilar, to the value that yielded the highest F1 score on the validation set.

B. DETECTION PERFORMANCE

In this section, we show the effectiveness of XTransformer in detecting vulnerable source code by evaluating both its overall detection performance across all code lengths and its performance within specific code length categories.

1) OVERALL PERFORMANCE

Table 2 showcases the overall detection performance (across all code lengths) of each competing method measured by the metrics of accuracy, precision, recall, and F1 score, evaluated on the Chromium, Debian, and Sun datasets.

For the large-scale datasets, Chromium and Debian, XTransformer achieves the highest accuracy with 0.9549 and 0.9627, respectively. The second-best accuracy is observed with VDSimilar at 0.9344 for the Chromium dataset and with CODE-SMASH at 0.9596 for the Debian dataset. In terms of F1 score, XTransformer also leads with values

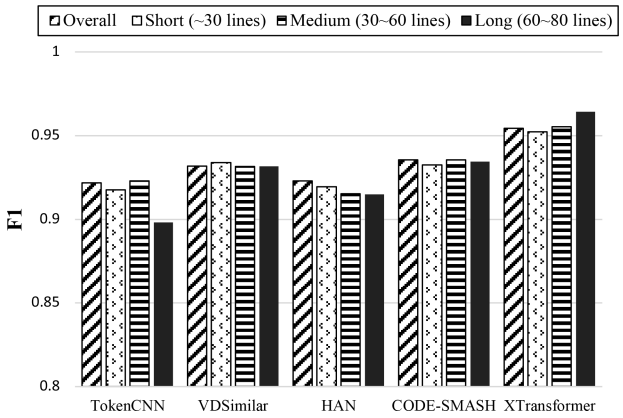

FIGURE 3. Performance comparison of various methods based on F1 score across different code lengths for the Chromium dataset.

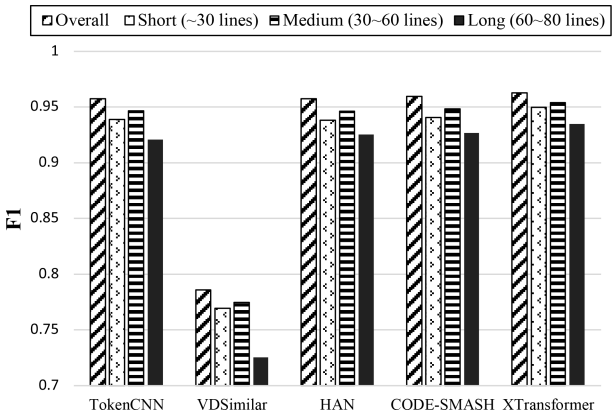

FIGURE 4. Performance comparison of various methods based on F1 score across different code lengths for the Debian dataset.

of 0.9545 for Chromium and 0.9515 for Debian, followed by CODE-SMASH with values of 0.9355 and 0.9469, respectively. While CODE-SMASH has the highest recall at 0.9711 for the Chromium dataset and the highest precision at 0.9940 for the Debian dataset, XTransformer closely follows with a recall of 0.9691 and precision of 0.9864.

For the Sun dataset, a small-scale dataset, XTransformer notably demonstrates the best performance across all metrics compared to other models. For instance, XTransformer achieves the highest accuracy at 0.9470, representing $\times 1.02$ improvement compared to the second-best model. Additionally, XTransformer achieves the highest F1 score at 0.9040, followed by CODE-SMASH at 0.8705, representing $\times 1.04$ improvement over CODE-SMASH.

Across all the datasets, XTransformer consistently demonstrates the best performance, particularly in terms of accuracy and F1 score. This indicates the potential of XTransformer for detecting vulnerabilities.

2) PERFORMANCE ACROSS DIFFERENT CODE LENGTH

To assess each competing method's effectiveness across varying code lengths, we categorized the functions into

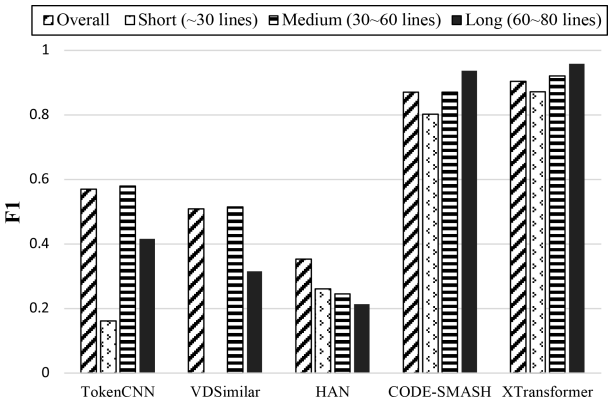

FIGURE 5. Performance comparison of various methods based on F1 score across different code lengths for the Sun dataset.

distinct length groups: short (fewer than 30 lines), medium (30 to 60 lines), long (60 to 80 lines) and evaluated each method on these categorized function groups. Figures 3, 4, and 5 display the results on the Chromium, Debian, and Sun datasets, respectively. Here, the overall performance for each dataset matches the results shown in Table 2.

For the Chromium dataset, XTransformer demonstrates superior performance across all code lengths. It achieves the highest F1 scores of 0.9524, 0.9554, and 0.9644 for short, medium, and long code lengths, respectively. CODE-SMASH, a method designed for handling different code lengths, follows closely with F1 scores of 0.9325, 0.9355, and 0.9344, respectively, showing the second-best performance but not surpassing XTransformer. TokenCNN, HAN, and VDSimilar show reasonable performance for short code lengths but fall behind as the code length increases.

For the Debian dataset, a similar trend is observed where XTransformer achieves the highest F1 scores across all code lengths with 0.9498, 0.9538, and 0.9348 for short, medium, and long code lengths, respectively. CODE-SMASH shows the second-best performance with 0.9406, 0.9483, and 0.9268. Notably, compared to VDSimilar, our method shows performance improvements of $\times 1.24$, $\times 1.23$, and $\times 1.29$ for short, medium, and long code lengths, respectively.

For the Sun dataset, all competing methods, except for CODE-SMASH, show significantly lower performance, with F1 scores below 0.6 across all code lengths. In contrast, our proposal method XTransformer achieves the highest F1 scores of 0.8721, 0.9203, and 0.9579 for short, medium, and long code lengths, respectively. While CODE-SMASH, the second-best model, achieves F1 scores above 0.8, our XTransformer outperforms it, with improvements of $\times 1.09$, $\times 1.06$, and $\times 1.02$ for short, medium, and long sequences, respectively.

Overall, XTransformer consistently shows the highest F1 scores and accuracies across different code lengths and datasets, indicating its effectiveness in handling a variety of code lengths.

TABLE 3. Effectiveness of our training strategy: 'Not Applied' refers to XTransformer without applying our training strategy, while 'Applied' denotes the version of XTransformer with our training strategy applied. The highest values are highlighted in bold.

(A) Chromium				
Training Strategy	Accuracy	Precision	Recall	F1
Not Applied	0.9323	0.9326	0.9349	0.9337
Applied	**0.9549**	**0.9464**	**0.9691**	**0.9545**
(B) Debian				
Training Strategy	Accuracy	Precision	Recall	F1
Not Applied	0.9580	0.9513	**0.9433**	0.9471
Applied	**0.9627**	**0.9864**	0.9195	**0.9515**
(C) Sun				
Training Strategy	Accuracy	Precision	Recall	F1
Not Applied	0.9179	0.8311	0.8843	0.8556
Applied	**0.9470**	**0.9098**	**0.9004**	**0.9040**

C. EFFECTIVENESS OF THE TRAINING STRATEGY

To analyze the effectiveness of our training strategy, we compared the performance of XTransformer trained without the strategy, using a training approach similar to other competing methods, to our final version, which applies our training strategy. Both versions share the same architecture and configuration, differing only in their training approaches.

Table 3 showcases the accuracy, precision, recall, and F1 score of XTransformer without our training strategy (denoted as 'Not Applied' in the table) and with our training strategy (denoted as 'Applied' in the table), evaluated on the Chromium (denoted as (A) Chromium), Debian (denoted as (B) Debian), and Sun (denoted as (C) Sun) datasets.

Notably, even without our training strategy ('Not Applied' in Table 3), XTransformer can outperform or deliver competitive performance compared to other competing methods shown in Table 2. For example, on the Debian dataset, XTransformer without our training strategy outperforms the second-best model, CODE-SMASH, in F1 score. On the Chromium and Sun datasets, it achieves the third-best F1 score. This underscores the effectiveness of our model's structure.

The most significant improvement is observed when our training strategy is applied (denoted as 'Applied' in Table 3), showing performance enhancements across all datasets compared to the 'Not Applied' case. In particular, on the Sun dataset ((C) in the table), which is a small-scale dataset, the effectiveness of our training strategy is remarkable, with an improvement of $\times 1.06$ in F1 score compared to the 'Not Applied' case. Additionally, unlike the 'Not Applied' case, when our training strategy is applied, XTransformer outperforms all other competing methods shown in Table 2.

D. COMPUTATION TIME

To effectively apply deep learning-based source code vulnerability detection models in real-world settings, it is important to achieve both accurate vulnerability detection and efficient computation to provide timely results. Table 4 presents the average inference times required by each competing method to detect source code vulnerabilities.

TABLE 4. Comparison of inference times, measured in seconds, for each competing method. The table displays the average runtime (denoted as mean) and its standard deviation (denoted as std) based on 2,000 randomly selected samples from each of the Chromium and Debian datasets.

Model	Mean (std) time
TokenCNN	0.0015 (0.0008)
VDSimilar	0.0544 (0.0046)
HAN	0.0293 (0.0043)
CODE-SMASH	0.0237 (0.0038)
XTransformer	0.0058 (0.0024)

TABLE 5. Performance comparison of XTransformer with varying numbers of sub-modules in the teacher encoder and student encoder. The best values are boldfaced.

Teacher Encoder	Student Encoder	(A) Chromium Accuracy	Precision	Recall	F1
2	2	0.9329	0.9335	**0.9863**	0.9479
4	4	**0.9549**	**0.9464**	0.9691	**0.9545**
6	6	0.9333	0.9325	0.9847	0.9469
Teacher Encoder	Student Encoder	(B) Debian Accuracy	Precision	Recall	F1
2	2	0.9616	**0.9906**	0.9124	0.9499
4	4	**0.9627**	0.9864	**0.9195**	**0.9515**
6	6	0.9591	0.9732	0.9234	0.9474

The results presented in Table 4 reveal that our XTransformer is faster than the VDSimilar, with computation speeds faster by $\times 9.38$. While faster computation times are observed with TokenCNN, its performance in predicting source code vulnerability falls short for practical use, as detailed in Table 2 and Figs 3, 4, and 5.

Notably, compared to CODE-SMASH and HAN, which consider token- to higher-level features of source code for handling lengthy code sequences, our method is $\times 4.09$ and $\times 5.05$ faster, respectively, while also achieving better detection performance, as discussed in Section IV-B. These results suggest that XTransformer will be better suited for deep learning-based source code vulnerability detection in real-world settings.

E. IMPACT OF THE NUMBER OF SUB-MODULES

Table 5 presents the accuracy, precision, recall, and F1 score for XTransformer with different numbers of sub-modules in both the teacher encoder and student encoder.

The results demonstrate that our optimal configuration, which consists of four sub-modules in both encoders, achieves the best detection performance in terms of accuracy and F1 scores across all datasets, indicating the most balanced overall detection capability. While simpler configurations (2 sub-modules in both encoders) may achieve higher precision or recall, they fail to provide the same level of balanced performance. Conversely, more complex configurations (6 sub-modules in both encoders) show performance degradation in F1 scores, suggesting that increasing the

number of sub-modules can introduce unnecessary model complexity without yielding significant gains in detection performance.

V. CONCLUSION

In this study, we introduced XTransformer, a novel deep learning-based source code vulnerability detector based on source code similarity. Unlike traditional approaches that rely on the Siamese architecture, we proposed a novel split structure based on the transformer, consisting of the teacher encoder, student encoder, and similarity assessment component, for processing the similarity between archived vulnerable source code and new code that needs to be checked for vulnerabilities. This approach employs a hierarchical cross-attention mechanism to effectively compare the similarity between two source codes at different semantic levels, from token-level to higher-level information, making it suitable for handling lengthy and complex codes.

Furthermore, we proposed a training strategy specifically designed to improve our model's capability to extract core features from source codes, even under limited training data conditions, thereby enhancing our model's robustness in detection performance. Through comprehensive experiments, we demonstrated that XTransformer outperforms other state-of-the-art methods across various datasets with different data volumes and source code lengths, marking a significant advancement in applying deep learning to real-world situations.

However, XTransformer has several limitations due to a potential dependency on the characteristics of specific training datasets, such as certain platforms and programming languages. To address these issues, one approach is to enhance the diversity of training datasets by including a wider variety of source code from different languages and platforms.

Despite these challenges, given the increasing prevalence of software-based systems and the growing risk of source code vulnerabilities, we believe that XTransformer can help tackle the escalating challenges in software security.

In our future work, we will explore methods to mitigate the data dependency issue and improve the effectiveness of XTransformer. Additionally, we plan to enhance our method with eXplainable AI (XAI) techniques, such as those described in [54] and [55], to increase the trustworthiness of XTransformer's detection results, making it a more reliable tool for source code vulnerability detection. Furthermore, we plan to incorporate defense mechanisms against model stealing attacks, such as those proposed in [56], to protect our model and enhance its robustness, particularly in real-world deployment scenarios where it may be vulnerable to attacks like [57].

REFERENCES

[1] J. Wang, M. Huang, Y. Nie, and J. Li, "Static analysis of source code vulnerability using machine learning techniques: Asurvey," in *Proc. 4th Int. Conf. Artif. Intell. Big Data (ICAIBD)*, May 2021, pp. 76–86.

[2] N. S. Harzevili, A. B. Belle, J. Wang, S. Wang, Z. Ming, and N. Nagappan, "A survey on automated software vulnerability detection using machine learning and deep learning," 2023, *arXiv:2306.11673*.

[3] A. M. Pitney, S. Penrod, M. Foraker, and S. Bhunia, "A systematic review of 2021 Microsoft exchange data breach exploiting multiple vulnerabilities," in *Proc. 7th Int. Conf. Smart Sustain. Technol. (SpliTech)*, Jul. 2022, pp. 1–6.

[4] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: Aframework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2244–2258, Jul. 2022.

[5] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.

[6] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Dec. 2017, pp. 1298–1302.

[7] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "*Devign*: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–11.

[8] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically detecting Software Vulnerabilities using deep graph neural network," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–33, Apr. 2021.

[9] M. Gu, H. Feng, H. Sun, P. Liu, Q. Yue, J. Hu, C. Cao, and Y. Zhang, "Hierarchical attention network for interpretable and fine-grained vulnerability detection," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, May 2022, pp. 1–6.

[10] K. Kontogiannis, M. Galler, and R. DeMori, "Detecting code similarity using patterns," in *Proc. Work. Notes 3rd Workshop AI Softw. Eng.*, vol. 6, 1995, pp. 1–7.

[11] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An automated vulnerability detection system based on code similarity analysis," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 201–213.

[12] H. Sun, L. Cui, L. Li, Z. Ding, Z. Hao, J. Cui, and P. Liu, "VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches," *Comput. Secur.*, vol. 110, Nov. 2021, Art. no. 102417.

[13] S. Han, H. Nam, J. Kang, K. Kim, S. Cho, and S. Lee, "CODE-SMASH: Source-code vulnerability detection using Siamese and multi-level neural architecture," *IEEE Access*, vol. 12, pp. 102492–102504, 2024.

[14] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–12.

[15] *Common Vulnerabilities and Exposures*. Accessed: 2024. [Online]. Available: https://www.cve.org/

[16] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 1, Jun. 2005, pp. 539–546.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. U. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2017, pp. 6000–6010.

[18] P. Khosla, P. Teterwak, C.Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," in *Proc. NIPS*, vol. 33, 2020, pp. 18661–18673.

[19] S. Jeon and H. K. Kim, "AutoVAS: An automated vulnerability analysis system with a deep learning approach," *Comput. Secur.*, vol. 106, Jul. 2021, Art. no. 102308.

[20] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. 3rd Int. Conf. Learn. Represent.*, San Diego, CA, USA, 2015.

[21] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[22] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCfinder: D-CCfinder," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, vol. 45, May 2007, pp. 106–115.

[23] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Inf. Softw. Technol.*, vol. 55, no. 7, pp. 1165–1199, Jul. 2013.

[24] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2016, pp. 87–98.
[25] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School Comput. TR*, vol. 541, no. 115, pp. 64–68, 2007.
[26] G. Pirró and J. Euzenat, "A feature and information theoretic framework for semantic similarity and relatedness," in *Proc. 9th Int. Semantic Web Conf. Semantic Web (ISWC)*, Shanghai, China. Cham, Switzerland: Springer, Nov. 2010, pp. 615–630.
[27] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity—A survey," *ACM Comput. Surveys*, vol. 54, no. 2, pp. 1–37, 2021.
[28] J.-Y. Jiang, M. Zhang, C. Li, M. Bendersky, N. Golbandi, and M. Najork, "Semantic text matching for long-form documents," in *Proc. World Wide Web Conf.*, May 2019, pp. 795–806.
[29] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Proc. Conf. Empirical Methods Natural Lang. Process., 9th Int. Joint Conf. Natural Lang. Process. (EMNLP-IJCNLP)*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds., Hong Kong, 2019, pp. 3982–3992.
[30] S. V. Moravvej, M. Joodaki, M. J. M. Kahaki, and M. S. Sartakhti, "A method based on an attention mechanism to measure the similarity of two sentences," in *Proc. 7th Int. Conf. Web Res. (ICWR)*, May 2021, pp. 238–242.
[31] S. Peng, H. Cui, N. Xie, S. Li, J. Zhang, and X. Li, "Enhanced-RCNN: An efficient method for learning sentence similarity," in *Proc. Web Conf.*, vol. 1, Apr. 2020, pp. 2500–2506.
[32] J. Xie, K. Cai, L. Kong, J. Zhou, and W. Qu, "Automated essay scoring via pairwise contrastive regression," in *Proc. 29th Int. Conf. Comput. Linguistics*, N. Calzolari, C.-R. Huang, H. Kim, J. Pustejovsky, L. Wanner, K.-S. Choi, P.-M. Ryu, H.-H. Chen, L. Donatelli, H. Ji, S. Kurohashi, P. Paggio, N. Xue, S. Kim, Y. Hahm, Z. He, T. K. Lee, E. Santus, F. Bond, and S.-H. Na, Eds., Gyeongju, Republic of Korea, Oct. 2022, pp. 2724–2733.
[33] P. Neculoiu, M. Versteegh, and M. Rotaru, "Learning text similarity with Siamese recurrent networks," in *Proc. 1st Workshop Represent. Learn.*, 2016, pp. 148–157.
[34] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2, Jun. 2006, pp. 1735–1742.
[35] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. 37th Int. Conf. Mach. Learn.*, 2020, pp. 1597–1607.
[36] X. Chen and K. He, "Exploring simple Siamese representation learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2021, pp. 15745–15753.
[37] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 9726–9735.
[38] Z. Wu, S. Wang, J. Gu, M. Khabsa, F. Sun, and H. Ma, "CLEAR: Contrastive learning for sentence representation," 2020, *arXiv:2012.15466*.
[39] Y. Meng, C. Xiong, P. Bajaj, S. Tiwary, P. Bennett, J. Han, and X. Song, "COCO-LM: Correcting and contrasting text sequences for language model pretraining," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds. Red Hook, NY, USA: Curran Associates, 2021, pp. 23102–23114.
[40] T. Gao, X. Yao, and D. Chen, "SimCSE: Simple contrastive learning of sentence embeddings," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, M.-F. Moens, X. Huang, L. Specia, and S. W.-T. Yih, Eds., Santo Domingo, Dominican Republic, 2021, pp. 6894–6910.
[41] Z. Tang, M. Y. Kocyigit, and D. T. Wijaya, "AugCSE: Contrastive sentence embedding with diverse augmentations," in *Proc. 2nd Conf. Asia-Pacific Chapter Assoc. Comput. Linguistics, 12th Int. Joint Conf. Natural Lang. Process.*, Y. He, H. Ji, S. Li, Y. Liu, and C.-H. Chang, Eds., Nov. 2022, pp. 375–398.
[42] Y.-S. Chuang, R. Dangovski, H. Luo, Y. Zhang, S. Chang, M. Soljacic, S.-W. Li, S. Yih, Y. Kim, and J. Glass, "DiffCSE: Difference-based contrastive learning for sentence embeddings," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, M. Carpuat, M.-C. de Marneffe, and I. V. Meza Ruiz, Eds., 2022, pp. 4207–4218.
[43] L. Wang, N. Yang, X. Huang, B. Jiao, L. Yang, D. Jiang, R. Majumder, and F. Wei, "Text embeddings by weakly-supervised contrastive pre-training," 2022, *arXiv:2212.03533*.
[44] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
[45] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.
[46] J. Ba, J. Kiros, and G. Hinton, "Layer normalization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016.
[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[48] L. Liu, X. Liu, J. Gao, W. Chen, and J. Han, "Understanding the difficulty of training transformers," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds., 2020, pp. 5747–5763.
[49] J. Turton, R. E. Smith, and D. Vinson, "Deriving contextualised semantic features from BERT (and other transformer model) embeddings," in *Proc. 6th Workshop Represent. Learn.*, A. Rogers, I. Calixto, I. Vulić, N. Saphra, N. Kassner, O.-M. Camburu, T. Bansal, and V. Shwartz, Eds., 2021, pp. 248–262.
[50] Y.-S. Chuang, Y. Xie, H. Luo, Y. Kim, J. R. Glass, and P. He, "DoLa: Decoding by contrasting layers improves factuality in large language models," in *Proc. 12th Int. Conf. Learn. Represent.*, 2024.
[51] G. Jawahar, B. Sagot, and D. Seddah, "What does BERT learn about the structure of language?" in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, A. Korhonen, D. Traum, and L. Màrquez, Eds., Florence, Italy, Jul. 2019, pp. 3651–3657.
[52] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.
[53] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent.*, Y. Bengio and Y. LeCun, Eds., San Diego, CA, USA, May 2015.
[54] S. Lee and S. Han, "Libra-CAM: An activation-based attribution based on the linear approximation of deep neural nets and threshold calibration," in *Proc. 31st Int. Joint Conf. Artif. Intell.*, L. D. Raedt, Ed., Jul. 2022, pp. 3185–3191.
[55] S. Han, J. Lee, and S. Lee, "Activation fine-tuning of convolutional neural networks for improved input attribution based on class activation maps," *Appl. Sci.*, vol. 12, no. 24, p. 12961, Dec. 2022.
[56] J. Lee, S. Han, and S. Lee, "Model stealing defense against exploiting information leak through the interpretation of deep neural nets," in *Proc. 31st Int. Joint Conf. Artif. Intell.*, L. D. Raedt, Ed., Jul. 2022, pp. 710–716.
[57] J. Lee, S. Han, and S. Lee, "SwiftThief: Enhancing query efficiency of model stealing by contrastive learning," in *Proc. 33rd Int. Joint Conf. Artif. Intell.*, K. Larson, Ed., Aug. 2024, pp. 422–430.

SUNGMIN HAN received the B.S. degree in computer engineering from Kwangwoon University, Seoul, South Korea, in 2021, and the M.S. degree in cybersecurity from Korea University, Seoul, in 2024. He is currently pursuing the Ph.D. degree with the School of Cybersecurity, Korea University. His research interests include trustworthy AI, with a focus on explainable AI, model stealing attacks and defenses, adversarial robustness, and backdoor robustness.

MIJU KIM received the B.S. degree in convergence security engineering from Sungshin Women's University, Seoul, South Korea, in 2024. She is currently pursuing the M.S. degree with the School of Cybersecurity, Korea University, South Korea. Her main research interests include AI for security, including anomaly detection and explainable AI.

JAESIK KANG received the bachelor's degree in computer engineering and the master's degree in computer engineering from Chungnam National University, South Korea, in 2015 and 2020, respectively. Since July 2022, he has been affiliated with LIG Nex1, a leading defense industry in South Korea, as a Researcher of cyber security. His research interests include AI, cyber security, and software engineering.

KWANGSOO KIM received the B.S. degree in information and computer engineering and the Ph.D. degree in computer engineering from Ajou University, Republic of Korea, in 2009 and 2017, respectively. Since January 2017, he has been affiliated with LIG Nex1, a leading defense industry in South Korea, as a Researcher of cyber security. His research interests include network security and cyber warfare, especially cyber training systems.

SEUNGWOON LEE received the B.S. degree in information and computer engineering, the M.S. degree in software engineering, and the Ph.D. degree in computer engineering and security from Ajou University, Republic of Korea, in 2015, 2017, and 2022, respectively. Since January 2022, he has been affiliated with LIG Nex1, a leading defense industry in South Korea, as a Researcher of cyber security. His research interests include cyber warfare and cyber resilience.

SANGKYUN LEE (Member, IEEE) received the B.S. and M.S. degrees in computer science from Seoul National University, Seoul, South Korea, in 2003 and 2005, respectively, and the M.S. and Ph.D. degrees in computer science from the University of Wisconsin–Madison, Madison, WI, USA, in 2008 and 2011, respectively.

From 2011 to 2014, he was a Postdoctoral Fellow with the Collaborative Research Center SFB876 at TU Dortmund University, Germany. From 2015 to 2016, he was a Project Leader with the Collaborative Research Center SFB876, leading the C1 division. From 2017 to 2019, he was an Assistant Professor with the Division of Computer Science, College of Computing, Hanyang University ERICA, Ansan, South Korea. From 2020 to 2021, he was an Assistant Professor with the School of Cybersecurity, Korea University, Seoul, South Korea. Since 2022, he has been an Associate Professor with the School of Cybersecurity, Korea University. His main research interests include trustworthy AI, model compression, and AI for security.