



CodeGrapher: An Image Representation Method to Enhance Software Vulnerability Prediction

Ramin Fuladi¹ ^a and Khadija Hanifi² ^b

¹*Ericsson Research, Istanbul, Turkey*

²*Sabanci University, Istanbul, Turkey*


Keywords: Software Vulnerability Prediction, CodeGrapher, ML Algorithms, Semantic Relations, Source Code Analysis, Similarity Distance Metrics, Image Generation.


Abstract: Contemporary software systems face a severe threat from vulnerabilities, prompting exploration of innovative solutions. Machine Learning (ML) algorithms have emerged as promising tools for predicting software vulnerabilities. However, the diverse sizes of source codes pose a significant obstacle, resulting in varied numerical vector sizes. This diversity disrupts the uniformity needed for ML models, causing information loss, increased false positives, and false negatives, diminishing vulnerability analysis accuracy. In response, we propose CodeGrapher, preserving semantic relations within source code during vulnerability prediction. Our approach involves converting numerical vector representations into image sets for ML input, incorporating similarity distance metrics to maintain vital code relationships. Using Abstract Syntax Tree (AST) representation and skip-gram embedding for numerical vector conversion, CodeGrapher demonstrates potential to significantly enhance prediction accuracy. Leveraging image scalability and resizability addresses challenges from varying numerical vector sizes in ML-based vulnerability prediction. By converting input vectors to images with a set size, CodeGrapher preserves semantic relations, promising improved software security and resilient systems.

1 INTRODUCTION

Software vulnerabilities are inherent weaknesses in code that arise during the development or committing process and can be exploited by malicious actors to gain unauthorized access to a system and execute harmful actions (Hanifi et al., 2023). To mitigate potential risks, organizations employ various methods to identify and predict these vulnerabilities. Two widely adopted approaches are static and dynamic analysis (Palit et al., 2021). Static analysis involves scrutinizing the source code or binary code without actually executing it. This method seeks to identify existing vulnerabilities by examining the code's structure, syntax, and potential logical flaws. It is particularly useful during the early stages of development when the codebase is accessible and can be analyzed thoroughly. Static analysis tools can quickly scan large codebases, providing valuable insights into potential vulnerabilities without the need for runtime execution (Schiewe et al., 2022). On the other hand, dynamic

analysis involves running the code and analyzing its behavior during runtime. By observing the code in action, this method can detect vulnerabilities that may not be apparent in the static analysis phase. Dynamic analysis is advantageous when dealing with complex and interactive applications, as it allows for a better understanding of how the software responds to real-world inputs and interactions (Lin et al., 2020). The advantage of static analysis lies in its ability to detect vulnerabilities before the code is executed, which can potentially save significant time and resources in the development process. It can catch issues early on, reducing the chances of encountering critical vulnerabilities later in the software lifecycle. Moreover, static analysis can be automated, making it scalable and efficient for large-scale projects (Halepmollası et al., 2023). Despite its benefits, static analysis may produce false positives or false negatives, meaning that it can flag non-existing vulnerabilities or miss actual flaws due to the inherent complexities of code analysis. Additionally, static analysis tools may not capture vulnerabilities arising from runtime-specific conditions or interactions. Recently, Machine Learning (ML) and Artificial Intelligence (AI) techniques

^a  <https://orcid.org/0000-0003-4142-1293>

^b  <https://orcid.org/0000-0001-7044-3315>

have been integrated into static analysis tools to enhance vulnerability detection. By leveraging ML/AI, these tools can adapt and improve their accuracy over time. However, applying ML/AI to static code analysis presents its own set of challenges. The structural differences between code and traditional text data can hinder the effectiveness of standard ML algorithms, requiring specialized approaches and tailored models for optimal results (Bilgin et al., 2020). In order to leverage ML algorithms for predicting software vulnerabilities, it is essential to represent the source code as numerical vectors. This process, commonly referred to as code embedding, aims to transform the code into meaningful numerical representations. This can be achieved either through manual extraction of features such as lines of code or code complexity, or by employing ML-based techniques to automatically learn the vector representation of the code.

Several methods have been proposed for code embedding (Hanifi et al., 2023; Bilgin et al., 2020; Şahin et al., 2022; Duan et al., 2019; Halepmollası et al., 2023), but they all encounter a common challenge: the variability in the length of the source code, which results in varying sizes of the output vectors. Moreover, in source code, multiple lines can be interconnected, meaning that one line of code may be dependent on or related to another line within the code. This interrelation and semantic code relationship are also relevant when dealing with vulnerabilities. However, ML models often require fixed input sizes, necessitating the vector representation of source codes to be adjusted to match the model's input size. Consequently, some vectors need to be truncated, while others must be padded with zeros to achieve uniformity in input size (Telang and Wattal, 2007). The process of modifying vector sizes through truncation or padding leads to information loss, adversely impacting the accuracy of vulnerability analysis. As a result, the occurrence of false negatives and false positives increases, undermining the overall effectiveness of the analysis. Addressing this challenge is crucial for enhancing the accuracy and reliability of software vulnerability prediction using ML algorithms.

In this study, we introduce a novel approach called CodeGrapher, which aims to convert the numerical vector representation of source code into one or more images. Due to the different size of the source codes, the size of the resulted images vary. By re-sizing the size of the images to a global and constant value (i.e. $n \times n$), they can be utilized by ML methods to predict vulnerabilities in the source code. CodeGrapher addresses the challenge of varying numerical vector sizes resulting from differences in the sizes of source codes. By resolving this discrepancy in input sizes

for ML models during static analysis, our solution ensures that all relevant information within the source code is retained.

The conversion of numerical vector representations into images is advantageous for resizing due to the inherent adaptability of image variables. Images can be easily resized to a standardized $n \times n$ format without compromising the integrity of the information they encapsulate. During the resizing process, CodeGrapher ensures that relevant details within the images are preserved, maintaining the essential characteristics of the source code representations. This adaptability and preservation of information contribute to the effectiveness of CodeGrapher in addressing the challenges posed by varying source code sizes in the context of static analysis for machine learning models.

To achieve this, we employ similarity distance metrics on the numerical representation of the source code, which allows us to generate an image based on the vector input. Subsequently, the size of the image is adjusted to a fixed value to align with the requirements of ML models. In converting the source code to the numerical representation, we utilize the Abstract Syntax Tree (AST) representation along with the skip-gram embedding algorithm, which is an ML-based algorithm commonly used in Natural Language Processing (NLP). This research makes significant contributions to the field of software vulnerability prediction and software security. The main contributions of this paper can be listed as follows:

- **Innovative Numerical-to-Image Transformation:** Our innovative solution converts numerical vectors of source code into images, enhancing ML model inputs. This unique transformation preserves semantic relationships within code elements, bridging code analysis and image processing for intuitive vulnerability prediction.
- **Semantic Relationship Preservation:** By integrating similarity distance metrics into image generation, we improve the accuracy of vulnerability prediction by preserving code element relations, minimizing false results in software analysis.
- **Uniform ML Model Input:** Our solution successfully preserves all source code information while addressing varying numerical vector sizes, ensuring uniformity in ML model input. This enhances accuracy in vulnerability prediction, contributing to the development of resilient software systems.

The remainder of this paper is structured as follows: In Section 2, we provide a comprehensive review of relevant studies. Section 3 delves into a de-

tailed explanation of our proposed approach, while Section 4 presents the implementation of CodeGrapher. Section 5 provides the threats to validity of our findings. Finally, in Section 6, we draw our study to a conclusion.

2 RELATED WORK

The use of numerical vector representation is crucial for harnessing ML algorithms in predicting software vulnerabilities. This technique, referred to as code embedding (Kanade et al., 2020), revolves around converting the source code into numerical vectors. This transformation can be accomplished either manually, where features such as line of code and code complexity are extracted, or automatically through ML-based techniques that learn vector representations (Alon et al., 2019).

Several techniques have been proposed for code embedding. Alon et al. (Alon et al., 2019) introduced code2vec, a neural network-based model that represents source code as a continuous distributed vector. They break down the Abstract Syntax Tree (AST) of the code into paths and learn the atomic representation of each path, aggregating them as a set. Lozoya et al. (Lozoya et al., 2021) built upon code2vec and developed comit2vec, which focuses on embedding representations of code changes. They utilized the obtained representations for vulnerability fixing commit prediction. Furthermore, word embedding techniques, like word2vec (Alon et al., 2019), have been used to convert source code into numerical vectors. Hare et al. (Harer et al., 2018) applied word2vec on C/C++ tokens to generate word embedding representations for ML-based software vulnerability prediction. Henkel et al. (Henkel et al., 2018) utilized the GloVe model to extract word embedding representations from the AST of C source code. Fang et al. (Fang et al., 2020) introduced FastEmbed, employing the FastText technique (Feutrill et al., 2018), and achieved an F1-score of 0.586. Sahin et al. (Şahin et al., 2022) proposed a vulnerability prediction model using different source code representations. They explored whether a function at a specific code change is vulnerability-inducing or not by representing function versions as node embeddings learned from their AST. They built models using Graph Neural Networks (GNNs) with node embeddings, Convolutional Neural Networks (CNNs), and Support Vector Machines (SVMs) with token representations. Their experimental analysis on the WireShark project showed that the GraphSAGE model achieved the highest AUC rate with 96%, while the

CNN model achieved the highest recall and precision rates with 77% and 82%, respectively. In study by Bilgin et al. (Bilgin et al., 2020), the authors introduced a technique for software vulnerability prediction at the function level in C code. They achieve this by converting the Abstract Syntax Tree (AST) of the source code into a numerical vector. Subsequently, they utilize a 1D Convolutional Neural Network (CNN) for software vulnerability prediction. Similarly, in another study by Duan et al. (Duan et al., 2019), the authors utilized the Control Flow Graph (CFG) and AST as graph representations to predict vulnerabilities. By employing soft attention, they extract high-level features crucial for vulnerability prediction from the graphs. Moreover, Zhou et al. (Zhou et al., 2019) proposed a function-level software vulnerability prediction method based on a graph representation. Their approach incorporates not only the AST but also dependency and natural code sequence information to enhance the prediction process.

Previous studies utilize embedding techniques for software vulnerability prediction, facing challenges with ML algorithms requiring uniform vector lengths, often resulting in truncation or zero-padding. These studies often overlook semantic relations between code components. In our research, we introduce a novel approach converting code vectors into images, preserving semantic relations and resolving variable-sized outputs. Utilizing images captures intricate code relationships, maintaining fixed output size, and improving vulnerability prediction accuracy and effectiveness.

3 PROPOSED APPROACH

Figure 1 serves as a visual representation of the solution devised to standardize the input dimensions for ML-based static analysis tools. The primary goal is to ensure uniformity in the input data size. The process commences by acquiring a numerical vector representation of the source code. This vector, along with one or more similarity distance functions like cosine similarity or dot product, is then input into the CodeGrapher module. The application of these distance functions serves a dual purpose: first, to identify similarities among elements within the vector, and second, to retain the underlying semantic information associated with the source code. The pivotal role of the CodeGrapher module is to convert the numerical vector into one or more images, either in RGB or gray-scale format, adhering to predetermined dimensions. These resultant images constitute the module's outputs. This transformation is facilitated by the utilization of the

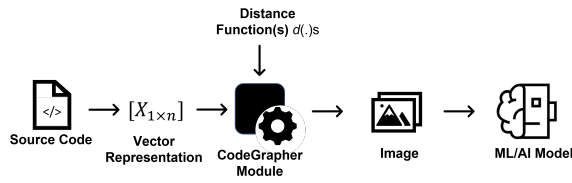


Figure 1: An illustrative scheme outlining the key steps for CodeGrapher implementation.

aforementioned distance function(s). To elaborate on the image generation process, Figure 2 provides a visual breakdown. The process involves selecting a window of size w and systematically sliding this window across the numerical vector. With each window position, corresponding matrices are generated. The diagram showcases the resulting matrices for two instances: $w = 1$ and $w = k$. These matrices are subsequently transformed by mapping their values to the range of $[0, 255]$. Furthermore, their format is altered to *uint8*, effectively transitioning the matrices into image format. Upon successful image generation, the images' dimensions are adjusted to match the required input dimensions of the machine learning model. The generated images are then harnessed by

$$X = [x_1, x_2, \dots, x_n]$$

$$\text{if } w = 1: \begin{bmatrix} d(x_1, x_1) & d(x_1, x_2) & \dots & d(x_1, x_n) \\ d(x_2, x_1) & d(x_2, x_2) & \dots & d(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ d(x_n, x_1) & d(x_n, x_2) & \dots & d(x_n, x_n) \end{bmatrix}$$

$$\vdots$$

$$\text{if } w = k: \begin{bmatrix} d([x_1, \dots, x_k], [x_1, \dots, x_k]) & d([x_1, \dots, x_k], [x_2, \dots, x_{k+1}]) & \dots & d([x_1, \dots, x_k], [x_{n-k+1}, \dots, x_n]) \\ d([x_2, \dots, x_{k+1}], [x_1, \dots, x_k]) & d([x_2, \dots, x_{k+1}], [x_2, \dots, x_{k+1}]) & \dots & d([x_2, \dots, x_{k+1}], [x_{n-k+1}, \dots, x_n]) \\ \vdots & \vdots & \ddots & \vdots \\ d([x_{n-k+1}, \dots, x_n], [x_1, \dots, x_k]) & d([x_{n-k+1}, \dots, x_n], [x_2, \dots, x_{k+1}]) & \dots & d([x_{n-k+1}, \dots, x_n], [x_{n-k+1}, \dots, x_n]) \end{bmatrix}$$

Figure 2: Visual Breakdown of Image Generation Process for Different Window Sizes ($w=1$ and $w=k$).

the ML model to identify potential software vulnerabilities present in the source code. The process is depicted in Figure 2. Importantly, the number of outputs from the CodeGrapher module is adaptable based on the configuration settings. For instance, if there exists only one distance function and one window, the output will be a single image. Conversely, if \mathcal{D} distinct distance functions and \mathcal{L} separate windows are employed, the number of generated images will amount to $\mathcal{D} \times \mathcal{L}$. This level of flexibility allows for the production of varying numbers of images, depending on the specific demands of the analysis procedure. This accommodates diverse analysis requirements and provides versatility in tailoring the solution to the specific needs of software vulnerability prediction.

4 CodeGrapher IMPLEMENTATION

For a comprehensive illustration of the process, we conducted an experiment focused on predicting vulnerabilities within C-source code at the function level. The schematic representation of the system employed in this experiment is visualized in Figure 3. This system leverages a Convolutional Neural Network (CNN) and adopts an image-based representation of the source code as input to successfully predict existing vulnerabilities in C-source code.

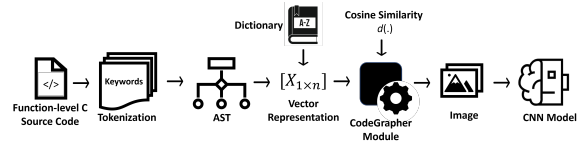


Figure 3: The main steps involved in implementing CodeGrapher to construct a system for predicting source code vulnerabilities.

To facilitate a step-by-step understanding of each stage, an illustrative code segment is presented below. This code snippet provides a simple way to check if a given integer is even or odd by returning 1 for even numbers and 0 for odd numbers.

```
int main() { // this the main function
  int a1 = 5; // define an integer variable
  printf(a1); // print the variable to the console
}
```

Now, we will proceed to demonstrate each step using the provided code sample:

4.1 Tokenization and AST Representation

The initial step involves the application of lexical analysis, whereby the source code is transformed into a sequence of tokens. This transformation is evident when examining the provided example source code, which is altered as follows:

```
int (keyword), isEven (identifier), LPAREN (delimiter),
int (keyword), num (identifier), RPAREN (delimiter),
= (operator), 5 (constant), ; (symbol)
```

Upon obtaining the token sequence from the source code, the parsing process is initiated. Within this parsing phase, the tokens generated during lexical analysis undergo a conversion process, typically resulting in the creation of a data structure such as a parse tree. This hierarchical structure offers a visual representation of the input code's structure while simultaneously validating its syntactical correctness based on the rules defined by a context-free grammar (CFG). The output of this step often manifests

as an Abstract Syntax Tree (AST) representation of the original source code. The AST, which stems from the tokens extracted during lexical analysis, encompasses both the structural layout and semantic insights pertaining to the code. This convergence of structural and semantic information within the AST has led to the development of a trend in source code analysis known as AST-based intelligent analysis (Chen et al., 2019). Figure 4 illustrates the AST generated to hierarchically represent the syntactic structure of the *isEven* function.

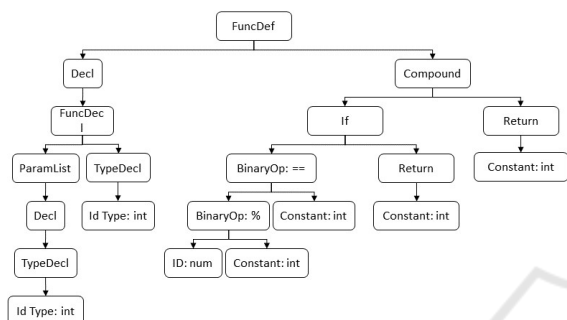


Figure 4: AST of the *isEven* Function.

4.2 Utilizing Embedding for Code Representation

Developing effective representations for source code is a complex endeavor, largely due to the intricate nature of programming languages, the incorporation of libraries, and the diverse coding styles employed by various developers. Leveraging established techniques from the realm of Natural Language Processing (NLP), we endeavor to extract embedding features from source code. However, it's important to note that the structure of source code differs significantly from that of regular textual content.

In light of these challenges, we opt for an approach that involves utilizing AST as the foundation of our code representation technique. The conversion of source code into its abstract structure allows us to capture the inherent syntax and structure of the code. To bridge the gap between AST and textual representation while maintaining information about each node's location in the AST, we employ the Breadth-First Search (BFS) technique. Subsequently, we employ Skip-Gram, a word embedding technique, to translate AST nodes into numerical vectors. The comprehensive steps we follow to extract embedding features are elaborated below:

1. **Normalizing Identifier Names:** While AST predominantly captures structural and content-related aspects, it omits certain details. For instance, grouping parentheses are implicit within

the tree structure and are not represented as distinct nodes in the AST. However, structural nodes such as function names are often irrelevant to our purpose, lacking vulnerability-related information. To mitigate this, prior to using AST nodes, we initiate a normalization process. This process replaces non-essential nodes with uniquely predefined names. For example, both variable and function names, being unimportant, are substituted with distinctive identifiers like **VARIABLE_NAME** and **FUNCTION_NAME**. For instance, in the examined example, *isEven* is replaced with *FUNCTION_NAME*, and *num* is replaced with *VARIABLE_NAME*.

2. **Conversion to Word Vectors:** To ensure the normalized AST is transformed into a one-dimensional array without sacrificing the relationships among AST nodes, we adopt the BFS technique. However, leaf nodes remain attached to their parent nodes, like the node `BinaryOp` and the operator `%`, as they serve as features rather than separate entities. The resulting array is then fed into the embedding model to derive the feature matrix. An equivalent word vector corresponding to the normalized AST of the example code is presented below:

```
[FuncDef, Decl, Compound, FuncDecl, If, Return, ParamList, TypeDecl, BinaryOp: ==, Return, Constant: int, Decl, Id Type: int, BinaryOp: %, Constant: int, Constant: int, TypeDecl, ID: num, Constant: int, Id Type: int]
```

3. **Conversion to Numerical Vectors (Skip-Gram):** The Skip-Gram method is employed to transition the aforementioned word vector into a numerical vector. Skip-Gram operates by extracting numerical features based on the relationships among neighboring nodes. This approach preserves contextual information, which is then mapped into the resulting numerical vector (Bamler and Mandt, 2017). The process involves two distinct steps:

- **Step 1:** we generate a feature matrix or a lookup dictionary using the SkipGram method, trained with the VDISC dataset (Russell et al., 2018). By the end of this pre-processing stage, we obtain an embedding feature matrix, referred to as the 'Dictionary'. This matrix represents each word as a numerical vector within $\mathcal{R}^{20 \times 1}$, considering its location within the code. As an example, the associative numerical representation of 'FuncDef' node is as below:

```
FuncDef: [-2.5182416, 3.1283994, 2.2289238, 0.7242722, 1.4296024, 1.5872365, 2.0136333, 0.49053535, -0.82888806, -3.0382762, 2.8487883, 1.5573912, -0.26117662, 1.3050934, -1.3061347, -0.31573528, -3.5838423, 1.6379417, 4.9378858, 0.04924774]
```

- **Step 2:** we use the acquired 'Dictionary' to transform the word vector into a numerical one. This involves substituting each node with its corresponding numerical representation from the Dictionary. Consequently, each function is depicted by a single numerical vector, with a length of $20 \times nodes\#$. This length varies based on the function's size and the number of nodes in the AST.

4.3 Code to Image Representation

After transforming the source code into vector form, the vector is fed into the CodeGrapher module to generate an image-based representation. For optimal utilization of the CodeGrapher module, an additional input in the form of a distance function is required. In this specific study, we've selected cosine similarity as our distance function. Cosine similarity is a mathematical metric utilized to evaluate the likeness between two vectors by computing the cosine of the angle formed between them. This metric finds applications across diverse domains such as natural language processing, recommendation systems, and image processing. One of its notable features is its disregard for vector magnitude, instead concentrating solely on directional alignment within a multidimensional space. This characteristic proves valuable when comparing documents, images, or other data where the magnitude may not offer as much meaningful information as the relative orientation of vectors. It's important to note that in this study, we empirically set the value of W to 10. As an illustrative example, Figure 5 provides visual representations of two distinct source codes. One of these codes contains vulnerabilities related to buffer overflow (CWE120), while the other code is devoid of such vulnerabilities. CWE (Common Weakness Enumeration) is a comprehensive catalog of software and hardware vulnerabilities, and CWE120 is a specific category pinpointing the weakness associated with buffer overflow vulnerabilities (Sane, 2020). Buffer overflow, as described in CWE120, is a critical software vulnerability where a program writes more data into a buffer (temporary data storage) than it can accommodate (Sane, 2020). This overflow can overwrite adjacent memory, potentially causing crashes, unauthorized access, or even remote code execution.

4.4 Dataset

In this study, we employed the publicly available Draper VDISC Dataset, as described in (Russell et al., 2018), to conduct our experiments. This dataset

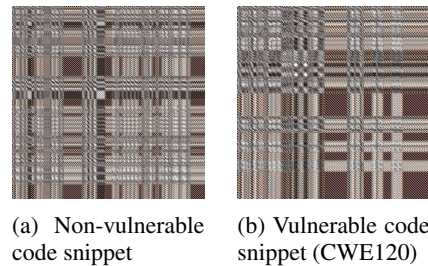


Figure 5: Images representing two distinct source codes: (a) non-vulnerable source code, (b) vulnerable code containing CWE120.

comprises an extensive collection of function-level source code samples gathered from various sources, including open-source projects such as the Debian Linux distribution (unk, 2023a), public git repositories on GitHub (unk, 2023b), and the synthetic SATE IV Juliet Test Suite (Black and Black, 2018) from NIST's Samate project. While the SATE IV Juliet Test Suite contains synthetic code, it constitutes only a small portion (approximately 1%) of the entire dataset. The authors of the dataset meticulously labeled the function-level codes based on findings from three different static analyzers, categorizing them into five distinct groups of Common Weakness Enumeration (CWE) vulnerabilities (Russell et al., 2018).

This categorization involved labeling functions flagged by static code analyzers as "vulnerable" for the respective CWE category, while labeling others as "non-vulnerable" functions. The dataset was thoughtfully partitioned into training (80%), validation (10%), and test (10%) sets to ensure the independence of these subsets, with strict avoidance of duplicate samples between training and test sets. Furthermore, we constructed balanced subsets in select experiments to gauge the detectability of different vulnerability categories on an equal footing. It's worth mentioning that the referenced dataset primarily contains functions written in C and C++ languages. For our specific implementation, we focused on functions written in the C language and ensured their parseability using the Pycparser (Bendersky, 2019) parser, resulting in several subsets derived from the original training, validation, and test datasets.

4.5 Vulnerability Prediction Results Based on CNN Model

After the transformation of source code into image representations, these visual depictions become the input data for our machine learning models. In this study, we employ a Convolutional Neural Network (CNN) as our chosen model for predicting vulner-

Table 1: Vulnerability prediction results.

CWE	Method in (Bilgin et al., 2020)			Our proposed solution		
	Precision	Recall	F1	Precision	Recall	F1
CWE119	0.504	0.515	0.509	0.813	0.841	0.824
CWE120	0.415	0.440	0.427	0.764	0.782	0.773
CWE469	0.060	0.187	0.090	0.460	0.552	0.502
CWE476	0.701	0.521	0.598	0.932	0.910	0.921
Other	0.218	0.353	0.270	0.624	0.652	0.640

abilities at the function-level within C source code. We select parameters such as the number of layers, the quantity of filters, and the dimensions of the filters empirically. Each filter within the neural network is defined as a 3×3 kernel. Collectively, the CNN classifier encompasses an impressive 33,573,505 parameters, all of which are learned during the training phase. For activation functions, all layers, except the final one which employs the Softmax activation function, utilize Rectified Linear Unit (ReLU) functions.

To evaluate our proposed method and compare its results with previous studies, we train the CNN model separately for the mentioned five different CWEs. We compare the performance results with those provided in (Bilgin et al., 2020). Our evaluation is based on standard metrics such as precision, recall, and F1-score. The tabulated results summarized in Table 1 present a comparative analysis of vulnerabilities between the method in (Bilgin et al., 2020) and our proposed image-based solution: This comparison underscores the improved performance of our image-based solution across various CWEs, with enhanced precision, recall, and F1-scores. It suggests the potential effectiveness of our approach in accurately detecting vulnerabilities, highlighting its promise for robust vulnerability analysis in software systems.

5 THREATS TO VALIDITY

It is important to acknowledge and address potential threats to the validity of our research. We recognize several key threats to the validity of our findings.

5.1 External Validity

Our evaluation centered on C, providing a detailed examination of the CodeGrapher approach in a controlled setting. However, this may limit the generalizability of our findings to other programming languages. Future work will broaden comparisons across multiple languages to enhance generalizability.

5.2 Scalability and Granularity

Our experiments primarily focused on predicting vulnerabilities within individual functions using CodeGrapher. However, software development occurs at multiple levels, posing a threat to external validity and generalizability. Future research will explore CodeGrapher’s scalability and versatility across different levels of analysis, ensuring a comprehensive evaluation of its applicability.

5.3 Vulnerability Type Coverage

In this study, we aimed to cover various vulnerability types, including buffer overflows, NULL pointer dereference, and pointer subtraction. However, the software vulnerability landscape is vast and evolving, potentially omitting certain types. This could impact construct validity, as our findings may not represent the full spectrum of vulnerabilities. Future research will address this by evaluating additional vulnerability types, enhancing the comprehensiveness of the CodeGrapher approach.

By recognizing and addressing these threats to validity, we aim to provide a more robust and comprehensive foundation for the application of CodeGrapher in software vulnerability prediction, extending its relevance to a wider array of programming languages, levels of analysis, and vulnerability types.

6 CONCLUSION

Software vulnerabilities pose significant threats to system security, motivating the development of effective detection methods. ML algorithms hold promise, yet varying source code lengths challenge accuracy, leading to false results. To overcome this, we introduce CodeGrapher, preserving semantic relations by converting numerical vectors into image sets for ML input. Image size adjustability ensures consistency, while similarity distance metrics maintain code element relationships. Experimental validation demonstrates superior performance. Future enhancements include adjusting window size and CNN parameters to improve accuracy, advancing software security.

ACKNOWLEDGEMENTS

This work was funded by The Scientific and Technological Research Council of Turkey, under 1515 Frontier R&D Laboratories Support Program with project no: 5169902.

REFERENCES

- (2023a). Debian – The Universal Operating System.
- (2023b). GitHub: Let’s build from here.
- Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29.
- Bamler, R. and Mandt, S. (2017). Dynamic word embeddings. In *ICML*, pages 380–389. PMLR.
- Bendersky, E. (2019). Github–eliben/pycparser: Complete c99 parser in pure python.
- Bilgin, Z., Ersoy, M. A., Soykan, E. U., Tomur, E., Çomak, P., and Karaçay, L. (2020). Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684.
- Black, P. E. and Black, P. E. (2018). *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology.
- Chen, L., Ye, W., and Zhang, S. (2019). Capturing source code semantics via tree-based convolution over api-enhanced ast. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 174–182.
- Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., and Wu, Y. (2019). Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In *IJCAI*, pages 4665–4671.
- Fang, Y., Liu, Y., Huang, C., and Liu, L. (2020). Fastembed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm. *Plos one*, 15(2):e0228439.
- Feutrill, A., Ranathunga, D., Yarom, Y., and Roughan, M. (2018). The effect of common vulnerability scoring system metrics on vulnerability exploit delay. In *CANDAR*, pages 1–10. IEEE.
- Halepmollası, R., Hanifi, K., Fouladi, R. F., and Tosun, A. (2023). A comparison of source code representation methods to predict vulnerability inducing code changes.
- Hanifi, K., Fouladi, R. F., Unsalver, B. G., and Karadag, G. (2023). Software vulnerability prediction knowledge transferring between programming languages. *arXiv preprint arXiv:2303.06177*.
- Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., Hamilton, L. H., Centeno, G. I., Key, J. R., Ellingwood, P. M., McConley, M. W., Opper, J. M., Chin, P., and Lazovich, T. (2018). Automated software vulnerability detection with machine learning. *CoRR*, abs/1803.04497.
- Henkel, J., Lahiri, S. K., Liblit, B., and Reps, T. W. (2018). Code vectors: understanding programs through embedded abstracted symbolic traces. In *ACM Joint Meeting on, ESEC/SIGSOFT FSE*, pages 163–174.
- Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. (2020). Learning and evaluating contextual embedding of source code. In *ICML*, pages 5110–5121. PMLR.
- Lin, G., Wen, S., Han, Q.-L., Zhang, J., and Xiang, Y. (2020). Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848.
- Lozoya, R. C., Baumann, A., Sabetta, A., and Bezzi, M. (2021). Commit2vec: Learning distributed representations of code changes. *SN Comput. Sci.*, 2(3):150.
- Palit, T., Moon, J. F., Monrose, F., and Polychronakis, M. (2021). Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937. IEEE.
- Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE.
- Şahin, S. E., Özyedierler, E. M., and Tosun, A. (2022). Predicting vulnerability inducing function versions using node embeddings and graph neural networks. *Information and Software Technology*, page 106822.
- Sane, P. (2020). Is the owasp top 10 list comprehensive enough for writing secure code? In *Proceedings of the 2020 International Conference on Big Data in Management*, pages 58–61.
- Schiewe, M., Curtis, J., Bushong, V., and Cerny, T. (2022). Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761.
- Telang, R. and Wattal, S. (2007). An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software engineering*, 33(8):544–557.
- Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.