k-Path CodeBERT for Vulnerability Detection

Abhinav Jain, aj70 [1]

1. Introduction

The pervasive integration of large language models (LLMs) has flourished in the domain of code generation and comprehension (Austin et al., 2021; Zan et al., 2023; Li et al., 2022). These models, exemplified by their prowess in tasks such as clone detection, natural language to code generation, and program search, have become instrumental in enhancing the efficiency and efficacy of software development processes. Their adeptness in discerning intricate patterns within codebases has been particularly noteworthy (Lu et al., 2021; Wang et al., 2021).

In addition to their established utility in diverse coding tasks, LLMs are increasingly being recognized for their potential in addressing security concerns within software systems (Ahmad et al., 2023; Steenhoek et al., 2023). One critical aspect of software security is vulnerability detection. Vulnerabilities, in the context of software, represent points in the code where inadvertent programming errors or oversights create vulnerabilities that could be exploited by malicious actors. These vulnerabilities are pervasive in existing codebases, such as the Linux operating system (Yamaguchi et al., 2014), and present a substantial attack surface for potential adversaries.

Consider, for instance, a common vulnerability in the C programming language: *buffer overflow*. This vulnerability arises when a program writes more data into a buffer than it can hold, leading to memory corruption (See Figure 1a(a)). Such vulnerabilities expose critical systems to potential exploits, making them imperative to identify and rectify.

While LLMs have demonstrated promise in the domain of vulnerability detection (Feng et al., 2020; Ahmad et al., 2023; Xia et al., 2023), their approach often involves the consideration of a string representation of the code. However, this representation may lack the depth necessary to capture the nuanced intricacies associated with potential security vulnerabilities.

To address this limitation, our research delves into the integration of program analysis tools for static analysis of code. By leveraging tools that generate code property graphs, such as control flow and data flow graphs, we aim to augment the capabilities of LLMs in discerning vulnerabilities within software systems.

Illustrating with the earlier example of a buffer overflow vulnerability, we demonstrate how static analysis can directly unveil the underlying security issue. By comprehensively examining the control flow and data flow within the code, static analysis provides insights into potential vulnerabilities that may elude purely string-based representations.

In this paper, we explore these code properties and investigate their impact on the effectiveness of LLMs in vulnerability detection. Our study aims to contribute valuable insights towards advancing the robustness of vulnerability detection methodologies in contemporary software development.

2. Problem Formulation

The primary goal of vulnerability detection can be defined as follows: With a dataset $\mathcal{D} = \{x_n, y_n\}_{n=1}^{N}$ containing instances of both vulnerable and non-vulnerable code blocks, the aim is to employ supervised learning to train a model θ that can predict whether a given functional block of code x implements a vulnerable procedure $(y = 1)$ or not $(y = 0)$.

3. Code Property Graphs for Static Analysis

Code Property Graphs (CPGs) (Yamaguchi et al., 2014) offer a comprehensive representation of source code, combining the characteristics of abstract syntax trees, control flow graphs, and program dependence graphs into a unified data structure. Control Flow Graphs (CFGs) explicitly outline the sequential execution of code statements and the conditions governing path selection, whereas Program Dependence Graphs (PDGs) (Ferrante et al., 1987) meticulously identify all program statements and predicates influencing the value of a variable at a specific statement.

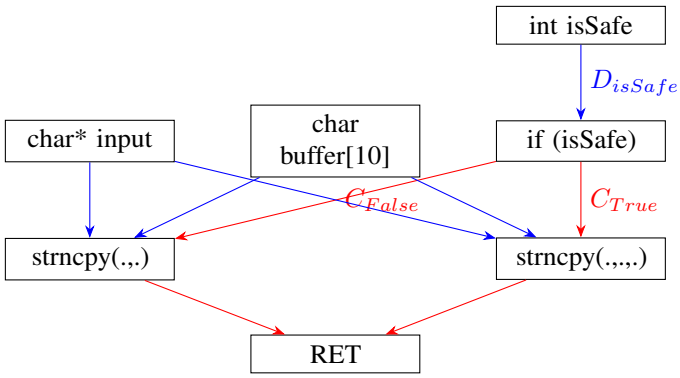
Unlike the limitations associated with string representations of code, the analysis facilitated by a code property graph-based data structure proves often more effective in capturing vulnerabilities, including resource leaks, division by zero, failure to release locks etc. In such scenarios, vulnerabilities can be specifically linked to control flow or data flow-based

[1]Department of Computer Science, Rice University, Houston, TX. Correspondence to: Nathan Dautenhahn <ndd@rice.edu>.

```c
void copyFunction(const char* input, int isSafe) {
    char buffer[10];

    if (isSafe) {
        // Safely copy input to buffer with bounds checking
        strncpy(buffer, input, sizeof(buffer) - 1);
        buffer[sizeof(buffer) - 1] = '\0';
        printf("Safe Copy: %s\n", buffer);
    } else {
        // No bounds checking, potential buffer overflow vulnerability
        strcpy(buffer, input);
        printf("Unsafe Copy: %s\n", buffer);
    }
}
```

(a) Vulnerable function block written in C



(b) Program dependence graph for the functional block. **(Red)** are the control dependence edges and **(Blue)** are data dependence edges.

Figure 1: An illustration of an example in C that contains a buffer overflow vulnerability within an if-else structure

path traversals within the overall CPG (Sparks et al., 2007; Gascon et al., 2013; Yamaguchi et al., 2014). For instance, resource leaks manifest along control flow paths from a memory acquisition statement (e.g., a call to MALLOC) to the function's end without encountering a corresponding resource release statement (e.g., a call to FREE).

Let's take a look at buffer flow for another example. It often occurs wherever a length field involved in a copy operation is not checked and can be identified by traversing data flow paths that go through the copy operation without performing bound checking. Refer to Figure 1b for an illustration, where the path traversing through the ELSE body is unsafe due to a STRCPY operation that does not do proper bound checking.

4. k-path CodeBERT

While Code Property Graphs (CPGs) have demonstrated effectiveness in detecting certain vulnerabilities (Yamaguchi et al., 2014), their interpretive capabilities are constrained by an inability to grasp code semantics, particularly in identifying equivalent yet differently formulated expressions. This limitation becomes apparent in scenarios where nuanced code variations exist. Contrarily, large language models have proven to be adept at addressing this limitation, excelling in tasks such as clone detection by virtue of their ability to comprehend code semantics (Lu et al., 2021).

In light of these complementary strengths, our proposed k-path CodeBERT model synergistically leverages the CODEBERT model (Feng et al., 2020) for interpreting code semantics and code property graphs for revealing structural properties. This combined approach aims to harness the best of both worlds, enhancing the model's ability to discern vulnerabilities by capturing both the semantic nuances and structural intricacies present in the code.

To encode specific code properties, we adopt a systematic strategy. We identify a sub-graph within the code property graph that studies specific code property such as control flow or program dependence. Subsequently, we traverse from their sources to sinks, in order to gather paths. For each sample, this process allows us to collect a subset of k paths, which is then incorporated into the dataset. The resulting dataset is denoted as $\mathcal{D} = \left\{x_n, y_n, \{p_{n,i}\}_{i=1}^{k}\right\}_{n=1}^{N}$. Now, we present our working of our k-path CODEBERT model θ.

Encoding. Each path p_i is independently encoded, along with its corresponding functional block of code x, using CODEBERT. The source code and the path are separated by a special token [SEP]. Each node in the path is represented by the string for the code it encapsulates to preserve semantics during path traversal. The edges between nodes are denoted by a special token [EDGE]. The overall input and its representation generated by the model can be represented as -

$$[\text{TOKS}]_{p_k} = [\text{TOKS}]_{n_1}; [\text{EDGE}]; [\text{TOKS}]_{n_2}; \ldots$$
$$h_{k,[CLS]} = \texttt{CodeBERT}\big([\text{CLS}]; [\text{TOKS}]_x; [\text{SEP}]; [\text{TOKS}]_{p_k}\big)$$

where, n_j are the nodes traversed along the path p_k, $[\text{TOKS}]_{n_j}$ represent the tokens obtained after tokenising string representing n_j, ; represents the concatenation operator and $h_{k,[CLS]}$ is the hidden representation computed by the CODEBERT model for a special token [CLS] prefixed to the overall model whose representation serves as the representation for x and the given path p_k.

Vulnerability Prediction Head. The generated representaion $h_{k,[CLS]}$ is processed by a feed-forward layer containing Multilayer Perceptron (MLP) heads. This process generates a logit l, encapsulating the model's confidence in predicting whether the functional block contains a vulnerability. This can be represented as $l_k = FFN(h_{[\text{CLS}]})$.

k-path prediction. The $\{l_1, \ldots, l_k\}$ logits from k-paths are subjected to max-pooling, followed by the application of a sigmoid function for final probability prediction μ. Max-pooling is implemented with the intuition that if the model exhibits confidence in predicting a vulnerability for the function based on any given path, it is reflected in the overall prediction. Importantly, this approach can also be used to classify such paths as *nefarious* for prompting the model to believe a vulnerability exists; additionally providing localization within the code block and indicating where the vulnerability is most likely to manifest. This can be represented as -

$$\mu = \texttt{sigmoid}\big(\texttt{MaxPool}(l_1, \ldots, l_k)\big)$$

Training. Finally, we train the model by minimising the following Cross Entropy loss -

$$\theta \leftarrow arg\min \frac{1}{N} \sum_{j=1}^{N} -(y_j \log(\mu_j) + (1 - y_j) \log(1 - \mu_j))$$

5. Experiment Setup and Result Discussion

5.1. Dataset

In our experiments on vulnerability detection at the function level, we employ the real-world benchmark provided by CodeXGLUE, as documented by (Lu et al., 2021). The dataset, initially curated by (Zhou et al., 2019), comprises 27,318 functions that have been meticulously annotated as either vulnerable or non-vulnerable. These functions were extracted from security-related commits within two prominent and expansive open-source projects written in the C programming language, namely QEMU and FFmpeg. The dataset exhibits a diverse range of functionalities.

We use (Lu et al., 2021) curated dataset partitioning into training, validation, and test sets, employing an 80-10-10 split ratio.

5.2. Implementation

Property graph extraction. We utilize *joern*[1] to construct Code Property Graphs (CPGs). *Joern* serves as a platform for robust analysis of source code, bytecode, and binary code. It facilitates the generation of CPGs and enables code mining through search queries formulated in a Scala-based domain-specific query language. Subsequently, we extract the $k-$ shortest paths by traversing the CPG along edges labeled as 'CFG' for control flow analysis and 'CDG' + 'REACHING_DEF' for program dependence analysis. While our current framework considers shortest paths that may effectively capture vulnerabilities leading to an early exit from a method, we acknowledge a more sophisticated approach to selecting representative paths. This refinement, open for future exploration, can be easily integrated into our framework.

Model	Accuracy
CodeBERT	61.79
CodeBERT + CFG($k = 1$)	62.19
CodeBERT + CFG($k = 2$)	**63.18**
CodeBERT + PDG($k = 1$)	62.66
CodeBERT + PDG($k = 2$)	**62.81**

Table 1: Baseline comparison on the Devign test-split.

CodeBERT Hyper-parameters. We train each model for a total of 5 epochs using a Adam optimizer with learning rate of $2e-5$. The learning rate undergoes a linear decay post a warm-up phase of approximately 700 training steps. The training process is conducted with a batch size of 16, while the evaluation phase employs a batch size of 64. To ensure efficient processing, the maximum length for encoding the string representation of code is set at 384, while the encoding of a path is constrained to a maximum length of 128. Lastly, the threshold to consider a prediction as vulnerable is set to 0.5. For transparency and reproducibility, our codebase is made publicly available here[2].

5.3. Results

5.3.1. Baseline Comparison

In Table 1, we present a comparative analysis of the proposed model, incorporating two distinct code properties (CFG and PDG), against the plain CodeBERT, which operates without utilizing any paths. Each variant of our model undergoes training and testing with k shortest paths extracted from the corresponding graphs.

By incorporating a single path ($k = 1$), we observe an improvement in accuracy over CodeBERT, with further enhancement as the number of paths increases ($k = 2$). This implies that augmenting the string representation of code with additional code properties enhances the LLM's capability to detect vulnerabilities.

Notably, the percentage improvement achieved by including paths varies when transitioning from CFG to PDG as the underlying code property graph changes. This discrepancy can be attributed to CFG and PDG exposing distinct code properties to the LLM, leading to the discovery of different vulnerabilities. For instance, CFG reveals the order of statement execution, exposing vulnerabilities such as improper lock releases but missing those that require modeling of attacker-controlled data. Conversely, PDG sacrifices the

[1] https://joern.io/

[2] https://github.com/jabhinav/k-path-CodeBERT

k-path Model	$k_{test} = 2$	$k_{test} = 4$	$k_{test} = 8$
CFG ($k_{train} = 1$)	62.52	62.59	**62.81**
CFG ($k_{train} = 2$)	63.18	63.32	**63.51**
PDG ($k_{train} = 1$)	62.30	62.26	**62.45**
PDG ($k_{train} = 2$)	62.81	62.81	**63.14**

Table 2: Zero-Shot performance of the proposed models with different number of paths k_{test} mined during evaluation

order of execution but offers insights into the influence of variables and predicates on the values of other variables. This can unveil vulnerabilities related to buffer overflows, missing permission checks, insecure arguments, etc.

To validate this argument, considering the absence of vulnerability descriptions in our chosen dataset, we profiled the true positives detected by both CFG- and PDG-based models ($k = 2$). While both models agreed on 36.89% of vulnerable samples, the PDG-based model indeed identified 7.09% vulnerable samples overlooked by the CFG-based model. Similarly, CFG-based model detected 16.33% vulnerable samples overlooked by PDG-based model.

A subsequent inquiry arises: "What if we combine the strengths of both models?" To explore this, we created a mixture-of-expert (*MOE*) model that predicts the existence of a vulnerability if either the CFG- or PDG-based model asserts so. And it predicts a sample as benign if neither model indicates a vulnerability. This composite model improves the precision of vulnerability detection to 60.32%, surpassing the individual precisions of each model—53.23% for CFG-based and 43.98% for PDG-based models.

5.3.2. Ablation Study: Evaluating with different number of paths

In the previous section, we observed performance boost achieved by incorporating more paths during the training of the proposed model. Throughout that analysis, we maintained consistency by employing the same number of paths during evaluation as used in the corresponding model's training, denoted as $k_{train} = k_{test}$. However, given the substantial gap in resource usage between training and testing with large language models, coupled with the intrinsic flexibility of our proposed model that can accommodate any number of paths, we now investigate the model's efficacy under different conditions by varying the number of k_{test} paths during evaluation.

In Table 2, we observe a consistent performance improvement for each model as the number of paths used during evaluation increases, marked by $k_{test} \geq k_{train}$. While this suggests that encoding more paths could enhance the likelihood of capturing the vulnerability, this strategy faces

Code Property	$k_{train} = 1$	$k_{train} = 2$	$k_{train} = 4$
CFG	60.54	60.61	**61.68**
PDG	57.61	58.05	**59.85**

Table 3: Performance comparison of the model trained and tested with paths as input (here $k_{train} = k_{test}$)

limitations due to two primary factors. Firstly, the number of path traversals grows exponentially with the number of vertices ($\mathcal{O}(2^n)$), introducing scalability challenges. Secondly, proposed large language models incur resource usage proportional to the number of paths, further underscoring the need for efficient strategies that balance computational demands with analytical insights.

In other words, this experimental insight implies that not all path traversals within the underlying graph expose the inherent vulnerability. This prompts further investigation into devising optimal strategies for graph traversals to unveil paths that are more likely to represent it. We consider this avenue as part of our future work.

5.3.3. Ablation Study: Vulnerability detection with path-only models

In this section, we assess the performance of k-path CodeBERT when the string representation of the complete functional block of code is ablated from the input. Table 3 reveals that the detection accuracy indeed improves with the inclusion of more paths, as anticipated. This enhancement is attributed to the incorporation of additional traversals of property graphs, thereby providing a more comprehensive representation of the code. Notably, with $k_{train} = 4$, the performance of the CFG-based model approaches the baseline CodeBERT's performance in Table 1. This observation suggests that, with the proper selection of a subset of paths representing the desired code property and structure, it could serve as a viable substitute for the string-only representation of the code.

With the availability of more computational resources, a more in-depth exploration using a diverse selection of paths would be an interesting avenue for future investigation.

6. Conclusion and Future Work

In this work, we introduced a novel vulnerability detection model, k-path CodeBERT, combining the interpretive prowess of Large Language Models (LLMs) with the structural insights derived from Code Property Graphs (CPGs). Our approach systematically encodes both code semantics and structural properties, leveraging LLMs for nuanced code interpretation and CPGs for capturing inherent code structures. Through extensive experimentation, we demonstrated

the efficacy of our model in improving vulnerability detection, with performance enhancements observed by incorporating multiple paths during training. Our exploration of varying path numbers during evaluation further revealed the model's resilience and the potential for uncovering vulnerabilities. Our findings highlight the complementary strengths of LLMs and CPGs, underscoring the significance of a hybrid approach for comprehensive vulnerability detection in software systems.

Future work could explore smarter path selection strategies, moving beyond the simplistic consideration of shorter paths to potentially predicting program slices as candidate proposals (Li et al., 2021) for more refined path traversal. Additionally, we envision extending our k-path CODEBERT as a foundation model for unsupervised clustering of dynamic tainted flow-based paths (Newsome & Song, 2005), aiming to identify 'nefarious' outliers that deviate from system-wide invariant properties (Engler et al., 2001). This opens avenues for a more nuanced understanding of vulnerabilities in software systems.

References

Ahmad, B., Thakur, S., Tan, B., Karri, R., and Pearce, H. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215*, 2023.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pp. 45–54, 2013.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1035–1047, 2022.

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

Newsome, J. and Song, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pp. 3–4. Citeseer, 2005.

Sparks, S., Embleton, S., Cunningham, R., and Zou, C. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 477–486. IEEE, 2007.

Steenhoek, B., Rahman, M. M., Jiles, R., and Le, W. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2237–2248. IEEE, 2023.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Xia, C. S., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

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

Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., Yongji, W., and Lou, J.-G. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7443–7464, 2023.

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