IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 15 June 2024, accepted 17 July 2024, date of publication 22 July 2024, date of current version 1 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3432323

RESEARCH ARTICLE

CODE-SMASH: Source-Code Vulnerability Detection Using Siamese and Multi-Level Neural Architecture

SUNGMIN HAN[1], HYUNKYUNG NAM[1], JAESIK KANG[2], KWANGSOO KIM[2], SEUNGJAE CHO[2], AND SANGKYUN LEE[1]

[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 in part by Korea University Grant and in part by LIG Nex1.

ABSTRACT The rapid evolution of software development, propelled by competitive demands and the continuous integration of new features, frequently leads to inadvertent security oversights. Traditional security practices, often reactive in nature, primarily focus on identifying known vulnerabilities, creating a significant shortfall in detecting emergent, zero-day threats. This paper introduces CODE-SMASH, a novel deep learning-based source code vulnerability detector that utilizes a Siamese neural network with a hierarchical architecture integrating BiGRU and attention mechanisms. Our experiments using real-world datasets, specifically the Chromium and Debian datasets, demonstrate CODE-SMASH's superiority over existing methods. It achieves significant improvements in detection performance across all key metrics, including accuracy, precision, recall, and F1-score, with average improvements of approximately 8.3%, 11.6%, 27.75%, and 17.7%, respectively, compared to the best-performing existing methods in our experiments. Moreover, CODE-SMASH shows its superior capability in handling complex and lengthy code sequences, with performance improvements for long-length code (60 to 80 lines) in F1 scores of 4.53 percentage points on the Chromium dataset and 5.62 percentage points on the Debian dataset compared to the second-best model's performance. We believe our research makes a significant contribution to the field of automated vulnerability detection by providing a high-precision solution to the growing challenges in software security. Furthermore, based on our findings, we anticipate that future research could enhance CODE-SMASH by expanding its generalizability to various programming languages and reducing computational demands to improve efficiency.

INDEX TERMS Code similarity, hierarchical attention network, Siamese neural network, source code vulnerability detection.

I. INTRODUCTION

Detecting potential software vulnerabilities is becoming increasingly crucial in safeguarding against cyber attacks, especially with the surge in software usage and the expansion of the Internet of Things (IoT) ecosystem [1], [2]. These developments have broadened the attack surface for cyber-criminals, creating more avenues to exploit vulnerabilities. In particular, source code vulnerabilities, which are flaws or security weaknesses in the source code underlying software that can be exploited by malicious attackers, are significant because they can lead to severe security breaches.

The 2017 Equifax data breach, one of the most significant cybersecurity incidents, illustrates the dangers of software

The associate editor coordinating the review of this manuscript and approving it for publication was Ganesh Naik.

vulnerabilities. This breach, which exposed the personal information of more than 143 million people, was the result of source code vulnerabilities in the company's web application software. The impact of this breach was substantial, causing financial losses and damaging trust [3].

To help prevent software vulnerabilities and track them in advance, cybersecurity experts often rely on the Common Vulnerabilities and Exposures (CVE) database [4]. However, the CVE database only lists vulnerabilities that have already been identified, leaving out so-called zero-day attacks, which exploit vulnerabilities that have not yet been discovered. This limitation creates a critical security gap in defending against zero-day attacks [5], [6], [7].

Vulnerability detection in source code primarily employs dynamic and static analysis. Dynamic analysis, performed while the source code is running, can detect vulnerabilities in actual execution environments. However, it demands a complex setup and is resource-intensive. Static analysis, conversely, involves examining the source code without execution. This analysis is effective in early development stages and crucial for ensuring software security and integrity by allowing early detection and remediation of vulnerabilities. In this paper, we focus on static analysis scenarios.

Traditional methods for static code analysis include rule-based techniques, often relying on manual code reviews [8]. These methods, while useful, struggle with the complexity and dynamism of modern software vulnerabilities [9]. Therefore, more sophisticated methods, including deep learning-based approaches, have recently been proposed and shown their effectiveness [10], [11].

However, we find that existing deep learning-based approaches are also insufficient for detecting vulnerabilities in modern software's source codes. Due to the increasing complexity and diversity of modern software, its source codes are often lengthy and intricate. Therefore, it is crucial to handle such code accurately to identify vulnerabilities without generating false positives and false negatives, making it a practical detection tool for modern software vulnerabilities. Existing methods, however, struggle with this variability.

To overcome the limitations of existing deep learning methods in vulnerability detection, we propose CODE-SMASH, a similarity-based vulnerability detection model that identifies vulnerabilities by comparing the similarity of information extracted from the hierarchical structure of two source codes. Our model utilizes the architecture of a Siamese neural network (SNN), designed to compare the similarity between two inputs using twin sub-networks [12], [13], and incorporates a hierarchical structure with bidirectional gated recurrent unit (BiGRU) and attention mechanisms [14] as the sub-networks of the Siamese architecture to transform source codes into dense vectors that represent their hierarchical information. By transforming source codes into dense vectors that represent their hierarchical information and comparing them, our model can derive similarity scores between a new code and an archived code with known vulnerabilities.

Our contributions are as follows:

- We introduce a hierarchical neural architecture for code vulnerability detection, leveraging both token- and statement-level information of source code.
- Our approach, effective for long-length code data, addresses complexities in extensive code bases, outperforming existing methods by 34.4% on average in terms of F1-score for detecting vulnerabilities.
- The employment of a Siamese neural network enhances the model's performance, demonstrating its efficiency and adaptability in various data environments.

II. RELATED WORK

This section briefly reviews foundational research and recent advancements in source code vulnerability detection, focusing on static analysis scenarios. In addition, we explain methods for vector representation of source code and Siamese neural networks to facilitate understanding of our proposed method.

A. SOURCE CODE VULNERABILITY DETECTION

Source code vulnerability detection aims to identify security flaws within source codes that could be exploited by malicious actors. The methods for detecting source code vulnerabilities can be categorized into rule-based, classifier-based, and similarity-based approaches.

1) RULE-BASED METHODS

Rule-based vulnerability detection is a traditional approach that relies on identifying specific patterns or signatures associated with known vulnerabilities, which refer to vulnerabilities already reported and documented, such as those in the Common Vulnerabilities and Exposures (CVE) database [4].

Rule-based methods use predefined patterns created by domain experts to detect vulnerabilities in codebases. Examples include Flawfinder [15], RATS [16], and Checkmarx [17]. While these methods are effective at identifying known vulnerabilities, they require significant manual effort for pattern creation and maintenance, and they often produce high rates of false positives and false negatives for vulnerabilities that do not match their predefined patterns. Practical applications, such as Fortify SCA [18] and SonarQube [19], which employ rule-based methods, also struggle with new and emerging threats that are not yet included in their pattern databases.

2) CLASSIFIER-BASED METHODS

To address the limitations of rule-based methods, deep learning-based approaches have been proposed in recent years. By leveraging deep learning models that can automatically learn and extract vulnerability patterns from large datasets, deep learning-based source code vulnerability classifiers have demonstrated superior detection performance compared to traditional rule-based methods [20], [21], [22], [23], [24], [25], [26].

For instance, Li et al. proposed SySeVR [20], a deep learning-based framework for detecting software vulnerabilities in C/C++ programs. To capture vulnerability patterns reflecting source code's syntax and semantic meanings, they proposed a bidirectional recurrent neural network (BiRNN) based classification model. They demonstrated that the model could successfully detect vulnerabilities in real-world software products such as Libav, Seamonkey, Thunderbird, and Xen, uncovering previously unknown vulnerabilities. Russell et al. [21] introduced a convolutional neural network (CNN)-based method for vulnerability detection, which captures vulnerability patterns by directly applying the model to lexed source code. Through their evaluation conducted using source codes from both real software packages and the NIST SATE IV dataset [27], [28], they validated the superiority of their model. Unlike methods that utilize only CNN or RNN networks, Wu et al. [22] developed a hybrid model combining convolutional neural networks (CNNs) and long short-term memory (LSTM) networks to capture both local patterns and long-range dependencies.

Gu et al. [25] introduced a hierarchical classification model consisting of two-level bidirectional gated recurrent unit (BiGRU) layers with attention mechanisms [29] to extract vulnerability patterns from the source code's statement and token-level attributes. They used abstract syntax tree (AST) processing and token embedding for feature extraction, primarily relying on statement-level attributes for final decision-making. Other notable deep learning approaches include the work by Kalra et al. [26] who proposed Zeus, a deep learning framework that utilizes an ensemble of neural networks to detect vulnerabilities in smart contracts, showcasing the versatility of deep learning in various coding environments.

Despite advances in classifier-based source code vulnerability detection methods, approaches designed to handle lengthy and complex source code, which are essential for effectively detecting vulnerabilities in modern software, are still lacking.

3) SIMILARITY-BASED METHODS

Code duplication, or copying code, increases maintenance efforts and the risk of vulnerabilities. Studies indicate that 5% to 20% of software components may consist of duplicated code [30], [31], [32], [33], [34], implying a higher likelihood of additional vulnerabilities if a defect is found in one segment.

Similarity-based vulnerability detection methods focus on finding vulnerabilities by comparing the semantic similarity between analyzed code and known vulnerable code. Unlike classifier-based methods, this approach emphasizes analyzing code attributes and is adaptable to new and unidentified vulnerabilities [6], [7], [35]. These methods are particularly effective at identifying vulnerabilities with behaviors or structures similar to those previously detected.

For example, Li et al. [36] proposed VulPecker, which quantifies the similarity between two source codes using machine learning techniques such as support vector machines (SVM), while Li et al. [37] introduced a graph-matching network model to compute similarity scores through a cross-graph attention-based matching mechanism, demonstrating effectiveness in control-flow graph-based function similarity search. More recently, Sun et al. [38] proposed VDSimilar, a Siamese neural network [12] with bidirectional long short-term memory (BiLSTM) networks and attention mechanisms [29], which compares token-level attributes between two source codes to detect vulnerabilities. They showed that with a relatively small dataset, their similarity-based detection model can outperform classifier-based approaches.

Although these similarity-based methods have shown remarkable detection performance, they do not fully utilize information from the hierarchical structure of source code, including token statement, and function-level attributes, all of which are important for analyzing vulnerability patterns.

B. VECTOR REPRESENTATION OF CODE

Transforming source code into vector representations is essential for applying deep learning techniques to software analysis. Traditional approaches involved tokenizing code into a bag-of-words model, but this often failed to capture the semantic and structural nuances of programming languages.

More advanced techniques use embeddings, where code snippets are represented as vectors in high-dimensional space. Methods like word2vec [39] train neural networks on large codebases to generate vector representations that reflect the semantic information of source code, while graph-based embedding methods preserve more structural information. To preserve structural information, graph-based embedding methods process graph structures of source code, such as abstract syntax trees (ASTs), using graph neural networks and capture relationships and dependencies between code parts [23]. This method is advantageous in that it can utilize the structural information of source codes, but it has the drawback of requiring a lot of preprocessing effort. Recent studies by Alon et al. [40] on code2vec and Pradel and Sen [41] on DeepBugs highlight the effectiveness of using vector representations for code understanding and bug detection.

In our framework, we avoided using graph structures, such as ASTs or other tree-structured static analysis information, due to their extended preprocessing burden. Instead, we directly capture the hierarchical structure of source code, aligning with our goal of efficient processing while maintaining a deep understanding of code semantics.

C. SIAMESE NEURAL NETWORK

Siamese neural networks (SNNs) [12], [13] are powerful tools for similarity detection in natural text [42] and code analysis [14], [43], [44]. SNNs consist of two identical sub-networks with the same configuration, parameters, and weights. These sub-networks process input in parallel, and

TABLE 1. Comparison of vulnerability detection methods.

Model	TokenCNN [21]	CNN+LSTM [22]	VDSimilar [38]	HAN [25]	CODE-SMASH
No need for compilation		✓			✓
Handles long-length code well	✓			✓	✓
Works with small training data	✓		✓		✓
Types of source code features	tokens	tokens	tokens	tokens, statements	tokens, statements, functions

the outputs are combined to compute a similarity metric, such as cosine similarity. The key idea behind SNNs is to learn a similarity function that can compare two inputs and determine whether they are similar or not [12].

To train SNNs, the network is provided with pairs of inputs labeled as similar or dissimilar. During training, the network adjusts its parameters to minimize the difference between the outputs of similar pairs and maximize the difference between the outputs of dissimilar pairs [12], [13]. In vulnerability detection, SNNs compare a given piece of code against known examples of vulnerabilities, learning complex patterns and anomalies that signify potential security risks [38], [43], [44].

Table 1 compares our model, CODE-SMASH, against state-of-the-art models in source code-based vulnerability detection. Unlike models considering only tokens or a combination of tokens and statements, CODE-SMASH extends to function-level analysis, offering a more nuanced examination of the source code. This granularity enhances the model's performance with smaller datasets and longer code lengths, showcasing its superior capability in vulnerability detection compared to other baseline methods.

III. METHODOLOGY

In this section, we describe our framework for detecting source code vulnerabilities. This framework involves constructing pairs of source codes and analyzing them with a similarity-based neural detector named CODE-SMASH that leverages information extracted from a hierarchical code structure, including tokens and statements.

Our framework can serve as a valuable tool for software developers, empowering them to identify source code vulnerabilities during their development process and take proactive measures to address potential security flaws. Additionally, it can be deployed to identify vulnerabilities within open-source projects. For example, through the analysis of historical data from web server software like Apache, it can identify recurring patterns and similarities in code modifications that have previously led to security breaches.

A. CONSTRUCTING SOURCE CODE PAIRS

We detail a strategy for pairing source codes within our detection framework, focusing on source code preprocessing. This strategy involves removing comments and applying tokenization and vectorization to the codes to help our detector identify source code vulnerabilities effectively.

1) REMOVE COMMENTS

Given the nature of deep learning-based models, which rely on data to learn and understand aspects relevant to their specific tasks, their performance depends on the quality of data. Therefore, refining the data by removing irrelevant information is crucial for enabling the model to focus on task-specific information, thus ensuring effective performance.

In the context of source code vulnerability detection, which focuses on the executable parts of source code where potential vulnerabilities might exist, comments within the source code—intended to explain functionality without affecting execution—can be considered irrelevant to vulnerability detection and may act as noise hindering the accurate identification of vulnerabilities. Therefore, we eliminate comments from a source code to retain solely the executable segments of the code. Especially since comments in programming languages adhere to specific patterns (*e.g.*, //, /*...*/), we use regular expressions to identify and remove these patterns, effectively purging comments from the code.

2) SOURCE CODE TOKENIZATION

Tokens are the smallest units used to represent source code or text, and deep learning-based models analyze the full context of source code or text starting from these token-level units. Thus, employing suitable tokenization methods for source code can enhance the model's ability to interpret and process the underlying structure and semantics of source code.

Unlike traditional tokenization methods in natural language processing, which identify tokens as words separated by spaces, our approach for tokenizing source code extends this simple identification of spaces. We incorporate *CamelCase* naming convention [45], common in software development, as essential in defining the smallest units. This tokenization ensures that semantically related characters are grouped as one token, preserving the integral meaning within the code's syntax. It enables a more in-depth source code analysis, leading to the precise identification of patterns and potential vulnerabilities, as evidenced in [46] and [47]. Through our source code tokenization process, we convert source code, with comments removed, into a set of tokens.

3) SOURCE CODE VECTORIZATION

As deep learning models operate on numerical data, tokenized source codes need to be transformed into a numerical format (vectors). Therefore, we convert the tokenized codes into their corresponding vectorized forms.

For this purpose, we use Word2Vec [39], [48], a popular vectorization technique in natural language processing. The fundamental formulation of Word2Vec defines $p(w_{t+j}|w_t)$ utilizing the softmax function to maximize the following expression:

$$p(w_O|w_I) = \frac{\exp({v'}_{w_O}^T v_{w_I})}{\sum_{w=1}^{W} \exp({v'}_w^T v_{w_I})},$$

where w_O and w_I represent the center word and surrounding word, respectively. The matrices v_w and ${v'}_w$ serve as embedding matrices employed during input and between the hidden layer and the output layer to represent the vectors of w, with W denoting the total number of words in the vocabulary [48]. Maximizing this expression involves augmenting the similarity between the word vectors of the center and surrounding words, thereby enhancing their dot product. Conversely, decreasing the similarity with words beyond the context window diminishes their association with the center word, thereby effectively capturing the semantic relationships between words. Following Word2Vec, we train a word vector model for source code vectorization and utilize the embedding matrix from the trained model to transform each token into a vector representation.

Finally, we organize the vectorized tokens into a hierarchical structure that mirrors the inherent structure of the code. This is achieved by arranging the vectorized tokens according to the source code's statements, using newline characters as delimiters to guide the grouping.

4) CONSTRUCTION OF CODE-PAIRS

Our detection strategy aims to identify vulnerable source code by assessing the semantic likeness between source codes. To support this strategy, we construct pairs of source codes, each comprised of vectorized tokens, similar to the approach described in [38].

Specifically, for training our detector, we construct a dataset of code pairs that includes both pairs of vulnerable codes and combinations of vulnerable and non-vulnerable codes. We label pairs consisting solely of vulnerable code segments with a 1 to highlight their similarities. Conversely, pairs that mix vulnerable and non-vulnerable codes are labeled with a 0 to emphasize their differences. This approach not only facilitates the detection of vulnerable codes but also enables the uncovering of similarities across various versions of vulnerable code, considering that Common Vulnerabilities and Exposures (CVEs) can span multiple versions.

B. SIMILARITY-BASED DETECTION MODEL

We propose CODE-SMASH, a Siamese multi-depth attention-based hierarchical recurrent neural detector designed for detecting vulnerabilities in source code by analyzing code similarities. It processes a pair of source codes, denoted as $\{c^n\}_{n=1}^2$ as input, where each $c^n \in \mathbb{R}^{\ell \times k \times m}$ is a collection of multiple statements, defined by $c^n := \{s_i^n\}_{i=1}^{\ell}$. Each statement s_i^n is comprised of a sequence of vectorized tokens, represented as $s_i^n := \{t_{i,j}^n\}_{j=1}^k$, with each vectorized token $t_{i,j}^n \in \mathbb{R}^m$. CODE-SMASH evaluates the similarity between these source codes to identify potential vulnerabilities.

The core of CODE-SMASH is based not only on a Siamese architecture, known for its effectiveness in comparing the similarity between two source codes [42], [44], but also on a hierarchical feature encoder designed to extract features across various levels of a source code. This core facilitates a multi-level analysis, enabling the detailed examination and comparison of two source codes at multiple levels of abstraction, from the micro-level details of statements to the macro-level organization of functions. The overall architecture of CODE-SMASH is shown in Fig. 1.

1) FEATURE ENCODER

Following the Siamese neural network (SNN) framework, CODE-SMASH utilizes twin sub-networks with identical architecture and shared weights. Each sub-network consists of two key components: the statement-level encoder and the function-level encoder, each engineered to extract features from a source code at their respective levels of analysis.

a: STATEMENT-LEVEL ENCODER

For any given source code $c^n := \{s_i^n\}_{i=1}^{\ell}$, the statement-level encoder processes each statement $s_i^n := \{t_{i,j}^n\}_{j=1}^k$ for $1 \le i \le \ell$, by aggregating its sequence of vectorized tokens $\{t_{i,j}^n\}_{j=1}^k$. This procedure is aimed at extracting features at the statement level. To achieve this, the encoder uses a bidirectional Gated Recurrent Unit (BiGRU) along with an additive self-attention mechanism [29].

The BiGRU is composed of two GRU units designed to analyze the token sequence from both forward and reverse directions, thereby capturing dependencies within the sequence. The first GRU unit processes the sequence from the start to the end token in a forward direction, capturing forward contextual information, while the other unit processes it from the end to the start token in a backward direction, capturing backward contextual information. More formally, the forward GRU generates a sequence of forward hidden states denoted by $\overrightarrow{h_{i,j}^n}$ for each vectorized token $t_{i,j}^n$, resulting in the sequence $\{\overrightarrow{h_{i,1}^n}, \ldots, \overrightarrow{h_{i,k}^n}\}$. Similarly, the backward GRU generates a sequence of backward hidden states denoted by $\overleftarrow{h_{i,j}^n}$ also for each vectorized token, but in reverse order, yielding the sequence $\{\overleftarrow{h_{i,1}^n}, \ldots, \overleftarrow{h_{i,k}^n}\}$. Here, each hidden state $\overrightarrow{h}_{i,j}^n$ and $\overleftarrow{h}_{i,j}^n$ is a vector in $\mathbb{R}^d$, for $1 \le j \le k$, with each corresponding to the j-th vectorized token $t_{i,j}^n$ of the i-th statement in the n-th source code. By using these hidden states, we derive statement-level source code features.

Since not all hidden states are equally important for source code comparison, we integrate an additive self-attention mechanism alongside the BiGRU to reflect the importance of each hidden state for the comparison process. Specifically, we generate a concatenated hidden state $h_{i,j}^n$ by combining the hidden states from both the forward and backward GRUs:

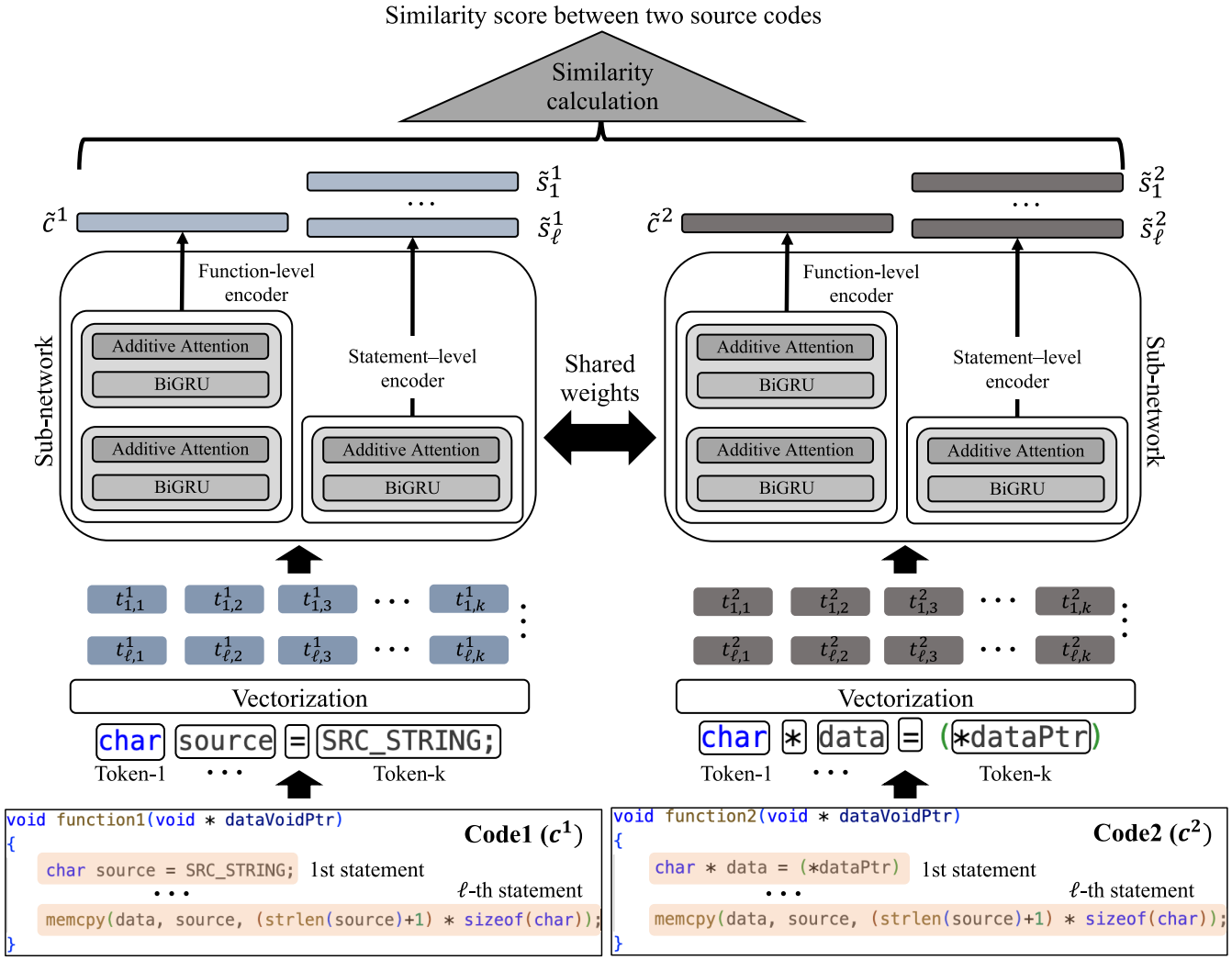

FIGURE 1. Vulnerability detection with CODE-SMASH. The term 'shared weights' means that two sub-networks share the same configuration and weights.

$h^n_{i,j} = \left[\overrightarrow{h^n_{i,j}}; \overleftarrow{h^n_{i,j}}\right]$, where $h^n_{i,j} \in \mathbb{R}^{2d}$ for $j \in [1, k]$. The importance of each representation $h^n_{i,j}$ is assessed using an additive self-attention layer that consists of two fully connected layers, followed by the softmax function:

$$v_j = \tanh(W^T h^n_{i,j} + b), \tag{1}$$

$$\alpha_j = \frac{\exp(v_j^T q)}{\sum_j \exp(v_j^T q)}, \tag{2}$$

where $v_j \in \mathbb{R}^{2d}$ represents the output of the first fully connected layer, with $W \in \mathbb{R}^{2d \times 2d}$ and $b \in \mathbb{R}^{2d}$, indicating the layer's weights and bias, respectively. The parameter $q \in \mathbb{R}^{2d}$ corresponds to the weights of another fully connected layer that is used in the calculation of the attention score $\alpha_j \in \mathbb{R}$, which determines the importance of the j-th concatenated hidden state $h^n_{i,j}$. After calculating the attention scores, we integrate them with each concatenated hidden state to get statement-level source code features:

$$\tilde{s}^n_i = \sum_{j=1}^{k} \alpha_j h^n_{i,j}, \tag{3}$$

where $\tilde{s}^n_i \in \mathbb{R}^d$ represents the extracted statement-level source code feature for each i within the range of $[1, \ell]$ in n-th source code. This aggregation effectively synthesizes the information across all tokens within a statement, weighted by their respective attention scores, to capture the essence of each statement's contribution to the overall code semantics.

b: FUNCTION-LEVEL ENCODER

The function-level encoder is designed to extract function-level features from statement-level source code features, thereby deepening the analysis of the source code. This process aims to provide a nuanced comprehension of function-level attributes and their connections, improving the detector's

ability to identify and interpret complex code patterns and vulnerabilities.

Expanding on the design of the statement-level encoder, the function-level encoder incorporates an additional BiGRU layer and an additive self-attention layer. The architecture is structured hierarchically into two main components: the first component is tasked with the extraction of statement-level features, and the second component, positioned above the first, concentrates on deriving function-level features from these extracted statement-level features.

Given statement-level source code features $\hat{s}_i^n \in \mathbb{R}^d$, ranging from $1 \leq i \leq \ell$ as processed by the statement-level component of the function-level encoder, the function-level component further processes these features similar to the statement-level processing. In the function-level component, the forward and backward GRU units generate sequences of hidden states $\overrightarrow{\hat{h}_i^n}$ and $\overleftarrow{\hat{h}_i^n}$, respectively. Each state corresponds to the i-th statement-level source code features $\hat{s}_i^n$ in the n-th source code. and these hidden states are vectors in $\mathbb{R}^{d/2}$. Subsequently, the hidden states from both the forward and backward GRUs are concatenated to form a comprehensive hidden representation $\hat{h}_i^n = \left[\overrightarrow{\hat{h}_i^n}; \overleftarrow{\hat{h}_i^n}\right]$, where each $\hat{h}_i^n$ is a vector in $\mathbb{R}^d$. Utilizing an additive self-attention layer, we calculate function-level attention scores for the concatenated hidden states $\hat{h}_i^n$:

$$\hat{v}_i = \tanh(\hat{W}^T \hat{h}_i^n + \hat{b}), \tag{4}$$

$$\hat{\alpha}_i = \frac{\exp(\hat{v}_i^T \hat{q})}{\sum_j \exp(\hat{v}_i^T \hat{q})}, \tag{5}$$

where $\hat{W} \in \mathbb{R}^{d \times d}$, $\hat{b} \in \mathbb{R}^d$ and $\hat{q} \in \mathbb{R}^d$ represent the parameters of the fully connected layer, and the $\hat{\alpha}_i \in \mathbb{R}$ is the function-level attention score. Finally, we summarize all the information of the statement in the function by integrating each concatenated hidden state with its corresponding function-level attention scores as follows:

$$\tilde{c}^n = \sum_{i=1}^{\ell} \hat{\alpha}_i \hat{h}_i^n, \tag{6}$$

where $\tilde{c}^n \in \mathbb{R}^d$ represents the function-level source code features.

2) SIMILARITY CALCULATION

Based on the feature encoder, CODE-SMASH constructs twin sub-networks, each a feature encoder with identical architecture and shared weights, to analyze the similarity between features from two distinct source codes. Unlike typical Siamese-based methods that directly compare extracted features using a distance metric [38], CODE-SMASH compares through additional fully connected layers with an activation function. For this comparative analysis, CODE-SMASH aggregates all the features extracted by the statement-level and function-level encoders of each sub-network. After aggregation, it flattens the aggregated features, concatenates them, and feeds to fully connected layers. This approach allows CODE-SMASH to thoroughly compare the source codes, considering the statement-level and function-level features of the source codes. Furthermore, this architecture facilitates end-to-end learning, enabling the holistic training of the entire detection system tailored to a specific task. This capability ensures that CODE-SMASH can be directly applied to various coding challenges, optimizing its effectiveness and adaptability to different scenarios.

More formally, for a pair of source codes, given statement-level features $\tilde{s}_i^1, \tilde{s}_i^2$ for $i \in [1, \ell]$ and function-level features $\tilde{c}^1$ and $\tilde{c}^2$, we construct the final feature vector x by flattening and concatenating all these features, where $x \in \mathbb{R}^{2d(\ell+1)}$. The similarity between the two codes is computed using two fully connected layers with the ReLU activation function: $\sigma\left(f'\left(\text{ReLU}\left(f\left(x\right)\right)\right)\right)$, where f' and f represent two fully connected layers, and σ is the sigmoid function that produces scores within the range of $[0, 1]$, indicative of similarity levels. Finally, with a predefined threshold $\tau > 0$, we assess the presence of source code vulnerabilities. If $\sigma\left(f'\left(\text{ReLU}\left(f\left(x\right)\right)\right)\right) > \tau$, the code is considered vulnerable; otherwise, it is deemed benign.

IV. EXPERIMENTS

In this section, we compare our CODE-SMASH model with four baseline methods. Additionally, we conduct an experiment with a new configuration of CODE-SMASH that includes only the token-level and statement-level stages.

A. SETTINGS

1) METRICS

We assess the efficacy of the vulnerability detection models using established classification metrics, notably the receiver operating characteristic (ROC) curve and the corresponding area under the curve (AUC). The ROC curve illustrates the true positive rate (TPR) plotted against the false positive rate (FPR) across various classification threshold values. Furthermore, we evaluate the performance of our model based on metrics including accuracy (A), precision (P), recall (R), and the F1 score (F1). Let TP and FP represent the number of correctly and incorrectly predicted vulnerabilities, while TN and FN denote the number of correctly and incorrectly identified non-vulnerable code segments, respectively. These metrics are defined as follows:

$$\begin{aligned}
\text{TPR} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}}, \\
\text{A} &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}, \\
\text{P} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{R} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \\
\text{F1} &= \frac{2 \times \text{P} \times \text{R}}{\text{P} + \text{R}}.
\end{aligned}$$

2) IMPLEMENTATION DETAILS

Our model is implemented using Pytorch. The default configurations of CODE-SMASH are outlined as follows: We

TABLE 2. Characteristics of the datasets used for experiments.

Dataset	Chromium	Debian
No. of vulnerable functions	200	250
No. of non-vulnerable functions	150	200
No. of similar pairs	19900	31125
No. of dissimilar pairs	30000	50000

employ Word2vec [39] to convert the code into vectors, with an embedding size set to 64. Dropout regularization is applied at a rate of 0.5, and the batch size is set to 64. We conduct training for 100 epochs.

For the BiGRU network, the principal parameters are specified as follows: The batch size remains at 64, with training conducted over 100 epochs. Dropout regularization is applied with a keep rate of 0.2, and the network comprises 256 hidden units across 2 layers. We utilize stochastic gradient descent as the optimizer, with a learning rate set to 10^{-3}.

3) BASELINE METHODS

For all tasks, we assess the performance of CODE-SMASH by juxtaposing it against four deep learning models, employing the Siamese architecture to gauge similarity. The methods under comparison include:

TokenCNN [21]: This vulnerability detection tool relies on deep feature representation learning and directly interprets lexed source code through convolutional neural networks (CNNs) for feature extraction [49]. Emphasizing the lexical characteristics of code, TokenCNN aims to exploit the inherent structure of code snippets.

CNN+LSTM [22]: This method proposes a fusion of convolutional neural network-long short-term memory (CNN-LSTM) layers. By utilizing function calls as features to represent patterns, it endeavors to capture both local features and long-range dependencies within the code.

VDSimilar [38]: Employing a Siamese network architecture coupled with bidirectional long short-term memory (BiLSTM) and attention mechanisms, this model delves into learning similarities among vulnerability snippets. It places emphasis on comprehending the intricate relationships present within the code.

HAN [25]: This approach adopts a two-level attention network, focusing on both line and token levels within the code. Leveraging an attention mechanism, HAN determines the significance of vectors corresponding to code segments in indicating vulnerabilities.

Each of these models offers distinctive perspectives on vulnerability detection, harnessing a diverse array of deep learning techniques.

B. DATASET

Our research methodology is applied and evaluated using datasets sourced from pivotal domains, namely Chromium and Debian, which respectively represent web browsers and operating systems.

TABLE 3. Detection performance of vulnerabilities. The best values are boldfaced and the second-best are underlined.

	Model	Accuracy	Precision	Recall	F1
Chromium	TokenCNN [21]	0.8134	0.6678	<u>0.8178</u>	<u>0.7352</u>
	CNN+LSTM [22]	0.5946	0.4207	0.7415	0.5368
	VDSimilar [38]	<u>0.8295</u>	<u>0.7766</u>	0.6483	0.7067
	HAN [25]	0.7450	0.5693	0.8008	0.6655
	CODE-SMASH	**0.9490**	**0.9342**	**0.9025**	**0.9181**
Debian	TokenCNN [21]	<u>0.8942</u>	<u>0.8565</u>	<u>0.8086</u>	<u>0.8319</u>
	CNN+LSTM [22]	0.6581	0.4815	0.7269	0.5793
	VDSimilar [38]	0.8043	0.7201	0.6473	0.6818
	HAN [25]	0.7820	0.6357	0.7656	0.6946
	CODE-SMASH	**0.9143**	**0.8814**	**0.9398**	**0.8766**

The Chromium dataset comprises vulnerabilities reported by users and is derived from the Reveal dataset [50]. This dataset furnishes detailed information associating individual functions with Common Vulnerabilities and Exposures (CVEs), thereby facilitating function-level similarity detection. Likewise, the Debian dataset, also sourced from the Reveal dataset, encompasses vulnerabilities reported by users and provides correspondence details between functions and CVEs, thereby facilitating our analysis of function-level similarity.

Given the substantial size of the entire Reveal dataset, containing approximately 180, 000 functions, it is impractical to consider all possible pairs. Consequently, we adopted a sampling strategy wherein we selected a subset of 800 functions and generated approximately 130, 000 pairs in total for experimentation. This scale of data was determined to be adequate based on our experience for effectively training and evaluating prediction performance. Each pair is categorized as similar (labeled as 1) if they share the same CVE or different (labeled as 0) otherwise. The characteristics of these datasets are summarized in Table 2.

In adherence to the fixed input size requirements of deep learning models, our methodology employs 10 tokens for statement length and 80 tokens for function length per code snippet. Consequently, data padding or splitting is necessary to conform to the specified input dimensions. Additionally, we conduct random partitioning of the pairs dataset into training, validation, and test sets in each iteration, allocating proportions of 60%, 20%, and 20%, respectively.

C. EVALUATION

1) VULNERABILITY DETECTION PERFORMANCE

In this section, we evaluate the prediction performance of CODE-SMASH, a novel vulnerability detection model, against the second-best methods identified in our experiments. Our evaluation is based on key performance measures including accuracy, precision, recall, and F1 score, which are critical metrics for assessing the efficacy of vulnerability detection systems. The results are summarized in Table 3.

a: CHROMIUM DATASET

We identified VDSimilar as the second-best method in terms of accuracy and precision, while TokenCNN emerged

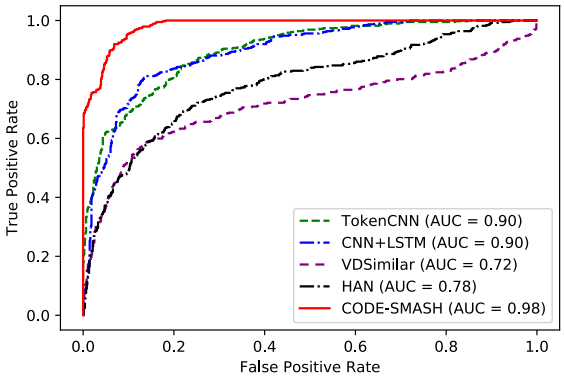

FIGURE 2. ROC curves and AUC values of vulnerability detection for the Chromium dataset.

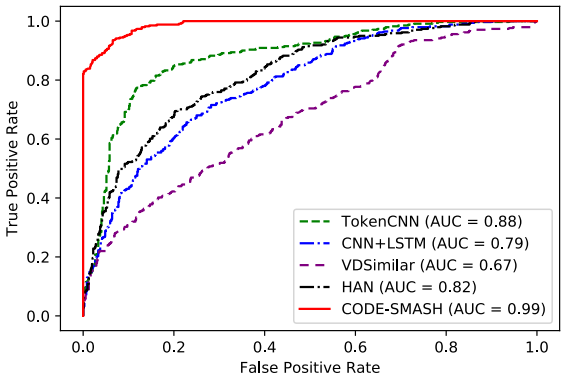

FIGURE 3. ROC curves and AUC values of vulnerability detection for the Debian dataset.

as the second-best for recall and F1 score. Our findings reveal that CODE-SMASH outperformed VDSimilar in accuracy and precision by approximately 14.4% and 20.3%, respectively. Additionally, CODE-SMASH demonstrated significant improvements over TokenCNN in terms of recall and F1 score, exhibiting approximately 39.3% and 30.0% enhancements, respectively. These results underscore the superiority of CODE-SMASH in accurately identifying vulnerabilities within the Chromium dataset.

b: DEBIAN DATASET

The results revealed TokenCNN as the second-best method across all performance measures. Our evaluation demonstrated that CODE-SMASH showcased notable enhancements over TokenCNN, with improvements of approximately 2.2%, 2.9%, 16.2%, and 5.4% in accuracy, precision, recall, and F1 score, respectively. This suggests that CODE-SMASH offers superior predictive capabilities compared to TokenCNN in detecting vulnerabilities within the Debian dataset.

Overall, our analysis indicates that CODE-SMASH significantly outperforms the second-best methods identified across both datasets, underscoring its efficacy in vulnerability detection. The substantial improvements observed in accuracy, precision, recall, and F1 score highlight the potential of CODE-SMASH as a robust solution for identifying

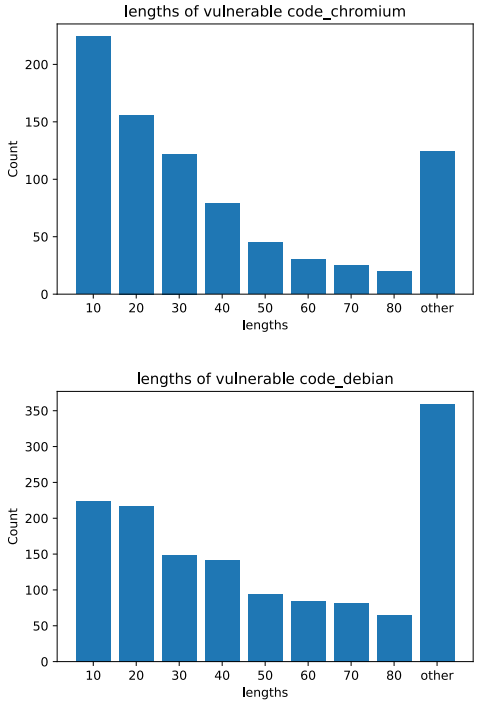

FIGURE 4. Length histograms of vulnerable functions in the Chromium and Debian datasets.

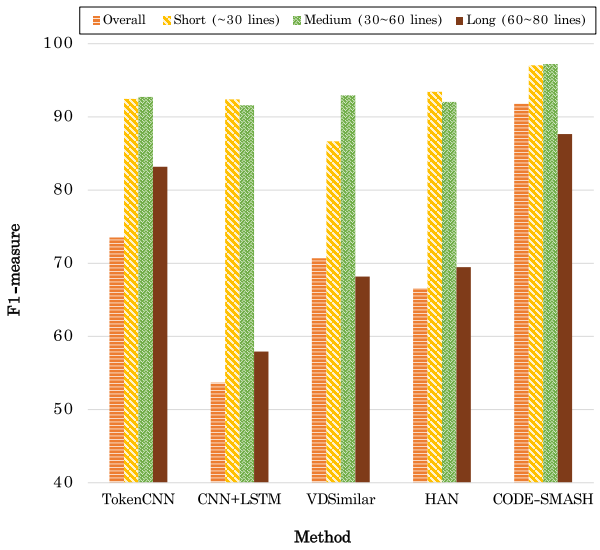

FIGURE 5. Vulnerability detection F1 scores for different code-length groups for the Chromium dataset.

vulnerabilities in software systems, considering that the Debian and Chromium datasets are comprised of a mix of code versions with varying lengths.

2) PERFORMANCE IN ROC AND AUC

Figures 2 and 3 present the Receiver Operating Characteristic (ROC) curves and their associated Area Under the Curve (AUC) values for the vulnerability detection methods, including our proposed model, CODE-SMASH. The ROC curve serves as a vital tool in evaluating the trade-offs

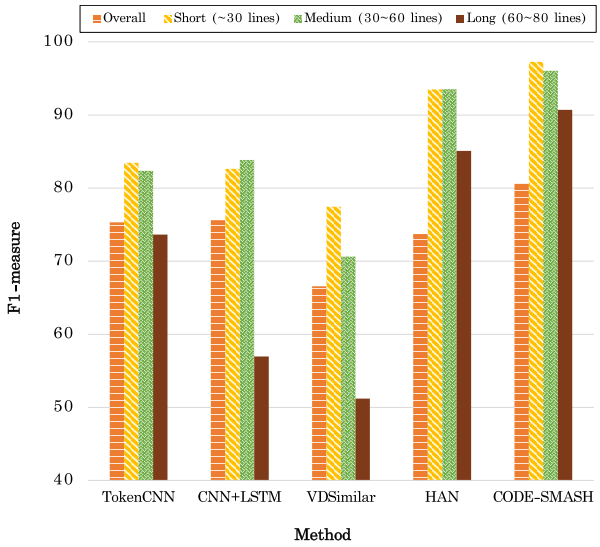

FIGURE 6. Vulnerability detection F1 scores for different code-length groups for the Debian dataset.

TABLE 4. Performance comparisons with CODE-SMASH versions denoted by letters in parentheses: T is for token-level information only and S is for statement-level information only (data: Debian, long code inputs with 60 to 80 lines).

Model	Accuracy	Precision	Recall	F1
TokenCNN [21]	0.9087	0.9088	0.7744	0.8363
CNN+LSTM [22]	0.6578	0.4870	0.6573	0.5415
VDSimilar [38]	0.6370	0.4523	0.7270	0.5571
HAN [25]	0.7442	0.5670	0.8023	0.6635
CODE-SMASH(T)	0.9428	0.8578	0.9802	0.9149
CODE-SMASH(S)	0.9161	0.8237	0.9062	0.8630
CODE-SMASH	0.9517	0.9247	0.9593	0.9417

between sensitivity (true positive rate) and specificity (1 - false positive rate). The AUC values summarize the ROC curves into scalars ranging from 0 to 1. Essentially, a larger AUC value signifies a model's enhanced ability to distinguish between vulnerable and non-vulnerable code. Notably, our CODE-SMASH model demonstrates the largest AUC among the compared methods, indicating its superior efficacy in vulnerability detection. Quantitatively, CODE-SMASH surpasses state-of-the-art models, improving detection performance by approximately 20% in terms of the AUC score.

A critical observation from CODE-SMASH's ROC curve is its exceptional true positive rate, maintained even when the false positive rate is close to zero. This highlights the model's capability to accurately identify vulnerabilities while minimizing false alarms.

3) PERFORMANCE OVER DIFFERENT CODE LENGTHS

One anticipated advantage of CODE-SMASH's multi-level architecture is its proficiency in extracting features from longer code inputs, which is a challenge for non-hierarchical architectures. This capability is particularly relevant given the diverse range of code lengths observed in the two real-world datasets utilized in our experiments, as depicted in Fig. 4.

TABLE 5. Comparison of inference times for each competing method, measured in seconds. The table shows the average runtime (denoted as mean) and its standard deviation (denoted as std) across 1,000 randomly selected samples.

	Model	Mean (std) time
Chromium	TokenCNN [21]	0.0019 (0.0052)
	CNN+LSTM [22]	0.0032 (0.0048)
	VDSimilar [38]	0.0375 (0.0060)
	HAN [25]	0.0237 (0.0057)
	CODE-SMASH	0.0159 (0.0051)
Debian	TokenCNN [21]	0.0020 (0.0058)
	CNN+LSTM [22]	0.0033 (0.0048)
	VDSimilar [38]	0.0385 (0.0064)
	HAN [25]	0.0241 (0.0062)
	CODE-SMASH	0.0154 (0.0059)

To comprehensively assess CODE-SMASH's effectiveness across varying code lengths, we categorized the functions into three distinct length groups: short (fewer than 30 lines), medium (30 to 60 lines), and long (60 to 80 lines). The performance of CODE-SMASH in vulnerability detection for these different length categories is depicted in Figs 5 and 6, corresponding to the Chromium and Debian datasets, respectively.

The results in Figs 5 and 6 indicate that CODE-SMASH consistently outperforms other competing methods across all code length categories. Notably, CODE-SMASH achieved significant improvements in F1 scores compared to the next-best methods for long-length code: it improved by 4.53 percentage points on the Chromium dataset and 5.62 percentage points on the Debian dataset, surpassing TokenCNN in Chromium and HAN in Debian. This improvement underscores CODE-SMASH's robustness and its capacity to effectively handle a wide range of code lengths, further reinforcing its utility in practical applications of vulnerability detection.

4) EFFECTIVENESS OF SIAMESE AND HIERARCHICAL STRUCTURE

To assess the impact of our architectural choices, we conducted an experiment comparing simplified variants of our CODE-SMASH model: CODE-SMASH(T) and CODE-SMASH(S) variants that use token and statement information only from the inputs, respectively.

In this experiment, we used the 'long' category (60 to 80 lines) from the Debian dataset for training and testing, and we reported the mean values of results obtained through 5-fold cross-validation. The results are presented in Table 4. Our findings indicate that the competing methods employing solely token-level information, such as TokenCNN, CNN+LSTM, and VDSimilar, demonstrated comparable yet inferior performances compared to CODE-SMASH variants. Specifically, TokenCNN, the best-performing among these baselines, achieved an F1 score of 0.8363, which is notably lower than any variant of CODE-SMASH. This difference underscores the limitations of models reliant on token-level information alone when handling more complex, longer code inputs.

The HAN model, which integrates statement-level knowledge, shows improved performance over some of the token-level methods, with an F1 score of 0.6635. However, it still falls short when compared to the CODE-SMASH variants. This outcome suggests that while incorporating statement-level information is beneficial, it is not sufficient on its own to handle the intricacies of longer code sequences effectively.

The CODE-SMASH(T) variant, focusing solely on token-level information, achieves a significant leap in performance with an F1 score of 0.9149. Similarly, CODE-SMASH(S), which utilizes only statement-level information, also shows strong results with an F1 score of 0.8630. These results indicate that even when focusing on a single hierarchical level, the Siamese architecture inherent in CODE-SMASH provides a substantial advantage in processing and understanding code structures.

The complete CODE-SMASH model, which combines both token and statement levels, further enhances performance, reaching an F1 score of 0.9417. This demonstrates that the integration of multiple hierarchical levels, combined with the Siamese architecture, significantly improves the model's ability to understand and analyze complex, long code sequences.

5) COMPUTATION TIME

Detecting vulnerabilities within a reasonable time is important for deep learning-based vulnerability detectors to serve as practical tools for real-world applications. Table 5 shows the average inference time and its standard deviation for each competing method across 1, 000 samples from each dataset.

According to the table, CODE-SMASH has an average inference time of less than 0.0160 seconds. Although TokenCNN and CNN+LSTM models have faster inference times than CODE-SMASH, their detection performance is lower than CODE-SMASH as demonstrated in Tables 3 and 4. Notably, compared to VDSimilar, which is the second-best detector in terms of detection accuracy on the Chromium dataset as shown in Table 3, CODE-SMASH has a much faster inference time, averaging $\times 2.43$ faster. These results indicate that CODE-SMASH detects vulnerabilities effectively and does so within a reasonable time.

V. CONCLUSION

In this paper, we presented CODE-SMASH, a novel deep learning-based source code vulnerability detector that uses a Siamese neural network with hierarchical structures comprising BiGRU and a self-attention mechanism, extracting information reflecting the hierarchical structure of source codes and analyzing their similarity to detect vulnerabilities. Alongside CODE-SMASH, we introduced our source code preprocessing steps for effective source code analysis, including comment removal, tokenization, and vectorization. Through our experiments, we showed that CODE-SMASH outperforms existing methods in detection performance and is more effective in handling lengthy code sequences, emphasizing its efficiency and potential to be a key component in software security applications.

However, CODE-SMASH has several remaining challenges, such as potential dependency on the nature of training datasets and the model's high computational demands. Although it has shown reasonable computation time, there is still room for improvement to be more effectively used in real-world applications. To address these challenges, one approach is to increase the diversity of training datasets by incorporating a broader range of source code from various languages, platforms, and domains. Exploring data augmentation techniques for source code can also help diversify the training data by generating synthetic variations of source code. In addition, optimizing computational efficiency could be achieved through model compression methods, such as pruning or quantization, to reduce computational overhead while maintaining performance. We suggest exploring these approaches in future research to enhance the effectiveness of CODE-SMASH while mitigating the identified challenges.

Despite these identified challenges, given the growing importance of automated vulnerability detection in modern software development as software-based systems become more prevalent and the risk from source code vulnerabilities increases, our research offers a significant contribution with the potential for further development to improve software security. We believe that with the use and further development of CODE-SMASH, it can serve as an effective tool for automated vulnerability detection, helping to address the escalating challenges in software security.

ACKNOWLEDGMENT

(Sungmin Han and Hyunkyung Nam contributed equally to this work.)

REFERENCES

[1] *The OWASP IoT Top 10 Vulnerabilities and How to Mitigate Them*. Accessed: May 17, 2023. [Online]. Available: https://www.sisainfosec.com/blogs/the-owasp-iot-top-10-vulnerabilities-and-how-to-mitigate-them/

[2] *OWASP Internet of Things*. Accessed: May 17, 2023. [Online]. Available: https://owasp.org/www-project-internet-of-things/

[3] I. Kabanov and S. Madnick, "Applying the lessons from the equifax cybersecurity incident to build a better defense," *MIS Quart. Executive*, vol. 20, pp. 109–125, Aug. 2021.

[4] *Common Vulnerabilities and Exposures*. Accessed: Jul. 7, 2023. [Online]. Available: https://www.cve.org/

[5] *Zero-Day Attack*. Accessed: Aug. 4, 2023. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/zero-day-attack/

[6] S. Abdelnabi, K. Krombholz, and M. Fritz, "VisualPhishNet: Zero-day phishing website detection by visual similarity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1681–1698.

[7] N. Ben-Asher and C. Gonzalez, "Training for the unknown: The role of feedback and similarity in detecting zero-day attacks," *Proc. Manuf.*, vol. 3, pp. 1088–1095, Jul. 2015.

[8] I. V. Krsul, *Software Vulnerability Analysis*. West Lafayette, IN, USA: Purdue University, 1998.

[9] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *J. Syst. Softw.*, vol. 158, Dec. 2019, Art. no. 110427.

[10] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surveys*, vol. 50, no. 4, pp. 1–36, Jul. 2018.

[11] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software vulnerability analysis and discovery using deep learning techniques: A survey," *IEEE Access*, vol. 8, pp. 197158–197172, 2020.

[12] 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.*, Aug. 2005, pp. 539–546.

[13] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," in *Proc. ICML Deep Learning Workshop*, 2015, pp. 1–28.

[14] 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.*, 2019, pp. 795–806.

[15] O. Ferschke, I. Gurevych, and M. Rittberger, "Flawfinder: A modular system for predicting quality flaws in wikipedia," in *Proc. CLEF Online Work. Notes/Labs/Workshop*, 2012, pp. 1–10.

[16] *Rough Auditing Tool for Security (RATS)*. Accessed: Nov. 9, 2023. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[17] (2023). *Shift Everywhere With the Leading Cloud-Native Appsec Platform*. [Online]. Available: https://www.checkmarx.com/

[18] (2020). *Fortify Software Security Center*. [Online]. Available: https://www.microfocus.com/en-us/solutions/application-security

[19] (2020). *Sonarqube*. [Online]. Available: https://www.sonarqube.org

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

[21] 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.*, 2018, pp. 757–762.

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

[23] 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. 1–28.

[24] 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.

[25] 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*, 2022, pp. 1–6.

[26] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–28.

[27] (2023). *SARD*. [Online]. Available: https://samate.nist.gov/SARD/

[28] (2023). *SySeVR*. [Online]. Available: https://github.com/SySeVR/SySeVR

[29] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*.

[30] 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.

[31] 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)*, May 2007, pp. 106–115.

[32] 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.

[33] 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.

[34] 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.

[35] S. Yang, Z. Xu, Y. Xiao, Z. Lang, W. Tang, Y. Liu, Z. Shi, H. Li, and L. Sun, "Towards practical binary code similarity detection: Vulnerability verification via patch semantic analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, pp. 1–29, Nov. 2023.

[36] 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.*, 2016, pp. 201–213.

[37] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 3835–3845.

[38] 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.

[39] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.

[40] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–29, Jan. 2019.

[41] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. 1, pp. 1–25, Oct. 2018.

[42] P. Neculoiu, M. Versteegh, and M. Rotaru, "Learning text similarity with Siamese recurrent networks," in *Proc. 1st Workshop Represent. Learn. NLP*, 2016, pp. 148–157.

[43] W. Chen, R. Guo, G. Wang, L. Zhang, J. Qiu, S. Su, Y. Liu, G. Xu, and H. Chen, "Smart contract vulnerability detection model based on Siamese network," in *Proc. Int. Conf. Smart Comput. Commun.*, 2022, pp. 639–648.

[44] H. Hindy, C. Tachtatzis, R. Atkinson, E. Bayne, and X. Bellekens, "Developing a Siamese network for intrusion detection systems," in *Proc. 1st Workshop Mach. Learn. Syst.*, Apr. 2021, pp. 120–126.

[45] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or underscore," in *Proc. IEEE 17th Int. Conf. Program Comprehension*, May 2009, pp. 158–167.

[46] W. Uddin Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," 2020, *arXiv:2005.00653*.

[47] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2021, pp. 292–303.

[48] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013, pp. 1–28.

[49] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2921–2929.

[50] 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.

SUNGMIN HAN received the B.S. degree in computer engineering from Kwangwoon University, Seoul, South Korea, in 2021. He is currently pursuing the Ph.D. degree with the School of Cybersecurity, Korea University. His main research interests include trustworthy AI, including eXplainable AI, model stealing attack and defense, adversarial robustness, and backdoor robustness.

HYUNKYUNG NAM received the B.S. degree in information and communication engineering from Inha University, South Korea, in 2018. She is currently pursuing the M.S. degree with the School of Cybersecurity, Korea University, South Korea. She is doing research on vulnerability detection and cybersecurity risk management.

JAESIK KANG received the B.S. and master's degrees 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 for 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 for cyber security. His research interests include network security and cyber warfare, especially cyber training systems.

SEUNGJAE CHO received the M.S. degree in modeling and simulation engineering from Hannam University, Republic of Korea, in 2011. Since May 2001, he has been affiliated with LIG Nex1, a leading defense industry in South Korea, as a Researcher for cyber security. His research interests include network security and system engineering.

SANGKYUN LEE 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, TU Dortmund University, Germany, and from 2015 to 2016, he was the Project Leader of 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-si, South Korea. From 2020 to 2021, he was an Assistant Professor with the School of Cybersecurity, Korea University, Seoul, and since 2022, he has been affiliated as an Associate Professor with Korea University. His main research interests include trustworthy AI, model compression, and AI for security.