MultiCode: A Unified Code Analysis Framework based on Multi-type and Multi-granularity Semantic Learning

Xu Duan
Intelligent Software Research Center,
Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
duanxu2019@iscas.ac.cn

Jingzheng Wu
Intelligent Software Research Center,
Institute of Software,
Chinese Academy of Sciences
Beijing, China
jingzheng08@iscas.ac.cn

Mengnan Du
Nanjing Institute of Software Technology,
Institute of Software,
Chinese Academy of Sciences
Nanjing University Of Chinese Medicine
Nanjing, China

Tianyue Luo
Intelligent Software Research Center,
Institute of Software,
Chinese Academy of Sciences
Beijing, China

Mutian Yang
Beijing ZhongKeWeiLan
Technology Co.,Ltd.
Beijing, China

Yanjun Wu
Intelligent Software Research Center,
Institute of Software,
Chinese Academy of Sciences
Beijing, China

***Abstract*—Code analysis is one of the common way to ensure software reliability. With the development of machine learning technology, more and more learning-based code analysis methods are proposed. However, most existing methods are aimed at specific code analysis tasks, which leads to the extra effort to implement different models for different tasks in industrial applications. In this paper, we propose MultiCode, a novel unified code analysis framework, which learns code semantic information of different types and granularities to cover the semantic information required by different tasks, so that it can be effectively adapted to multiple tasks with higher accuracy. To prove the effectiveness of MultiCode, we demonstrate and evaluate it on two common tasks: vulnerability detection and code clone detection. Experimental results show that MultiCode achieves F1-scores of 94.6%, 92.5% and 97.1% on SARD-BE, SARD-RME and OJClone datasets, which is significantly higher than the advanced existing methods.**

***Index Terms*—Machine learning, Graph neural networks, Code analysis.**

I. Introduction

In recent years, machine learning technology has developed rapidly, which can learn potential laws from a large amount of existing data in a data-driven way, and then make predictions on unknown data. Due to this advantage, more and more machine learning methods are gradually being applied in code analysis tasks to achieve higher preformance. By training on the large amount of code data, machine learning models can learn the potential laws of code data in specific tasks, and thus avoid the shortcomings of traditional methods such as heavy reliance on expert knowledge and difficulty in covering complex and changeable code scenarios.

This work is supported by National Key R&D Program of China (No.2018YFB0803600)

However, most existing methods [5], [8] are aimed at specific code analysis tasks, which leads to the extra effort in industrial applications for different tasks. Even if there are a few unified models [9] for multiple tasks, there is still room for improvement in their accuracy due to the limited considered semantic information. The key of the learning-based code analysis method lies in the learning of semantic information, which determines whether the model can identify the essential factors in code analysis tasks. More importantly, a single granularity or a single type of semantic information is not sufficient to represent code features. For example, AST (Abstract Syntax Tree) represents the syntax structures, which can model code well in code clone detection task. Nevertheless in vulnerability detection tasks, buffer overflows caused by insufficient input validation are more fully reflected in control and data dependencies, which should be modeled with PDG (Program Dependency Graph). Moreover, the coarse-grained features of the code are enough to determine its category when classifying algorithms, but in the vulnerability detection task, a small error can cause vulnerabilities, which should be modeled with more fine-grained features.

In this paper, we propose MultiCode, a novel unified code analysis framework, which learns code semantic information of different types and different granularities, so that it can be effectively adapted to multiple tasks with higher accuracy. Similar to word, sentence, and paragraph levels in natural language, MultiCode divides a code snippet into two levels with different granularities, which are intra-statement level and inter-statement level. Besides, AST is used to model the syntax structure within a statement, which is called intra-statement level AST. Tree-based convolution is conducted over the AST

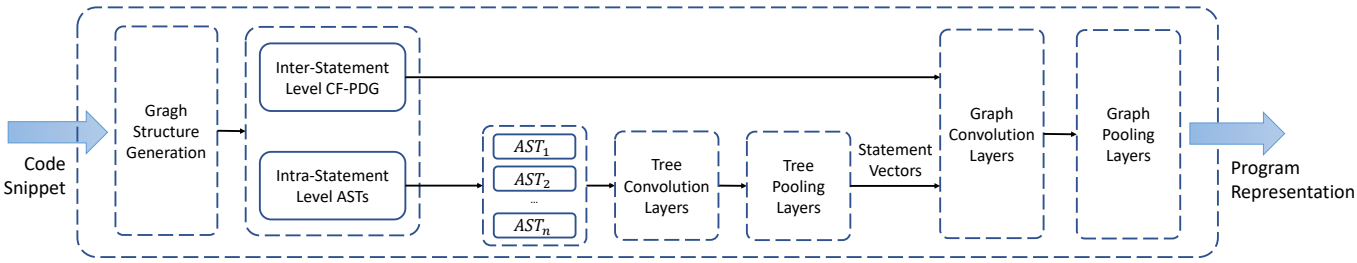


Fig. 1. Overall framework of MultiCode.

to learn the distributed representation of statements. Then, CFG and PDG are used to model control flow, control and data dependencies between the statements, which is called inter-statement level CF-PDG. Similarly, graph-based convolution is conducted over the CF-PDG to learn the final distributed representation of the code snippet. The above three graph structures greatly cover different types of semantic information in code. Through simple adaptation, MultiCode can be applied to multiple code analysis tasks.

To prove the effectiveness of MultiCode, we evaluate it on two completely different tasks, which are vulnerability detection and code clone detection. SARD-BE, SARD-RME and OJClone are selected as benchmark datasets on the two tasks. For baselines, we select not only TBCNN [9] to compare the capabilities of learning code representations, but also professional tools on the specific tasks, such as Flawfinder, RATS and Deckard, to prove that the model built with MultiCode performs adequately well on the corresponding task. Experimental results show that the models built with MultiCode can achieve F1-scores of 94.6%, 92.5% and 97.1% on the SARD-BE, SARD-RME and OJClone benchmark datasets, which are significantly better than those of the other advanced models.

The contributions of our work are summarized as follows:

- We design a unified code analysis framework called MultiCode, which can handle multiple code analysis tasks by learning multiple types and granularities of semantics.
- We implement a prototype of MultiCode, which can automatically generate graph structures and learn distributed vector representations of C programs.
- We evaluate MultiCode on two completely different tasks. Experimental results show that it can achieve higher accuracy than the other advanced tools.
- MultiCode has been used by Beijing ZhongKeWeiLan Technology Co.,Ltd., and is serving several large-scale Internet and government enterprises, which shows that MultiCode has high industrial value.

II. Related Work

With the great success of machine learning techniques in many research fields, more and more studies have applied them to analyze code. Many studies model the semantics of the code uniformly. Due to the complicated structural information in code, TBCNN [9] learns semantics via the tree-based convolutions over the AST. Code2vec [1] makes use of a collection of paths in AST to model the code. To address the problem that some key information such as tokens and APIs cannot be explicitly expressed in AST, Yu et al. [15] and Chen et al. [2] use token-enhanced and API-enhanced AST to model semantics of code, respectively.

Moreover, many studies that apply machine learning to specific code analysis tasks. In terms of vulnerability detection, VulDeePecker [8] directly treats the source code as a token sequence, and then uses word2vec to learn the code representation. Kim et al. [6] leverages a machine learning approach to model code representation using AST and CFG. For code clone detection, DeepSim [16] encodes CFG and DFG (Data Flow Graph) as a semantic feature matrix to detect code functional similarity. CDLH [14] utilizes AST-based LSTM to obtain the representation of code snippets. FCDetector [3] uses AST and CFG to model the source code, and utilizes word2vec and graph2vec technologies to embed the two structures, thereby realizing clone detection at the semantic level. The difference between MultiCode and the above methods is reflected in two aspects. First, the above methods cannot fully model the semantics in different graph structures. For example, FCDetector uses word2vec to learn the AST embeddings on the traversal sequence in preorder, but the sequence cannot fully reflect the hierarchical relationship of the tree. Second, some of the above methods are designed for specific tasks, and only contain the semantics required for the target task, resulting in the lack of semantics when facing other code analysis tasks.

III. Methodology

The overall framework of MultiCode is shown in Figure 1. For a given code snippet, MultiCode first generates the intra-statement level ASTs and the inter-statement level CF-PDG. Then, the statement representations are computed by tree-based convolutions and poolings over the intra-statement level ASTs. After obtaining the vector representations of the statements, graph-based convolutions and poolings are conducted to learn a representation of the code snippet, which can be used for various code analysis tasks.

A. Code Graph Structures

Before using code as input to MultiCode, the semantics should be modeled initially with graph structures. As mentioned in Section 1, the graph structures should include not

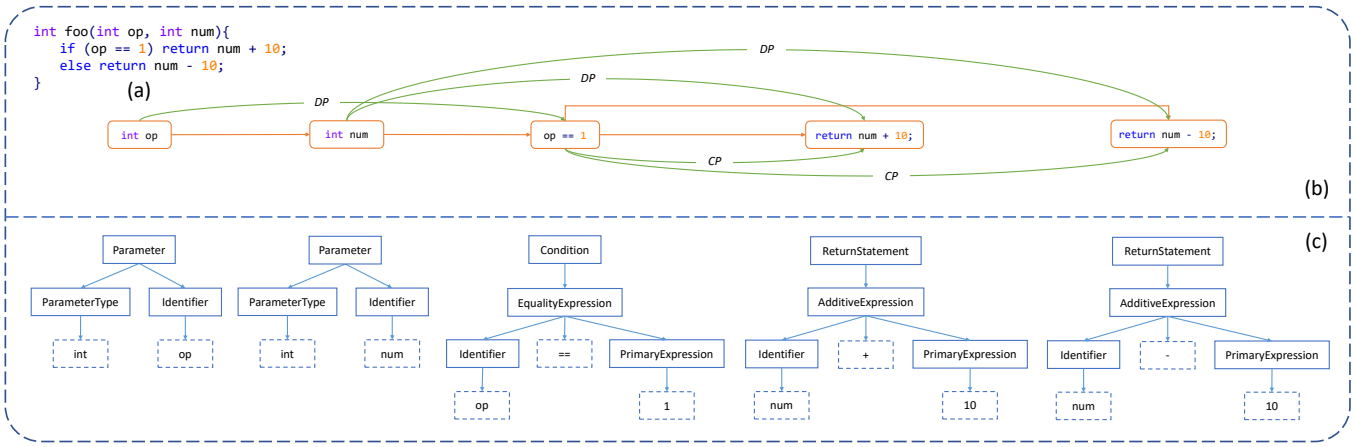


Fig. 2. (a) A snippet of example code. (b) The inter-statement level CF-PDG. Orange nodes represent statements in code. Orange lines represent the control flows, corresponding to the edges in the CFG. Green lines labeled DP or CP represent the data dependencies or control dependencies, respectively, corresponding to the edges in the PDG. (c) The intra-statement level ASTs. Blue nodes and lines represent the nodes and edges of AST, respectively. The dotted nodes represent token-enhanced nodes.

only different types, but also different granularities of semantic information. In terms of the different types of semantic information, AST, CFG and PDG are used to represent the semantics, including syntax structure, control flow, control dependencies and data dependencies in code. Comprehensive semantics can make the model effectively adapt to different code analysis tasks without a lot of changes. Although some semantics are not so important for certain specific tasks, subsequent deep neural networks can filter out the key features for the current task through layer-by-layer feature learning. Moreover, inspired by the word, sentence and paragraph level in natural language, we divide a code snippet into two levels, intra-statement level and inter-statement level. Features in the intra-statement level are fine-grained, while features in the inter-statement level are coarse-grained.

After selecting the graph structures and granularities, a challenge is how to combine them together. Considering that control flows and dependencies are more reflected between statements, and the syntax structure within a statement is more important. MultiCode combines them as the intra-statement level AST and inter-statement level CF-PDG. For better understanding, Figure 2 shows an example of intra-statement level AST and inter-statement level CF-PDG. Each AST in Figure 2(c) corresponds to a statement in the code. Besides, since the original AST only contains information of node types, while tokens (e.g., identifiers, operators) would also carry a lot of semantic information, we enhance the original AST with tokens. Concretely, when an AST node contains a token, the token is added as a child node to the corresponding AST node. For implementation, Joern[1] is used to generate the initial graph structures.

[1]https://joern.readthedocs.io/en/latest/index.html

B. Intra-Statement Level Tree Convolution

Before learning the features of the ASTs, each node must be given an initial vector representation. Since most of programming languages allow developers to arbitrarily define identifiers under certain rules, it will cause serious *Out of Vocabulary* problem [4]. To deal with this problem, we use Position-Aware Character Embedding (PACE) [15] to generate the initial representations of the AST node types and tokens. For a given string, PACE treats it as a linear combination of one-hot encoding of its characters, where weights reflect the position of the characters in the string. Suppose a string s is composed of the characters $c_1, c_2, ..., c_k$, then the representation of s is calculated as follows:

$$vec(s) = \sum_{i=1}^{k} \frac{k-i+1}{k} \times vec(c_i) \tag{1}$$

where $vec()$ denotes the representation of string or character. Since semantically similar AST node types or tokens usually have some same characters (e.g., *AndExpression* and *OrExpression*), PACE can effectively express their relevance.

After the generation of initial vector representations, tree-based convolutions are conducted over the intra-statement level ASTs to learn the semantic representations of the statements. Unlike grid-based convolution, tree-based convolution slides the convolution kernel on a triangular subtree composed of a parent and its child nodes [9]. In each convolution, the representation of the parent node is updated by its child nodes once. Suppose the parent representation is $\vec{h}_0$, and the children representations are $\vec{h}_1, \vec{h}_2, ..., \vec{h}_n$, then the new representation $\vec{h}'_0$ is calculated as follows:

$$\vec{h}'_0 = \sigma(\sum_{i=0}^{n} \boldsymbol{W}_i \vec{h}_i + \boldsymbol{b}) \tag{2}$$

where $\boldsymbol{W}_i$ denotes the weight matrix of node i. $\boldsymbol{b}$ denotes the bias term. σ denotes an activation function such as *tanh*.

It should be noted that the number of weight matrices can be uncertain, due to the varying number of child nodes in AST. To address this code, we treat AST as a *continuous binary tree* [11]. Each weight matrix $\boldsymbol{W}_i$ is calculated by a linear combination of three weight matrices $\boldsymbol{W}^t$, $\boldsymbol{W}^l$ and $\boldsymbol{W}^r$ as follows:

$$\boldsymbol{W}_i = \eta_i^t \boldsymbol{W}^t + \eta_i^l \boldsymbol{W}^l + \eta_i^r \boldsymbol{W}^r \tag{3}$$

$$\eta_i^t = \frac{d_i - 1}{d - 1} \tag{4}$$

$$\eta_i^r = (1 - \eta_i^t)\frac{w_i - 1}{w - 1} \tag{5}$$

$$\eta_i^l = (1 - \eta_i^t)(1 - \eta_i^r) \tag{6}$$

where η_i^t, η_i^r and η_i^l represent the coefficients of the three weight matrices, which reflect the position of the nodes in the sliding window. d_i and w_i denote the vertical and horizontal position of the nodes in the window, respectively. d and w denote the depth and width of the window, which can be regarded as hyperparameters. In this paper, d is set to 2, which means that each convolution involves only two generations of nodes. w is set to the maximum number of child nodes, which means that all children of a single node are processed in each convolution. When the number of child nodes is less than w, zero vectors are padded. With continuous binary trees, the number of weights to be learned is fixed, and the position of nodes in the tree is fully considered.

After the tree-based convolutions, the representation of each node is updated, and the shape of the tree is unchanged. To obtain the final vector representation of the entire tree, pooling is applied over the tree. Specifically, the maximum value in each dimension is extracted and concatenated as the final vector representation. So far, the vector representations of the statements in the code are obtained, which will be used as the initial vector representations of the nodes in inter-statement level CF-PDG for subsequent processing.

C. Inter-Statement Level Graph Convolution

In the previous steps, the fine-grained semantic information within statements are successfully learned through tree-based convolutions and pooling. Then, the coarse-grained semantic information between the statements, such as control flows, control dependencies, and data dependencies are learned through graph-based convolutions. In the graph-based convolutions, the representation of a node is updated by the representations of its neighbors [7], [10]. Each convolution updates the representations of nodes once to learn the features of one-hop neighbors. Further, multiple convolutions are iteratively applied to learn the features of multi-hop neighbors. Besides, attention mechanism is also used in the convolutions to capture the different importances of different neighbors [13]. Suppose a node representation is $\vec{h}_0$, and its ont-hop neighbor representations are $\vec{h}_1, \vec{h}_2, ..., \vec{h}_n$, then the new representation $\vec{h}_0'$ is calculated as follows:

$$\vec{h}_0' = \sigma(\sum_{i=0}^{n} \alpha_{0i} \boldsymbol{W} \vec{h}_i) \tag{7}$$

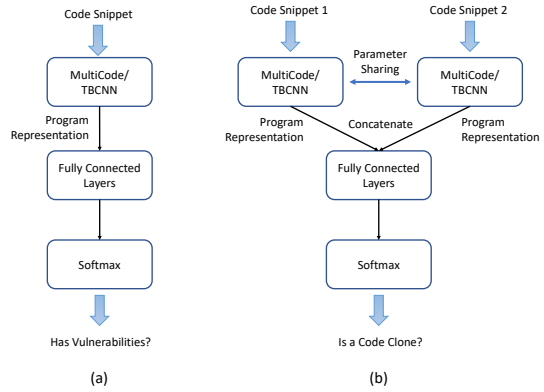


Fig. 3. (a) Model on the vulnerability detection task. (b) Model on the code clone detection task.

where σ denotes an activation function, such as elu. $\boldsymbol{W}$ represents the weight matrix. α_{0i} represents the normalized attention weight between the node 0 and node i. Note that there are multiple types of edges in CF-PDG, representing control flows, control and data dependencies, thus the features of the edges should be considered in the calculation of attention weight. Concretely, α_{0i} is calculated as follows:

$$\alpha_{0i} = \frac{exp(\sigma(\vec{a}^T(\boldsymbol{W}\vec{h}_0 || \boldsymbol{W}\vec{h}_i || \vec{h}_{(0,i)})))}{\sum_{k=0}^{n} exp(\sigma(\vec{a}^T(\boldsymbol{W}\vec{h}_0 || \boldsymbol{W}\vec{h}_k || \vec{h}_{(0,k)})))}, \tag{8}$$

where $||$ denotes the concatenation operation. $\vec{a}^T$ denotes the transpose of the attention weight vector. $\vec{h}_{(0,i)}$ represents the feature vector of the edge from node 0 to node i, which is the one-hot encoding of the edge type.

After the graph-based convolutions, pooling is also applied to obtained the final representation of the entire graph. The final vector representation is composed of the maximum value in each dimension. During the graph-based convolutions and pooling, the semantic information between the statements are learned. Finally, the distributed vector representation of the code is obtained, which greatly contains semantics of different types and granularities. Taking this representation as input for subsequent customized network layers can effectively adapt to different tasks, which is shown in Figure 3.

IV. Experiments

This section will introduce experimental details. The following experiments are conducted on a machine equipped with TITAN RTX 24GB GPU, Intel Xeon Gold 6126 CPU @ 2.60GHz and 128GB RAM.

A. Vulnerability Detection

a) Task Description: A vulnerability is a program defect that can be exploited by attackers to cause damage to software security. Discovering vulnerabilities by analyzing source code is a common method of vulnerability detection. In this paper, we consider vulnerability detection as a binary classification problem. In other words, for a given code snippet, it should

TABLE I
FALSE POSITIVE RATE (FPR), FALSE NEGATIVE RATE (FNR), PRECISION (P), RECALL (R) AND F1-SCORE (F1) COMPARISON OF DIFFERENT TOOLS ON THE THREE DATASETS.

Dataset	Tools	FPR (%)	FNR (%)	R (%)	P (%)	F1 (%)
SARD-BE	Flawfinder	82.0	11.5	88.5	38.4	53.6
	RATS	65.1	27.0	73.0	39.3	51.1
	TBCNN	25.2	41.4	58.6	60.9	59.7
	MultiCode	0.4	9.5	90.5	99.2	94.6
SARD-RME	Flawfinder	50.4	45.5	54.5	31.8	40.2
	RATS	38.5	55.2	44.8	33.4	38.3
	TBCNN	3.0	55.5	44.5	87.7	59.0
	MultiCode	6.1	1.7	98.3	87.4	92.5
OJClone	Deckard	0.7	92.0	8.0	92.1	14.7
	TBCNN	4.9	1.4	98.6	95.3	96.9
	MultiCode	1.4	4.4	95.6	98.6	97.1

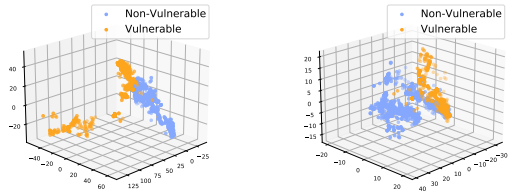
be classified as positive if it has vulnerabilities, otherwise be classified as negative.

b) Dataset: Software Assurance Reference Dataset (SARD)[2] contains a large number of vulnerability instances. Each instance contains a positive sample with a specific vulnerability, and several negative samples where the vulnerability has been fixed in different ways. we collect two types of vulnerabilities from SARD, Buffer Error (BE) and Resource Management Error (RME). The reason for choosing these two types of vulnerabilities is that they are very common in C programs and have the largest number of instances in SARD to support model training. Finally, 4242 positive samples and 6669 negative samples in the SARD-BE dataset, and 3033 positive samples and 6550 negative samples in the SARD-RME dataset are collected. Besides, both the two datasets are divided into training, validation and test sets at a ratio of 6:2:2.

c) Baselines: On the vulnerability detection task, since TGVD deals with C programs, we select three tools for C programs as baselines, which are Flawfinder[3], RATS[4] and TBCNN [9]. Flawfinder and RATS are two widely used vulnerability detection tools, which detect vulnerabilities based on lexical analysis. TBCNN is a model for programming language processing, which models code as an AST and then processes it using only tree-based convolutions. The above three baselines cover both professional tools for vulnerability detection, and models for unified code analysis, which can effectively illustrate the effectiveness of MultiCode.

d) Results: Since different tools work differently, we experiment with them in different ways. Flawfinder and RATS are directly used to scan the code in the test set, because they analyze code based on lexics. Since MultiCode and TBCNN are unified code analysis framework, fully connected layers and softmax are set behind them to output the final classification results, which is shown in Figure 3(a), .

[2]https://samate.nist.gov/SRD/index.php
[3]https://dwheeler.com/flawfinder/
[4]https://code.google.com/archive/p/rough-auditing-tool-for-security/



(a) (b)

Fig. 4. Code representations learned by MultiCode on (a) the SARD-BE dataset and (b) the SARD-RME dataset.

In addition, we find that some tokens (e.g., variable or function names) in SARD contain words like *"bad"* and *"good"*, indicating whether the case is vulnerable. However, it can lead to *data leakage* when evaluating the model, because such variable names do not appear in real code. To solve this problem, we obscure the tokens that contain these sensitive words. Concretely, these tokens are all renamed to *"fuzzed_token"* to eliminate their negative impacts.

As shown in Table I, the model built with MultiCode outperforms the other three baselines significantly, which achieves F1-scores of 94.6% and 92.5% on the SARD-BE and SARD-RME datasets. The main reason is that vulnerability detection involves different types and granularities semantics in code. They are successfully considered by MultiCode, but not by TBCNN.

To analyze whether MultiCode has learned the semantics related to the vulnerabilities, we visualize the code representations output from the final graph pooling layer with PCA dimensionality reduction. As shown in Figure 4, the vulnerable and non-vulnerable code is obviously separated from each other, which shows that MultiCode has successfully captured the differences in the semantics.

B. Code Clone Detection

a) Task Description: A code clone is a pair of similar code snippets that share the same semantics. Code clones can be divided into Type-1, Type-2, Type-3 and Type-4 based on different levels of similarity [12]. The higher the type level, the more syntactical similarity is ignored and semantic similarity is stressed. Therefore, to fully evaluate MultiCode, we perform experiments on Type-3 and Type-4 code clone detection. Concretely, we also consider it as a binary classification problem. In other words, for a given code pair, it should be classified as positive if it is a Type-3 or Type-4 code clone, otherwise be classified as negative.

b) Dataset: OJClone dataset, released by [9], is originally used to evaluate code classification problems, but it is gradually adjusted to evaluate code clone detection problems. The initial dataset contains 104 questions collected from an OJ system[5], each of which has 500 code snippets that solve the question. We select the first 4 questions from the initial dataset. These 2000 selected code snippets are divided into subsets of sizes 1200, 400 and 400 for training, validation and test. After

[5]http://poj.openjudge.cn/

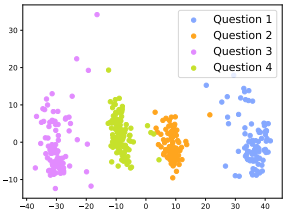


Fig. 5. Code representations learned by MultiCode on the OJClone dataset.

that, we combine the code snippets in each subset separately to obtain code pairs for evaluating code clone detection. A code pair that belongs to the same question can be regarded as a Type-3 or Type-4 code clone, otherwise not a code clone. In addition, we find that positive samples only accounted for a very small number of these code pairs, which would cause *data imbalance* problem. To solve this problem, we randomly resample 60,000, 20,000, and 20,000 samples as the training set, the validation set, and the test set, where the number of positive and negative samples is equal.

c) Baseline: On the code clone detection task, we select two tools as baselines, which are Deckard [5] and TBCNN [9]. Deckard is a popular AST-based code clone detection tool, which measures code similarity through LSH (Locality Sensitive Hashing) and clustering. To compare the performance difference between MultiCode and other unified code analysis frameworks, we also experiment with TBCNN on the code clone detection task.

d) Result: In the experiments, Deckard is used directly to generate a clone detection report, and its accuracy is calculated by checking whether the code clones are in the report. Since the input of clone detection is a code pair, we construct siamese models with TBCNN and MultiCode as shown in Figure 3(b). After learning the representations of the two code snippets, they are concatenated together. Then, fully connected layers and softmax are used to output the classification results.

As shown in Table I, the F1-score of the model built with MultiCode reaches 97.1% on the OJClone dataset, which is higher than Deckard and TBCNN. The main reason for Deckard is that it merely focuses on the syntax of code, which makes it fail to identify the high level code clones and shows a particularly low recall. The close results of MultiCode and TBCNN show that a single granularity AST is almost sufficient to model code on the code clone detection task. Nevertheless, MultiCode takes into account more types of features, which brings a slightly higher accuracy than TBCNN.

Moreover, we also visualize the dimensionality reducted code representation on the code clone detection task. As shown in Figure 5, the code representations that belong to different questions are concentrated in different clusters, and the semantically dissimilar code are also far away from each other, which shows that MultiCode has successfully captured the code semantics.

V. Conclusion

In this paper, we propose a unified code analysis framework called MultiCode. It takes into account different types and granularities of semantic information at the same time to enhance the capabilities of learning code features. Besides, intra-statement level AST and inter-statement level CF-PDG are proposed to combine different types and granularities of semantics in a subtle way. We evaluate MultiCode on vulnerability detection and code clone detection tasks. Experimental results show that the models built with MultiCode achieve F1-scores of 94.6%, 92.5% and 97.1% on SARD-BE, SARD-RME and OJClone datasets, which demonstrates the effectiveness of MultiCode.

References

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *PACMPL*, vol. 3, no. POPL, pp. 40:1–40:29, 2019. [Online]. Available: https://doi.org/10.1145/3290353

[2] L. Chen, W. Ye, and S. Zhang, "Capturing source code semantics via tree-based convolution over api-enhanced AST," in *CF*, 2019, pp. 174–182. [Online]. Available: https://doi.org/10.1145/3310273.3321560

[3] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *ISSTA*. ACM, 2020, pp. 516–527. [Online]. Available: https://doi.org/10.1145/3395363.3397362

[4] V. J. Hellendoorn and P. T. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *ESEC/FSE*, 2017, pp. 763–773. [Online]. Available: https://doi.org/10.1145/3106237.3106290

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105. [Online]. Available: https://doi.org/10.1109/ICSE.2007.30

[6] J. Kim, D. Hubczenko, and P. Montague, "Towards attention based vulnerability discovery using source code representation," in *ICANN*, 2019, pp. 731–746. [Online]. Available: https://doi.org/10.1007/978-3-030-30490-4_58

[7] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[8] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *NDSS*, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

[9] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI*, 2016, pp. 1287–1293. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

[10] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *ICML*, 2016, pp. 2014–2023. [Online]. Available: http://proceedings.mlr.press/v48/niepert16.html

[11] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *KSEM*, 2015, pp. 547–553. [Online]. Available: https://doi.org/10.1007/978-3-319-25159-2_49

[12] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[13] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *ICLR*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[14] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *IJCAI*, 2017, pp. 3034–3040. [Online]. Available: https://doi.org/10.24963/ijcai.2017/423

[15] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *ICPC*, 2019, pp. 70–80. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00021

[16] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *FSE*, 2018, pp. 141–151. [Online]. Available: https://doi.org/10.1145/3236024.3236068