What You See is What it Means!

Semantic Representation Learning of Code based on Visualization and Transfer Learning

PATRICK KELLER, University of Luxembourg
ABDOUL KADER KABORÉ*, University of Luxembourg, CITADEL at Université Virtuelle du Burkina Faso
LAURA PLEIN, Saarland University
JACQUES KLEIN, University of Luxembourg
YVES LE TRAON, University of Luxembourg
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg

Recent successes in training word embeddings for NLP tasks have encouraged a wave of research on representation learning for source code, which builds on similar NLP methods. The overall objective is then to produce code embeddings that capture the maximum of program semantics. State-of-the-art approaches invariably rely on a syntactic representation (i.e., raw lexical tokens, abstract syntax trees, or intermediate representation tokens) to generate embeddings, which are criticized in the literature as non-robust or non-generalizable. In this work, we investigate a novel embedding approach based on the intuition that source code has visual patterns of semantics. We further use these patterns to address the outstanding challenge of identifying semantic code clones. We propose the WySiWiM (*"What You See Is What It Means"*) approach where visual representations of source code are fed into powerful pre-trained image classification neural networks from the field of computer vision to benefit from the practical advantages of transfer learning. We evaluate the proposed embedding approach on the task of vulnerable code prediction in source code and on two variations of the task of semantic code clone identification: code clone detection (a binary classification problem), and code classification (a multi-classification problem). We show with experiments on the BigCloneBench (Java), Open Judge (C) that although simple, our WySiWiM approach performs as effectively as state of the art approaches such as ASTNN or TBCNN. We also showed with data from NVD and SARD that WySiWiM representation can be used to learn a vulnerable code detector with reasonable performance (accuracy~90%). We further explore the influence of different steps in our approach, such as the choice of visual representations or the classification algorithm, to eventually discuss the promises and limitations of this research direction.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → *Software creation and management*; • **Security and privacy** → *Vulnerability management*; • **Human-centered computing** → **Visualization**.

Additional Key Words and Phrases: semantic clones, embedding, visual representation, representation learning

*Corresponding author

Authors' addresses: Patrick Keller, University of Luxembourg, patrick.keller@uni.lu; Abdoul Kader Kaboré, University of Luxembourg,, CITADEL at Université Virtuelle du Burkina Faso, abdoulkader.kabore@uni.lu; Laura Plein, Saarland University, laura_plein_1996@hotmail.fr; Jacques Klein, University of Luxembourg, jacques.klein@uni.lu; Yves Le Traon, University of Luxembourg, yves.letraon@uni.lu; Tegawendé F. Bissyandé, University of Luxembourg, tegawende.bissyande@uni.lu.

Manuscript submitted to ACM

1 INTRODUCTION

Semantic code clone identification is a long-standing challenge in software engineering [41]. It has applications in diverse automation tasks, including bug and vulnerable code prediction, program repair and synthesis, etc. Until recently, semantic clones were reliably identified using dynamic approaches, such as DyCLINK [56], which compare execution traces to decide whether two code fragments behave similarly.
Unfortunately, such approaches typically require high-coverage testing to guarantee accuracy. In consequence, they do not effectively scale, and are not usually practical since they require complete and executable code as input. Recent advances in neural networks have provided a new play field for researching static approaches that attempt to learn semantic representations of code via source code embeddings.

Semantic representation learning of source code has attracted significant attention in the research community in the last couple of years [2, 3, 12, 18, 29, 33, 38, 60–62]. Traditionally, the literature proposes approaches that process code directly or use a syntactic tree representation, where code is treated as sentences. Then, specific approaches inspired by techniques from the Natural Language Processing (NLP) field are used to yield embeddings of these "sentences". Various works in this realm face robustness issues [10] since simplification of Abstract Syntax Trees (AST), to cope with implementation constraints, weaken the capability of neural network models to capture real and complex semantics [63]. To address these limitations, the state-of-the-art ASTNN [62] approach proposes to split each large AST into a sequence of small statement trees, and recursively encodes the statement trees to vectors by capturing the lexical and syntactical information from statements. Although this approach shows promising results on benchmark samples, its reliance on lexical similarity eventually poses two challenges: (1) the model must be regularly trained on new datasets to allow the inner word2vec [44] model to capture new vocabulary; (2) the model could be misled by relying on tokens, given that two different library methods with the same names may have different semantics implemented outside the code fragment.

In this paper we propose to investigate another representation learning direction for capturing semantics. In contrast to recent works which focus on lexical and syntactical information to capture semantics, or process code to map with some pixel values, the intuition behind our approach is to leverage the power of visual representation: a program, either in its source code form or AST form can be viewed as an image that the programmer can analyse to look for structures that he/she has seen before, by applying his/her experiences. From those recognized structures, the programmer may identify patterns of functionality implementations or reveal vulnerable code fragments in his/her implementations. We follow this intuition to design the WySiWiM ("*What You See Is What It Means*") approach: instead of directly training a complex and opaque semantic representation or embeddings based on syntactical information in source code, we simply render source code into a visual representation[1]. Given the advances in the litterature of computer vision, the WySiWiM approach can directly benefit from the state-of-the-art, notably with transfer learning. After the visualization process, WySiWiM performs two different procedures. First, using the generated images, the visual structures of the code are extracted. To that end, a pre-trained image classification neural network, i.e., a neural network that has been trained on other image classification datasets is used to yield a vector of internal features which represent structural information of the input image (i.e., of the code). Optionally, the pre-trained network can be re-trained by adding samples of images representing code. Second, the feature vectors are used for learning to discriminate between samples implementing different semantics, just as a human developer would do. Eventually, we expect to leverage the produced classifiers for *code classification* (i.e., given a code fragment, predict its functionality label), *clone detection* (i.e., given a

[1]We do not claim that visualization is sufficient to perform all tasks. We are investigating a novel approach where features extracted from visualization artefacts may serve some downstream tasks. In any case we consider different formats of the code, including plain text and AST representations.

pair of code fragments, decide whether they are semantic clones) and *vulnerable code identification* (i.e., given a code fragment, predict if it contains a vulnerability or not).

Our main contributions are as follows:

- We propose a novel approach to semantic representation learning of code based on visual representations of code fragments. The WySiWiM approach is intuitively simple, and it builds on transfer learning to efficiently produce embeddings by exploiting powerful pre-trained image classification models from the field of computer vision.
- We apply the visual representation embeddings of WySiWiM to variant tasks of semantic code clone identification. Experimental validations against the BCB [58] and OJ [46] datasets show that WySiWiM is capable of keeping up with the state-of-the-art while providing significant potential for improvement.
- We also apply the visual representation embeddings to the task of vulnerable code prediction. Experimental validations using the dataset by Ponta et al. [52] show promising results.
- Finally, we provide an analysis of the influence of some implementation choices. Notably, we discuss the possibilities of visual representations of code and the challenges associated to duplicates in the clone benchmarks as part of threats to validity.

The remainder of this paper is structured as follows. In Section 2, we present background and review related work. In Section 3, we discuss the design of WySiWiM and its visualization methods, as well as the learning models that are used. Section 4 enumerates the research questions, overviews the datasets and details the experiments. In the Sections 5 and 6, respectively, we present our experimental results and provide some discussion about the threats to validity as well as the limitations of WySiWiM.

2 BACKGROUND & RELATED WORK

We provide in this section an overview of related work after defining essential concepts to facilitate the readers understanding of the WySiWiM approach description.

2.1 Definitions

2.1.1 Code clone concepts. We use the following clone-related definitions for our approach in Section 3.

- **Code Fragment:** Also referred to as code snippet, it is a piece of software. Formally, a code fragment is a contiguous set of code lines, which represents the input unit for clone identification. In practice, a code fragment can be a small set of instructions, a whole code block, a whole method or even a whole class.
- **(Code) Clone Pair:** It is a pair of code fragments that are syntactically or semantically similar to each other.
- **Clone Class:** This refers to a set of code fragments where any pairwise combination of code fragments is a clone pair.
- **Syntactic clone pair:** It is a clone pair where the code fragments were deemed similar according to a specific syntactic similarity measure.
- **Semantic clone pair:** It is a clone pair where the code fragments implement the same functionality, respectively the same behaviour or "semantics".
- **Candidate pair:** It refers to a pair of code fragments that may or may not constitute a clone pair. We use this terminology when we do not want to distinguish, or do not yet know, whether or not a pair of code fragments represents a clone pair.

We also recall for the reader the following well-accepted definitions of clone types [8, 54, 57]

- **Type-1**: Identical code fragments, except for differences in whitespace, layout, and comments.
- **Type-2**: Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences. They are also called *parameterized* or *renamed* clones.
- **Type-3**: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences. They are also called *gapped* or *near-miss* clones.
- **Type-4**: Syntactically dissimilar code fragments that implement the same functionality. They are also known as *functional* or *semantic* clones. In practice, Type-4 clones are often identified as Type-3 clones with an upper-bound threshold on the syntactic similarity with respect to a specific similarity measure. It should be noted that Type-1 to Type-4 clones are generally considered mutually exclusive.

⇒Semantic/functional clones are the primary target in this work.

2.1.2 Machine learning concepts. Since Section 3 develops a machine learning approach, we recall for the reader important concepts that we leverage. However, the inner details of these concepts are strictly out of the scope of this work.

Image Classification: Image classification is a well-studied problem in computer vision with several applications such as facial recognition in smart houses, object recognition for self-driving cars, or disease diagnostic in healthcare. The typical task consists in training a model to classify an image into a single or multiple predefined categories [26, 40]. Recent advances in deep neural networks have led to significant breakthroughs in image classification, where computers manage to match human-level accuracy under some conditions [20]. Convolutional Neural Networks (CNNs) is the most popular neural network model being used to address the image classification problem. The general idea behind CNNs is that a local understanding of an image is good enough. A convolution is then a weighted sum of the pixel values of the image, as a sliding window is moved across the whole image. Eventually, the CNNs extract low, middle and high-level features and classifiers in an end-to-end multi-layer fashion, and the number of stacked layers can enrich the "levels" of features. Simply explained, those image classification neural networks learn to recognize visual features from the images, such as structures and colorings.

However, CNNs have been shown to present a degradation problem when the deeper network starts to converge: with the network depth increasing, accuracy gets saturated and then degrades rapidly. Residual networks [20] (ResNets) have then been proposed to overcome this problem by explicitly letting deeper stacked layers to fit a residual mapping (instead of an underlying mapping as in CNNs). In this work, we will build on these tried and true models from the literature.

Transfer Learning: Transfer learning is a technique in machine learning which consists of transferring knowledge from a specific domain to another one [49, 64]. To give a real-world example to the concept, we could imagine that learning to play the piano can help a human to learn to play guitar later on. Even though the instruments are very different, the notes and the rhythms are the same, hence we can transfer this knowledge from one task to the other and thus reduce the effort to learn. In the transfer learning theory paper [9], authors have established the foundations of the approach. From this theory, given two domains (a source domain and a target domain), authors explain that when there is no classifier that performs well on both source and target domains, we cannot expect to find a good target model by training only on the source domain. Nevertheless, a study by Huh et al. [21] have already suggested that "most changes in the choice of pre-training data long thought to be critical, do not significantly affect transfer performance", which implies that the domains for transfer learning can be different as long as the tasks share commonalities. In

our case, we seek to learn features that help to discriminate between two (code) images using transfer learning from ImageNet [30, 31] pre-trained model. Thus the learned generic features from ImageNet diverse dataset of images constitute a good starting point. In the case of computer vision problems, we have the intuition that certain low-level features, such as edges, shapes, corners and intensity, can be shared between domains, thus allowing knowledge transfer between. Our empirical experiments further confirm that this choice is reasonable.

2.2 Related work

Our work is related to various research directions in the literature, including code visualization, code clone detection, computer vision, machine learning and software engineering benchmarking.

2.2.1 Code clone identification. Although code clone identification has been largely studied in the literature, relatively few techniques have explicitly targeted semantically similar code fragments. Most approaches indeed focus on textually, structurally or syntactically similar code fragments. The state-of-the-art techniques on static detection of code clones leverage various intermediate representations to compute code similarity. Token-based [6, 27, 36] representations are used in approaches that target syntactic similarity. AST-based [7, 23] representations are employed in approaches that detect similar but potentially structurally different code fragments. Finally, (program dependency) graph-based [29, 39] representations are used in detecting clones where statements may be intertwined with each other. Although similar code fragments identified by all these approaches usually have similar behavior, such static approaches still miss finding such fragments which have similar behavior even if their code is dissimilar [25].

To find similarly behaving code fragments, researchers have relied upon dynamic or concolic code similarity detection which consists in identifying programs that yield similar outputs for the same inputs [24, 28, 32, 34, 57]. Although these approaches can be very effective in finding semantic code clones, dynamic execution of code is not scalable and implies several limitations for practical usage (e.g., the need of exhaustive test cases to ensure confidence in behavioral equivalence).

Conceptually, the closest related work is by Ragkhitwetsagul et al. [53] who developed a syntactic code clone detection approach based on a visual representation of code. In order to visually represent the code, they pre-process the code by removing comments and normalizing the code formatting. The code is then rendered while applying a syntax highlighting, as done in an IDE, to create the code image. This image is then post-processed by applying various simple image transformations, such as blurring, to finally measure the resulting image similarity. The decision of whether or not two code snippets are clones is then based on the level of image similarity between the visual layout. The scope of such an approach is only limited to syntactic clones. Nevertheless, their experiments also show that the visual representation-based method can generally keep up with the state-of-the-art for syntactic clone detection. Although our approach shares the core concept of *visualizing code*, we have a different scope (semantic clones in our case) and we additionally augment this visual representation through transfer learning.

Recently, researchers have investigated leveraging advanced natural language processing and deep learning techniques to statically detect harder-to-detect clones (i.e., type-4 clones). Kim et al. [17] proposed the FaCoY code-to-code search engine where tokens from input code fragments are alternated by considering code fragments from related stackoverflow posts. This enables the search engine to identify syntactically dissimilar code fragments from the search database. This work, however, is rather competitive to online code search engines than code clone detectors. With their Oreo framework, Saini et al. [55] have proposed to use a combination of machine learning, information retrieval, and

software metrics to deal with all clone types. They build a specific deep neural network with siamese architecture to address type-4 clones with relative success.

2.2.2 Vulnerable code prediction. Deep learning has already been successfully used by researchers to automatically detect vulnerability. Li et al. [37] have proposed the first semantic framework to detect vulnerabilities using deep learning. Entitled "Syntax-based, Semantics-based, and Vector Representations (SySeVR)", the framework uses program representations that lead to syntax and semantic information pertinent to identify vulnerable code. The approach follows the notion of region proposal in image processing which, transferred to the software domain, divides a program into smaller pieces of code. In order to characterize vulnerabilities, authors introduced the concept of Syntax-based Vulnerability Candidates (SyVCs) and Semantics-based Vulnerability Candidates (SeVCs). Unfortunately, the definition of SyVC and SeVC, which is made from manual observations of most vulnerabilities, does not a priori cover all variants of vulnerable code. Li et al. [35] propose another deep learning based method for automated and intelligent vulnerability prediction in source code. Their approach, more broadly, is based on a minimum intermediate representation learning of source code.

2.2.3 Code visualization. Prior work has already investigated image representation of code for software engineering tasks. Chen et al. [11] indeed propose to convert code characters into pixels whose color values will be decided based on the ASCII decimal value of the associated characters. The resulting pixels are then arranged in a matrix, thus obtaining code images. In contrast, our representation is a simple and straightforward screenshot of either the code (displayed as is or its AST view).

2.2.4 Semantic representation learning. Deep learning advances have been exploited for statically learning semantic representations of code. A prominent work in this direction is the Tree-based convolutional neural network (TBCNN) proposed by Mou et al. [46]. The authors proposed an effective embedding method for programming language processing, and introduced a large dataset of functional clones which is necessary to train and evaluate the task of code classification. More recently, Zhang et al. [62] set the new state-of-the-art representation learning approach with ASTNN, which was demonstrated to be more effective than TBCNN for code clone identification tasks. ASTNN is a semantic embedding method which splits a given code AST into a sequence of smaller statement subtrees and applies a word2vec [44, 45] embedding to those subtrees. This way ASTNN manages to capture both the lexical and syntactical information within code fragments. We consider both ASTNN and TBCNN as the state-of-the-art for semantic clone identification, and thus they will be used as references for benchmarking our WySiWiM approach.

2.2.5 ResNets. In the literature, the Deep Residual Networks [20] are among the best performing models for image classification. The end-to-end deep learning benchmark and competition, DAWNBench [13] ranks first the ResNet models ahead of several other approaches on the basis of the imageNet dataset. This neural network architecture allows to create deeper neural networks for image classification while reducing the network complexity in comparison to other deep learning techniques. Experimental data confirmed that the strategy is effective and may lead to human-level accuracy for the task of image classification. Our approach builds on the success of these networks.

2.2.6 Benchmarks. In the code clone identification literature, two main benchmarks are widely used.

- BigCloneBench (BCB), released by Svajlenko et al. [58], is the first big-data-curated benchmark of real clones. It is widely explored in the literature for assessing state of the art clone detection tools. BCB contains 8 million clone pairs and is to the best of our knowledge the biggest publicly available Java code clone benchmark. It was built by

labeling pairs of code fragments from the IJaDataset-2.0 [4]. These pairs of clones in the dataset implement the same functionality and are divided into the 4 types of clones : type-1, type-2, type-3 and type-4. Particularly for type-3 and type-4, authors distinguish them following their syntactic similarity. However, there is no consensus on the minimum similarity of a type-3 clone, so it is difficult to separate type-3 and type 4 clones. Thus, in this paper we maintain the following proposal of the BCB authors: Types 3 and 4 are divided into three categories according to their syntactic similarity values: "strongly type-3" for a similarity in the range [0.7, 1.0], "moderately type-3" corresponding to similarity in the range [0.5, 0.7), and "weakly type-3+4" when the similarity is in the range [0.0, 0.5). Table 1 shows the distribution of the types of clones contained in the dataset.

- OpenJudge (OJ), released by Mou et al. [46], is another public dataset used to evaluate code-clone detection. It is mostly used in the literature for evaluating program classification approaches, although recent works [55, 62] have applied code clone detection approaches to it. The dataset consists of solutions submitted by students to 104 programming questions on OpenJudge[2], written in C. For each question, there are 500 corresponding solutions, each of which is verified to be correct by OpenJudge and are thus considered as clones.

Table 1. the distribution of clone types contained in BigCloneBench.

Clone Type	Number of samples
Type-1	48,116
Type-2	4,234
Strongly Type-3	175,743
Moderately Type-3	2,535,847
Weakly Type-3+4	5,820,213
All	8,584,153

3 WYSIWIM

In this section, we will overview the design of WySiWiM, providing details on the considered visualizations and the learning models.

3.1 Approach overview

The core of the WySiWiM approach is about the production of embeddings for a given code fragment. The idea is to take a code fragment and produce a vector of real numbers so that we receive an actionable representation of the embedded semantic information. As illustrated in Figure 1, we consider a deep feature extractor which works by producing embeddings for image renderings of code fragments.

Building a deep feature extractor requires a training step based on a large dataset of images. During such a training, the neural networks learn suitable representations for the images within a feature space. Given that deep neural network architectures for image classification are known to capture a large number of structural features of images, we postulate that pre-trained models can be explored in a transfer learning scenario (cf. Section 3.2). Transferring the knowledge, embedded in those pre-trained models, allows us to extract visual features without the need of huge amounts of task-specific data to train the feature extractor.

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

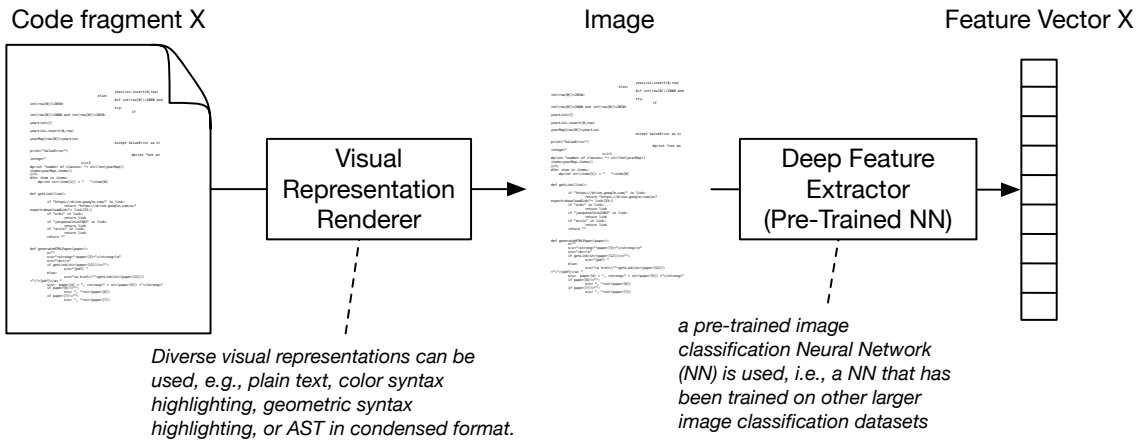


Fig. 1. Deep feature extraction (a.k.a, visualization-based code embedding)

Once the feature extractor is obtained, one can feed code rendered as images into it to collect the resulting feature vectors. Those can further be used to train simple binary classifiers that learn to apply the embedded semantic information. Simply put, the deep neural network is used to preprocess images so that they can be used to learn semantics by applying well-known classical machine learning algorithms.

Clone identification tasks. In this work we apply the WySiWiM approach of visualization-based code semantics learning to the problem of clone identification, which is approached in two different ways: as a classical *code clone detection* problem and as a *code classification* problem.

⋆ In code classification, the goal is to predict the functionality implemented by a code fragment. In practice, we must learn to map the code fragment to one of a set of predefined semantic functionality labels (i.e. *clone classes*). It is thus a multi-class classification problem that takes a single code fragment as input and outputs a functionality label.

⋆ In clone detection, the goal is to directly decide if two code fragments are clones. It is thus a binary classification problem that takes a pair of code fragments as input and outputs a Yes/No label on whether or not those fragments form a clone pair.

In principle, both tasks can be emulated by one another. On the one hand, the code classification task could be emulated by finding all clone pairs and building their transitive closure to generate the semantic clone classes. On the other hand, the clone detection task, could be emulated by directly comparing the code fragment labels. We have nevertheless opted in this work to build two separate workflows, both starting by first converting code fragments into their visual representations.

⋆ For code classification, the collected code "images" and their associated functionality labels are used to **fine-tune a pre-trained image classification network**. To that end, the **size of the output layer** of the pre-trained image classification network must be updated. Indeed the output layer nodes map to the classes that are seen during training. With new datasets, new classes appear.

⋆ For clone detection, the collected code "images" are **directly fed into a pre-trained image classification network in order to retrieve the corresponding embeddings** (which are numerical vectors representing the internal

structural features within images). Obtained feature vectors are then used for training and testing a classical binary classifier.

Vulnerable code prediction task. In this work we also apply WySiWiM to the problem of predicting vulnerabilities. We consider the task as a simple binary classification problem where the goal is to predict whether a code fragment of code provided as input is vulnerable or not.

3.2 Transfer learning from pre-trained models

In our approach, we transfer the embedded knowledge of the pre-trained image classification neural networks to our clone identification tasks (i.e., for both code classification and clone detection). The knowledge that is transferred in our case is *the ability to recognize visual patterns and structures from images*. Even though the data that those networks are trained on belong technically to a different domain, we expect that they still capture relevant structural knowledge that can be reused to extract the structural information from our specialized (code visualization) images. Thus, our hypothesis here is that, through the transfer learning, we can leverage powerful pre-trained networks which are able to effectively embed meaningful syntactic as well as semantic structures [20].

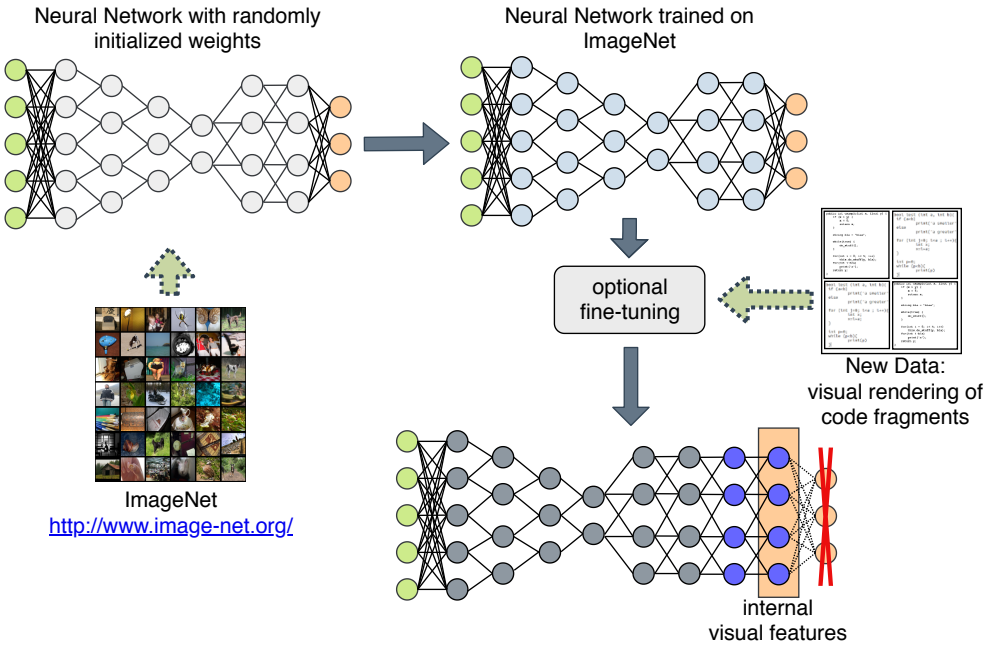


Fig. 2. Principle of transfer learning applied to build our deep feature extractor for code

Image classification neural networks consist of a multitude of convolutional layers that all learn different combinations and variations of the data contained in the previous layers. In addition, the networks have an input layer which accepts the input data and a fully connected output layer. This final layer is usually sized according to the number of possible labels and is in charge of deciding a label for the data coming from the previous layers. In our case, as depicted in Figure 2, we focus on retrieving the intermediate features that are accessible in the penultimate layer. Actually, these

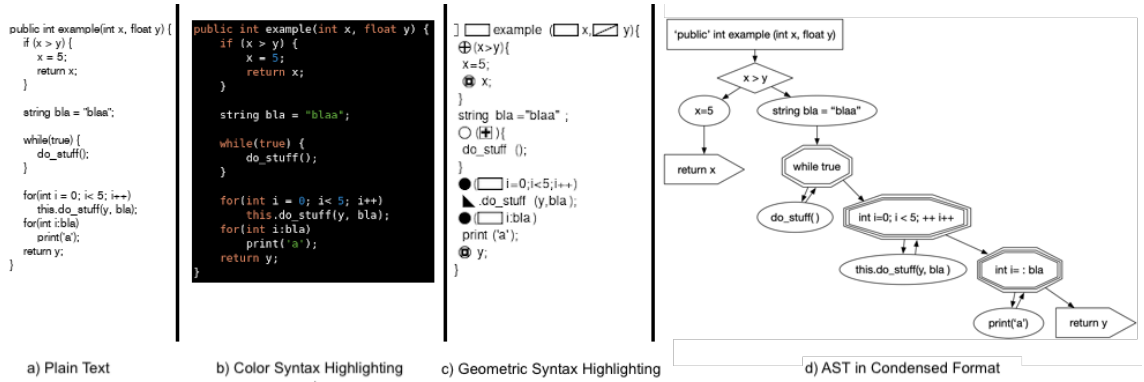


Fig. 3. Variations of visual representations of code

features could have been collected on any previous layers. For the sake of prototyping speed, we immediately accessed the readily-available features. Future work could investigate other layers.

It should be noted that transfer learning is gaining traction within the deep learning community, since several domains lack sufficient data for training [49]. Therefore, a fundamental motivation in the study of transfer learning is the fact that people can intelligently apply knowledge learned previously to solve new problems faster or with better solutions [49]. For example, it has been shown possible to use the knowledge about notes and rhythm, which were learned for playing the piano, to learn guitar playing; applying the vocabulary learned in French to infer English words as they share a certain base; or in audio-visual correspondence tasks [5].

3.3 Visualization options

We explore in WySiWiM four variations of code visualizations in order to assess the influence of the selected visual representation on the performance of WySiWiM. We describe each visual representation by explaining its principle, detailing its implementation and arguing about its relevance.

• Plain **Text:** The first visual representation is straightforward. It consists of simply rendering the textual representation of the code as a black and white image without highlighting any language construct. The rendering is implemented using the *pillow*[3] Python image drawing and manipulation library: source code text is rendered as-is, i.e., with the indentations used by the developer, while applying a white background. Plain text, illustrated in Figure 3(a), is considered as our baseline visual representation of code. With Plain Text, we expect the learning algorithm to learn the visual shape of characters and words (locally) as well as the code blocks shapes (globally) to identify structures of code logic (e.g., indentation, if conditions, loops, etc.).

• Color **Syntax Highlighting:** A simple variation of the Plain text visualization consists in rendering the code text while highlighting syntax with colors, similarly to what is done in programming environments. This rendering approach is implemented by first generating an html page to highlight the code using the `google code-prettify` javascript library. The web page is then saved as a PNG image using the *imgkit*[4], a python wrapper for the Webkit web browser engine. As illustrated in Figure 3(b), this visual representation is expliciting code structures for human

[3] https://pillow.readthedocs.io
[4] https://pypi.org/project/imgkit/

programmers. Therefore, we expect that color-based syntax highlighting can be relevant for semantic machine learning tasks.

• Geometric **Syntax Highlighting:** In the previous visualization option, emphasis is put on color. Yet, image classification neural networks are also known to capture shapes. We propose to build a rendering of code where language keywords are represented by specific geometric shapes (i.e., icons). The implementation is based on the tokenization of code fragments using the *javalang*[5] python library. We preset the mapping of language keywords with specific icons. During rendering, the text tokens are then replaced by the associated icons. Overall, although this representation could be nonsensical for humans, we expect that it will support the learning algorithm in the same way colored syntax does for programmers visual perception of code.

• Ast **in Condensed Format:** Finally, we consider a visual rendering of the abstract syntax trees. The implementation is based on the AST generated by the *javalang* python library and leveraging *graphviz* python bindings[6]. To render the resulting graph, we generate a "graphviz" graph model by traversing the AST and representing some subtrees (e.g., the "for" loop control) in a purely textual manner, while representing other elements as their actual tree structure. This helps to condense the AST since raw AST quickly explodes in depth and breadth even for small code fragments. In this representation we generated the graph such that the edges represent the possible control flows inside the code in order to capture its sequential nature. Further, we apply some geometric shapes to specific types of nodes in order to augment the visual strength of specific code structures.

Overall, we try in WySiWiM visualization options that emphasize on colors, shapes and structures, and compare against the baseline plain text rendering. Although the generation of visual renderings is stable (i.e., not a random process), it should be noted that the AST in condensed format is, by far, the slowest to compute, as it involves many complex steps.

Concretely, the output of the visualization rendering process is a single PNG image per visualization option and per code fragment. Each image may also be re-scaled to fit with the input requirements of the pre-trained image classification neural network.

3.4 Code classification architecture

Figure 4 provides a simple illustration of the overall architecture that we developed for code classification. We leverage neural networks (specifically, the powerful ResNets) that are pre-trained on the ImageNet dataset [16]. However, we perform a re-training step, which is actually aimed to fine-tune the neural networks, towards better learning to extract features that are semantically-relevant to different classes of code functionalities. To that end, we update the size of the output layer of the pre-trained image classification network so that the final size accounts also for the number of possible functionality labels in our code dataset. The re-trained (i.e., fine-tuned) network on the training dataset is then used as a classifier to predict the labels of code fragments in the test set based on their visual renderings. In order to ensure the stability of the re-training process and to provide some guarantees in the yielded results, we have conducted the same experiments thrice. Each retraining was performed over 20 epochs. Details on how training and test sets are split are provided and discussed later in Section 4.2.

[5] https://pypi.org/project/javalang/
[6] https://www.graphviz.org/

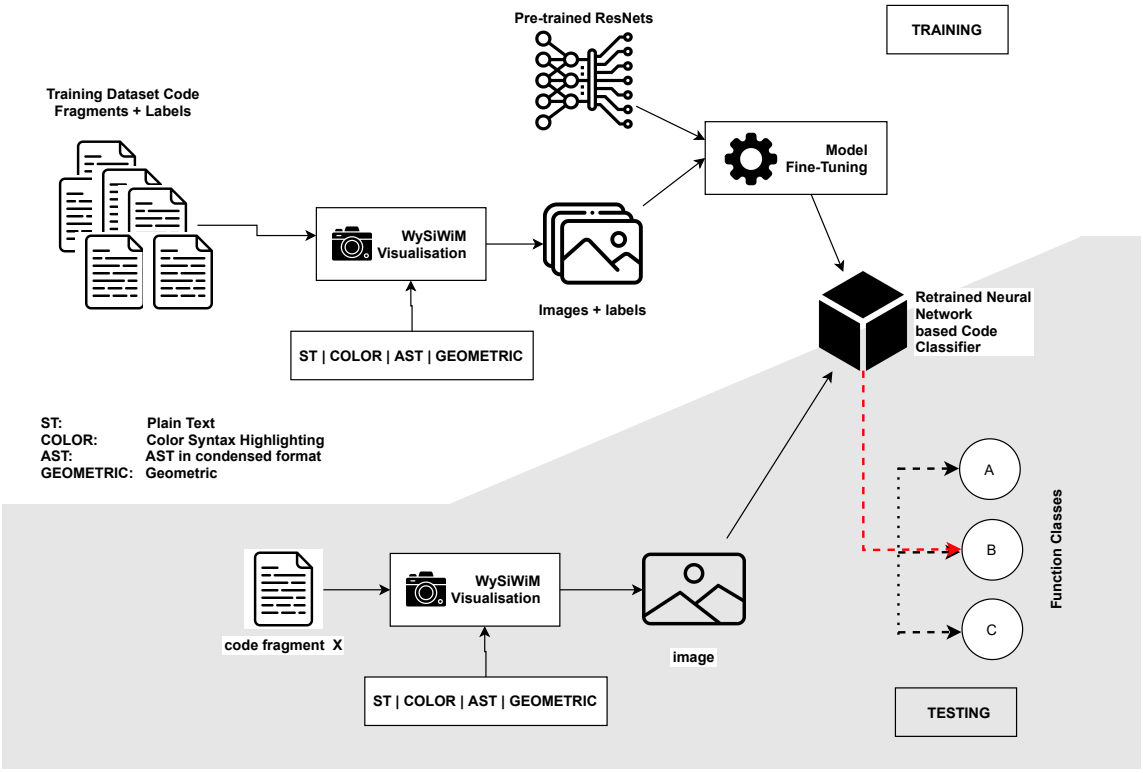


Fig. 4. Illustration of the architecture for WySiWiM's Code classification

3.5 Vulnerable code prediction architecture

Our vulnerability prediction task is a code classification task (hence with the similar architecture as for code classification) where the ouput is a binary class related to the presence of a vulnerability. Thus, as with the code classification pipeline, we use a pre-trained neural network to classify images obtained from different visual representations of code. Again, we use the ImageNet pre-trained model over resnet18 and resnet50 architectures. This architecture is refined as a binary classification where we predict whether or not a code fragment is vulnerable. Figure 5 provide an illustration of this architecture. Similarly to code classification architecture, we do not tune any hyper-parameters of these networks and we do not need a validation set. In order to re-train and evaluate WYSiWiM on vulnerability code prediction, we collected, process and split data for training and testing. Details on the dataset are provide in the next section.

3.6 Clone detection architecture

Figure 6 illustrates WySiWiM's architecture for clone detection. Similarly to the pipeline of code classification, we leverage a pre-trained neural network for visual classification to which we feed the images obtained from visual renderings of code snippets. In this case, however, our objective is to simply collect the embeddings produced during deep feature extraction. Thus, given that we do not need the network to learn about new classes in our new (code-related) image datasets, we propose to directly use the ResNets that were pre-trained on ImageNet datasets. We expect the

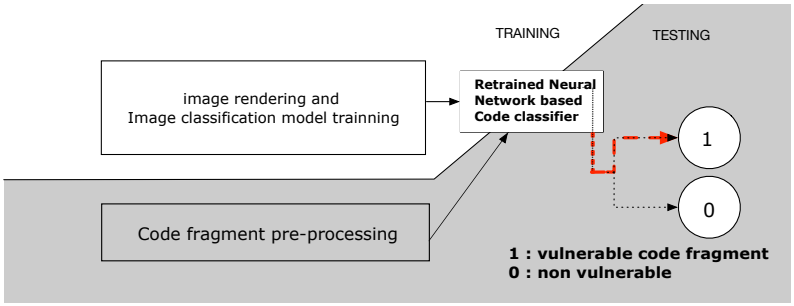


Fig. 5. Illustration of the architecture for WySiWiM's vulnerability prediction

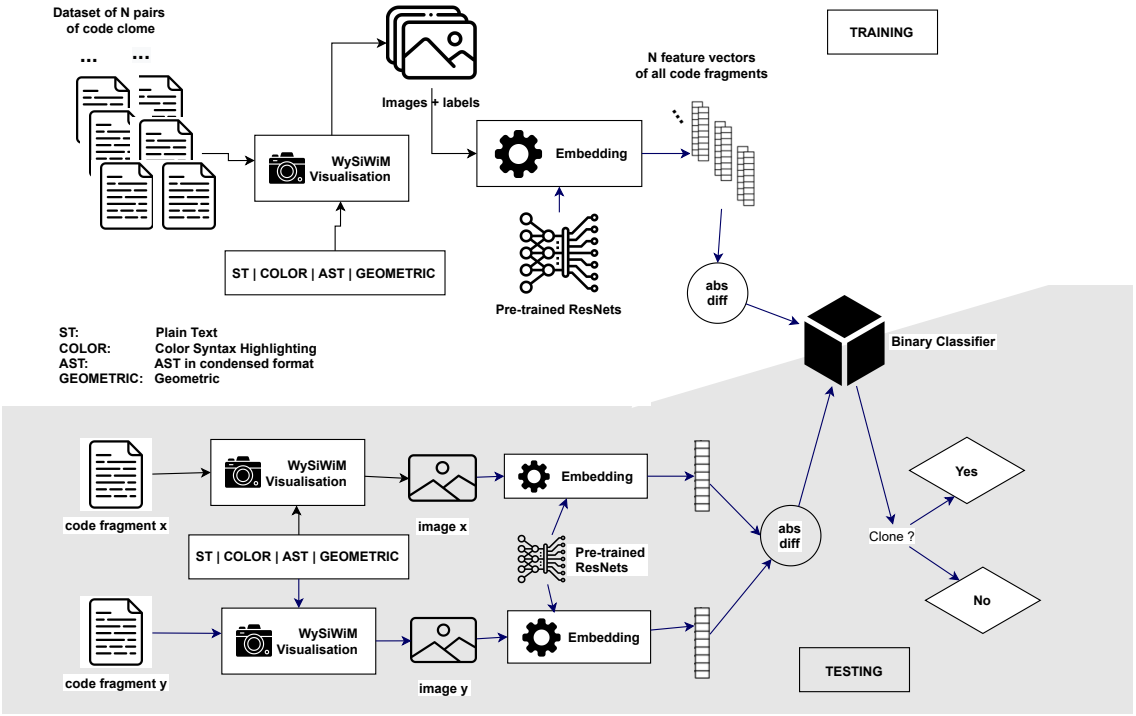


Fig. 6. Illustration of the architecture for WySiWiM's Clone detection

embeddings to still be relevant for capturing structural features. The feature vectors (i.e., embeddings) are then used to train binary classifiers as in traditional machine learning. Concretely, to train the binary classifiers, the first step is to calculate the absolute difference between the feature vectors of the candidate pair vectors. Those difference vectors can then be used to train our binary classifiers.

Algorithms for binary classification. In code classification, we directly reuse the in-built capability of the neural networks to perform classification (i.e., using the *softmax* activation function at the last layer). Indeed, given that the input of the task is a single image representing the visualization rendering of a code fragment, the classical image classification neural network is suitable.

In clone detection, however, the input is a pair of code fragments (precisely, a pair of images taken from their visualization renderings). This means that the architecture of image classification networks is not readily applicable for this case as it always expects a single input only. For sake of simplicity and optimization, we decided to use the neural network to collect embeddings for individual images, and train our final classifier separately. This strategy allows us to experiment with different traditional classification algorithms. Our experiments provide results with Support Vector Machines [14], k-Nearest Neighbours [15] and a simple binary classification neural network [19].

4 EXPERIMENTAL SETUP

We enumerate the research questions, overview the datasets used in the experiments and discuss some important implementation details. We open-source the implementation of our prototype implementation of WySiWiM and release all data related to the experiments recorded in this paper. The artifact web page is currently in an anonymous repository: https://github.com/wysiwim/wysiwim

4.1 Research Questions

RQ1: How does WySiWiM perform in comparison with the state-of-the-art? We investigate the ability of our novel approach of semantics learning based on visual representation of code to keep up with the state-of-the-art for the tasks of code classification, clone detection and vulnerable code prediction.

RQ2: How does the visual representation influence the performance of WySiWiM? Experiments for this research question are focused on the code clone detection task, where we try all the considered visual representations options and compare the performance differences. We also conducted similar (but less extensive) experiments for the task of vulnerable code prediction to further assess the learning power of different representations.

RQ3: What is the impact of the classification algorithms on WySiWiM? We investigate in this research question different supervised learning algorithms that can be leveraged to train the binary classifiers needed for the code clone detection architecture.

4.2 Selection of datasets

Table 2 summarizes all the datasets used in this paper. We will now detail each of these datasets.

Table 2. Summary of all the datasets

Task Name	Dataset Name	# of samples	
		Total	Used in Evaluation
Vulnerable Code Prediction	The KB Project	1,240	248
	SySeVR dataset (based on NVD and SARD data)	420,627	84,126

Code Classification: We assess the performance of WySiWiM for the code classification task based on the Open Judge (OJ) dataset as introduced in [46]. This choice is motivated by the need to directly compare against the state-of-the-art (namely, TBCNN [46] and ASTNN [62]), which also run experiments on this dataset. This dataset contains 104 different functionalities and 500 samples per functionality. In order to achieve balanced datasets for training and testing, we apply a stratified sampling over the functionalities with a ratio of 4:1 (i.e, 80% of data for training and 20% for testing).

Vulnerable code prediction: We leveraged two (02) different datasets for the vulnerability prediction experiments.

The KB Project. We collected our first dataset from the KB project [52] which supports the creation, management and aggregation of a distributed, collaborative knowledge base of vulnerabilities that affect open-source software. Thus, we successfully constructed a dataset containing 1,240 code samples (i.e., 620 pairs of vulnerable and non-vulnerable methods). The vulnerable code samples are spread over 167 Common Vulnerabilities and Exposures (CVE). Due to the important number of CVE, we mapped each CVE with its corresponding Common Weakness Enumeration (CWE) ID. The distribution of samples within the CWE categories is provided in Table 3. To achieve balanced datasets for training and testing, we apply a stratified sampling over the labels and CWE IDs with a ratio of 4:1 (i.e, 80% of data for training and 20% for testing).

NVD and SARD. In order to be able to compare ourselves directly with the state of the art in vulnerable code prediction "Syntax-based, Semantics-based, and Vector Representations (SySeVR)" [37], we used the same dataset as SySeVR's authors. The dataset is consisting of samples collected from two different sources : The National Vulnerability Database (NVD) [47] and the Software Assurance Reference Dataset (SARD) [48]. NVD contains vulnerabilities in production software. Some of the samples contain includes diff files that describe the difference between the vulnerable code and the correct code. This way we can have vulnerable and non-vulnerable data available. SARD contains vulnerabilities in production, synthetic and academic software. The samples in this dataset are categorised as "good" (i.e. having no vulnerabilities), "bad" (i.e. having vulnerabilities) and "mixed" (i.e. having vulnerabilities for which corrected versions are also available). In order to construct our dataset for experimentation, we have focused on the same programs and open source projects as in SyseVR [37]. In total, therefore, we have a set of 420,627 code fragments, 364,232 of which are vulnerable and distributed over 107 types (CWE [59] - Common Weakness Enumeration ID) of vulnerabilities. In addition, SySeVR's [37] authors found that the dataset exposes 4 syntactic characteristics of the vulnerability that are: *Library/API Function Call*, *Array Usage*, *Pointer Usage* and *Arithmetic Expression* and for some experiments we focus on this because of the rather high number of CWEs. Table 4 shows the distribution of the dataset according to the 4 syntactic characteristics.

Code Clone Detection: The state-of-the-art for code clone detection being ASTNN [62], we reuse the dataset that they release in their experiment artifacts. This enables a direct and unbiased comparison. The split into training and testing sets is also predefined and applied as-is. This dataset consists of 20k Type-4 clone pairs and 20k non-clone pairs.

Nevertheless, we found that the ASTNN dataset is not balanced with respect to the number of clones per functionality. Thus we selected a custom subset of BigCloneBench (BCB) (cf. Section 2.2.6) for our further experiments. We focus on code fragments related to three functionalities (i.e., #7 - `bubble-sort array` #13 - `shuffle array inplace` and # 44 - `check for palindrome`) which we consider the most suitable for our evaluation: these code fragments are dissimilar enough but concise; furthermore #7 and #13 code fragments deal all with arrays and yet semantically distant, offering an

Table 3. The dataset from KB Project: CWE IDs and description

CWE IDs	Description	# of samples
CWE-297	Improper Validation of Certificate with Host Mismatch	11
CWE-352	Cross-Site Request Forgery (CSRF)	40
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	74
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	65
CWE-20	Improper Input Validation	119
CWE-94	Improper Control of Generation of Code ('Code Injection')	27
CWE-287	Improper Authentication	34
CWE-306	Missing Authentication for Critical Function	21
CWE-502	Deserialization of Untrusted Data	68
NVD-CWE-noinfo	Insufficient Information	88
CWE-264	Permissions Privileges and Access Controls	73

Table 4. The dataset from NVD and SARD: The distribution of the dataset according to syntactic characteristics [37]

Syntactic Characteristics	# of vulnerable samples	# of correct samples
Pointer usage	28,391	263,450
Array usage	10,926	31,303
Arithmetic expression	3,475	18,679
API function call	13,603	50,800
All	56,395	364,232

opportunity to properly assess the semantic clone detection approach. The dataset is constructed by randomly sampling 500 Type-4 clone pairs and 500 non-clone pairs per functionality. The ground truth information of clone/non-clone is based on the annotations provided in BigCloneBench.

4.3 Implementation

Our proof-of-concept implementation of WySiWiM is written in Python using common frameworks and libraries. In particular, several Python libraries are leveraged for the code fragment processing towards producing visual renderings as images (cf. Section 3.3). We leverage the *pandas*[7] library for data management. Further for the stratified splitting of the datasets, we use the dataset splitting method from the *scikit-learn*[8] library.

Data Preprocessing. The pre-trained networks considered in our experiments have a limitation on the input image size being set to exactly 224x224 pixels. To fit with this requirement, we choose to simply re-scale the code visualizations to this size.

Data Augmentation. Usually in many machine learning applications, a data augmentation step is performed. In image classification in particular, one usually uses a set of random transformations to create many variations from the input data in order to artificially increase the size of the dataset. Those random transformations include rescaling,

[7] https://pandas.pydata.org/
[8] https://scikit-learn.org/

cropping, mirroring etc. For our approach however, we found that this is not beneficial since source code naturally does not appear up-side down or mirrored.

Code classification. The implementation of our code classification task is mainly based on PyTorch [50] and uses the pre-trained ResNet models provided by the PyTorch framework. In particular we use a ResNet18 and a ResNet50 to highlight the increase of performance when the number of layers is increased. Both are pre-trained on the ImageNet dataset [16].

Vulnerable code prediction: The implementation of our vulnerable code prediction task is also mainly based on PyTorch [50] and uses the pre-trained ResNet models provided by the PyTorch framework. In particular we use a ResNet18 and a ResNet50 to highlight the increase of performance when the number of layers is increased. Both are pre-trained on the ImageNet dataset [16].

Experiment with KB Dataset the dataset for these experiments is made of Java code, leading to the following constraint for Geometric visual rendering : we did not have enough geometric figures to represent each of the keywords of the language.

Experiment with SySeVR Dataset With this experiment, we wish to compare WySiWiM results directly with the state-of-art that use the same dataset. Moreover, we found that in this dataset samples whose code fragments do not give enough context in order to have their corresponding AST and generate errors. Instead of removing those problematic sample, we conduct this experiment without the Ast in condensed format visualization. method.

Clone detection. For the implementation of the binary clone detection task, we use again the pre-trained ResNet50 model and drop the last layer in order to generate the raw feature vectors for our visualized code fragments. Those vectors are then converted into *numpy*[9] arrays which facilitates the calculation of the absolute difference between vectors. For the final stage of learning and predicting, we use pytorch again to implement a simple binary classification network. SVM and k-NN algorithm implementations are taken from the *scikit-learn* library and concerns the different variants of the algorithms [51]. Thus, we SVM is used with different kernels (*linear*, *polynomial*, *sigmoid* and *rbf* known as Gaussian kernel) and k-NN with different algorithms used to compute the nearest neighbors (*ball_tree*, *kd_tree* and *brute*).

5 RESULTS

We now present the experimental results in response to the research questions, and based on the experimental settings presented previously.

5.1 RQ1: [Performance of WySiWiM]

5.1.1 Code classification. For performance comparison against the state-of-the-art for the task of code classification, we focus on the **accuracy** metric, which is used by the state-of-the-art ASTNN and TBCNN authors to report their performance (see. [46, 62]). As discussed previously, we also reuse the same OJ dataset that was used for ASTNN and TBCNN validation. We conducted these experiments using all WySiWiM's visualisation methods.

Results: Table 2 shows the results of all the experiments on code classification. To compare with TBCNN and ASTNN, we report in the table 6 our best performance (the accuracy values) according to the two (02) classification models used. Thus WySiWiM provides an accuracy of 89.7 and 86.4 percent for code classification on the OJ dataset

[9] https://www.numpy.org/

Visualization method	Model	Accuracy	Precision	Recall	F1 score
Plain Text	ResNet18	86.4	85.8	84.7	82.2
	ResNet50	89.7	85.7	87.4	86.3
Color Syntax Highlighting	ResNet18	79.8	80.2	79.8	79.8
	ResNet50	87.7	87.7	87.7	87.6
Geometric syntax highlighting	ResNet18	83.9	83.9	83.9	83.8
	ResNet50	83.3	83.3	83.4	83.3
Ast condensed format	ResNet18	84.3	84.2	84.3	84.0
	ResNet50	87.9	84.6	84.2	83.9

Table 5. WySiWiM Code Classification Task: All Experimental Results

with ResNet18 and ResNet50 respectively. These results suggest that we perform reasonably well in comparison to the state-of-the-art which are reported to yield accuracy scores of 94.0% and 98.2% for TBCNN and ASTNN respectively.

Method	Variation	Accuracy
TBCNN	-	94.0
ASTNN	-	98.2
WySiWiM	with ResNet18 pre-trained model	86.4
WySiWiM	with ResNet50 pre-trained model	89.7

Table 6. Accuracy comparisons for code classification.

5.1.2 Clone Detection. Experiments for Clone detection are done with the BigCloneBench which already has labels on pairs of clones and non-clones. For fair comparisons, we run ASTNN and WySiWiM on the same samples of Type-4 clones that were used to evaluate ASTNN by Zhang et al. [62]. For each visualization method of WySiWiM, we conducted the experiments with all the selected classification algorithms and we discuss the results for our best configuration: the Color syntax highlighting as the visualization option and the *linear* SVM as classification algorithm. We refer the reader to next research questions where we show that the visualization and algorithms have a limited impact on the performance of WySiWiM. Finally, contrary to previous experiments, we do not compare against TBCNN since this approach has not been applied for clone detection.

Results: The table 7 and the Figure 7 show respectively the results of all the experiments on the code clone detection task and the confusion matrix for our best performance. We report in the table 8 the performance of our best experiment (achieve with the the Color syntax highlighting visualization and the *linear* SVM algorithm) and provide a direct comparison with ASTNN, the state-of-the-art approach in the code clone detection task. This performance comparison show that, overall, we outperform the state-of-the-art in terms of F-Measure. It is further noteworthy that WySiWiM offers a better trade-off between precision and recall than ASTNN which presents quasi-perfect precision but lower recall.

5.1.3 Vulnerable code prediction. In order to answer to our first research question on WySiWiM performance in vulnerable code prediction task, we compute its prediction performance on two (02) different datasets (cf. Section 4.2): The KB project dataset and the SySeVR dataset built from NVD [47] and SARD [48] data. For each of these datasets, we conducted experiments with the Resnet18 and ResNet50 models, and using WySiWiM's visualization methods. Due

Visualization method	Classifier	Variation	Accuracy	Precision	Recall	F1 score
Plain Text	NN	-	90.4	92.2	88.6	0.90.3
	SVM	linear	97.2	96.9	96.9	96.8
		poly	93.2	90.3	**97.8**	94.0
		rbf	94.7	94.0	96.3	95.2
		sigmoid	68.3	70.4	72.3	71.3
	kNN	brute	95.8	97.2	94.9	96.0
		kd_tree	95.8	97.2	94.9	96.0
		ball_tree	95.8	97.2	94.9	96.0
Color Syntax Highlighting	NN	-	95.4	95.9	95.9	95.8
	SVM	linear	**97.9**	**98.6**	95.6	**97.1**
		poly	92.8	89.3	98.7	93.7
		rbf	95.9	97.2	95.3	96.6
		sigmoid	58.6	62.4	60.5	61.4
	kNN	brute	94.3	95.9	93.6	94.7
		kd_tree	94.3	95.9	93.6	94.7
		ball_tree	94.3	95.9	93.6	94.7
Geometric syntax highlighting	NN	-	88.1	86.3	91.1	88.6
	SVM	linear	96.5	96.9	96.6	96.8
		poly	93.2	90.3	97.9	94.0
		rbf	94.7	94.1	96.3	95.2
		sigmoid	68.3	70.4	72.3	71.3
	kNN	brute	91.6	91.4	93.2	92.3
		kd_tree	91.6	91.4	93.2	92.3
		ball_tree	91.6	91.4	93.2	92.3
Ast condensed format	NN	-	92.7	96.7	93.9	95.3
	SVM	linear	97.8	97.7	97.6	**97.1**
		poly	95.4	95.6	95.9	95.9
		rbf	95.6	95.9	95.9	95.9
		sigmoid	67.6	71.9	66.6	69.1
	kNN	brute	92.8	95.7	90.9	93.2
		kd_tree	92.8	95.7	90.9	93.2
		ball_tree	92.8	95.7	90.9	93.2

Table 7. WySiWiM Code Clone Detection Task: All Experimental Results

Method	F1 score	Precision	Recall
ASTNN	93.7	99.8	88.3
WySiWiM	97.1	98.6	95.6

Table 8. Performance comparison for clone detection.

to errors during the generation of ASTs for some samples in the SySeVR dataset, we have not considered the AST in condensed format visualization method for this dataset.

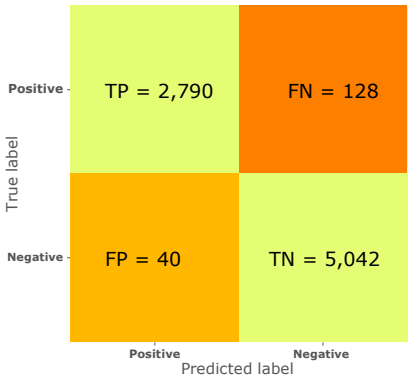


Fig. 7. Confusion Matrix: Code Clone Detection

Dataset	Visualization Method	Model	Accuracy	Precision	Recall	F1 score
KB Project	Plain Text	ResNet18	55.6	55.7	55.7	55.6
		ResNet50	62.5	62.8	62.5	62.2
	Color Syntax Highlighting	ResNet18	67.4	66.3	65.2	64.6
		ResNet50	**71.7**	**70.7**	**71.7**	**71.6**
	Geometric syntax highlighting	ResNet18	56.0	56.4	56.0	55.4
		ResNet50	62.1	62.2	62.1	62.0
	Ast condensed format	ResNet18	55.6	55.9	55.6	55.2
		ResNet50	56.4	56.4	56.4	56.4
SySeVR	Plain Text	ResNet18	88.1	88.3	88.3	88.1
		ResNet50	88.8	88.8	88.9	88.8
	Color Syntax Highlighting	ResNet18	88.6	88.6	88.7	88.6
		ResNet50	**90.9**	**90.**9	**91.0**	**90.9**
	Geometric syntax highlighting	ResNet18	56.0	56.4	56.0	55.4
		ResNet50	62.1	62.2	62.1	62.0

Table 9. WySiWiM Vulnerable Code Prediction Task: All Experimental Results

	Approach	Accuracy	Precision	Recall	F1 score
SySeVR	-	98.0	90.8	-	92.6
WySiWiM	with ResNet18 pre-trained model	88.6	88.6	88.7	88.6
WySiWiM	with ResNet50 pre-trained model	90.9	90.9	91.0	90.9
Checkmarx	On a different dataset [37]	72.9	30.9	-	36.1

Table 10. Performance Comparison for Vulnerable Code Prediction (with SySeVR Dataset)

Results: The table 9 shows all WySiWiM experimental results on the vulnerable code prediction. Figure 8 illustrates the confusion matrix of our best performance on the SySeVR dataset and obtained with the resnet50 model. In table 10, we summarize the results for the task of vulnerability prediction by WySiWiM and state of the art tools. With respect to Precision and F1 scores, WySiWiM and SySeVR yield very close results (90.9 vs 90.8 for Precision and 90.9 vs 92.6

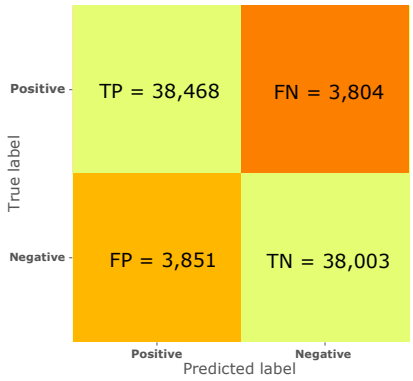


Fig. 8. Confusion Matrix: Vulnerable Code Prediction

for F1). Furthermore SySeVR's performance has not been analysed thoroughly by the authors to clear any doubts on overfitting. We have made an analysis of the dataset and found a huge imbalance in vulnerability syntactic characteristic (cf. Table 4), which may significantly bias performance result computation. We also compare against the results reported in the literature for the commercial Checkmarx tool. Although the datasets used for evaluation are different, we include this comparison as done by the SySeVR state of the art work. The results show that WySiWiM significantly outperforms Checkmarx in detecting vulnerable code.

In the light of SySeVR's authors findings on the dataset (cf. Section 4.2), we present in Figure 9 the percentage of correct predictions per vulnerability syntactic characteristic (*API Function Call*, *Array Usage*, *Pointer Usage* and *Arithmetic Expression*) following the different visualization methods. These results show that WySiWiM predicts better and more correctly samples exhibiting *Pointer Usage* as syntactic characteristic. Then we have the samples from *API Function Call* and *Array Usage* respectively. *ArithmeticExpression* is the one with weak results. The number of samples per vulnerability syntactic characteristic (cf. Table 4) seems to be at the origin of these discrepancies in results.

We have also applied WySiWiM on a different (smaller) dataset of vulnerable code from the KB project. The experimental results (Table 9) show that the performance is less significant, although it is still on par with the performance yielded by the Checkmarx commercial tool. The performance degradation that we observe however is in line with recent studies that show that given larger networks and more data the general performance of deep learning can be improved tremendously (e.g. see GPT-2 vs. GPT-3 [43].

Given the limitations that our implementation carries (cf. Section 6.2) and the potential for improvement (cf. Section 6.3), the yielded performance results for code classification, clone detection and vulnerable code prediction of WySiWiM are largely promising. In particular, WySiWiM achieves the best results on the clone detection and vulnerability prediction tasks. We will consider these two tasks in the next sections in order to answer the research questions related to the influence of the visualisation method and the algorithms.

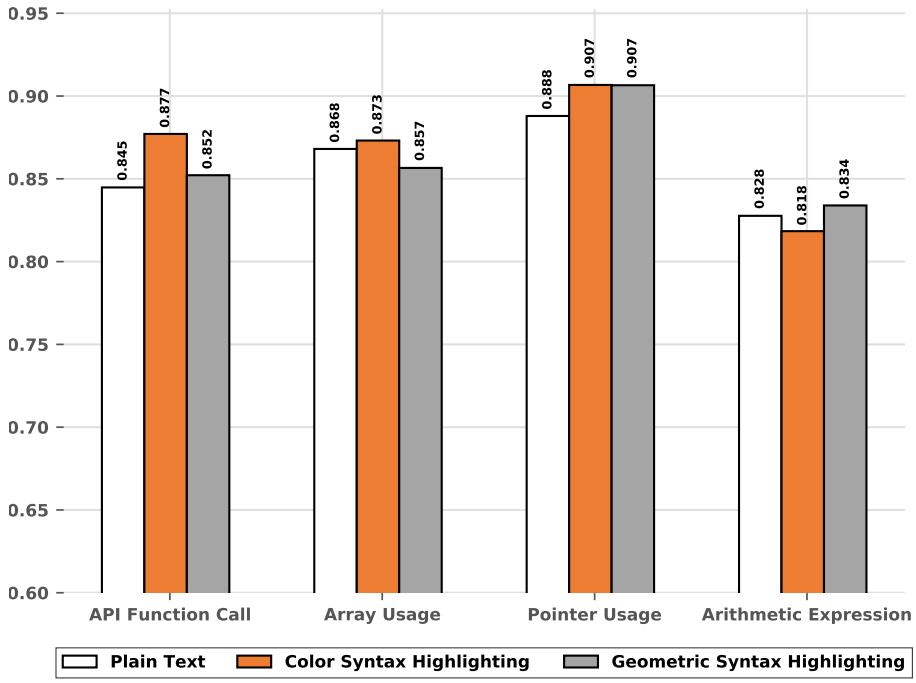


Fig. 9. Performance of WySiWiM-based vulnerable code prediction per vulnerability syntactic characteristic samples set (i.e., Percentage of vulnerability syntactic characteristic related samples accurately predicted by the learned prediction model) with ResNet50 architecture

5.2 RQ2: [Visualization influence]

5.2.1 Clone Detection. To examine the influence of visualization methods, we consider first the clone detection task where WySiWiM implements the binary neural network classifier for the final clone decision. The process is then performed for all previously-described visual representations options (cf. Section 3.3).

Results: The results depicted in figure 10 suggest that the Ast in condensed format and the Color syntax highlighting visual representations yield the best results (which are further very similar for these two representations). On the one hand, it is noteworthy that the Color syntax highlighting improves over the Plain text visualization, hence confirming our initial intuition that colors can help to better capture semantics visually. On the other hand, although Geometric syntax highlighting performs slightly less well than others, it's relatively high performance indeed suggests that visual shapes are expressive enough to help learn semantics of code structures. In any case, we also suspect that the performance degradation of Geometric syntax highlighting visualizations might emerge from a bad choice of the keyword substitution shapes. Finally, we note that, depending on the performance metric, any of the visualizations may perform better or worse than other visualizations.

5.2.2 Vulnerable code prediction. We also examine the impact of visualization methods on the vulnerability prediction task. We only focus (based on the insights of RQ1 on vulnerable code prediction) on the experiment on our

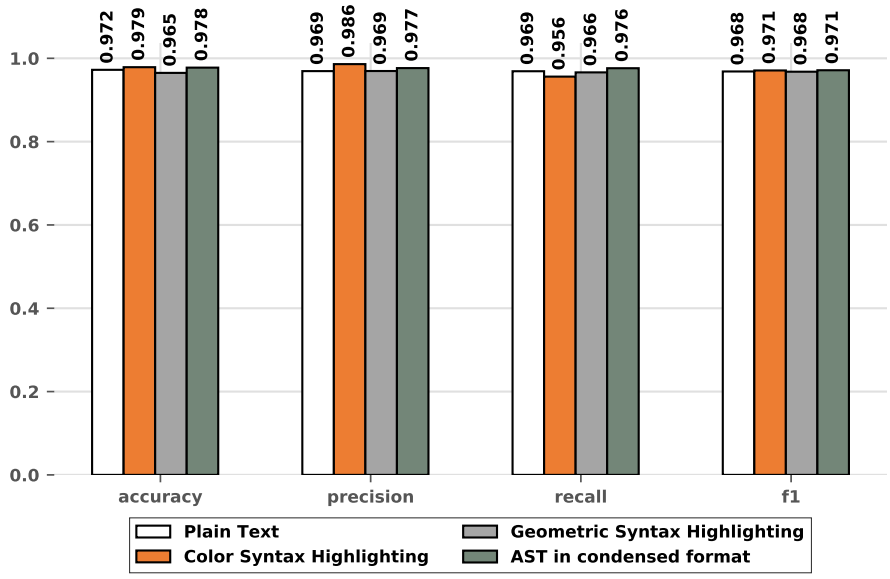


Fig. 10. Influence of visualization methods on clone detection performance. Comparison done with our best classification algorithm: SVM with *linear* kernel

SySeVR [37] dataset with the Resnet50 architecture like model. This experiment concerns the Plain text, the Color Syntax Highlighting and the Geometric as visualization methods (cf. Section 4.3).

Results: The results in figure 11 suggest that the Color syntax highlighting visual representations yield the best results. The Geometric and the Plain text visualization gave low results on this task. These results confirm for a second time our initial intuition : the Color syntax highlighting visualization help to better capture semantics of source code. Also, Just like the results on clone detection, the Geometric syntax highlighting performs slightly less well than other visualization methods.

5.3 RQ3: [Algorithm impact]

To run several experiments of clone detection while varying the classification algorithms, we leverage our main dataset sampled from BCB (cf. Section 4.2). We also fix the visualization option to the Color syntax highlighting. The experiments are then performed to compare the variations of sensitivity of the WySiWiM embeddings with respect to different algorithms. We use 3 variants of k-NN[10] (with the algorithms *ball_tree*, *kd_tree* and *brute* to compute the nearest neighbors) , a simple NN[11], and 4 variants of SVM (use of the kernels *linear*, *polynomial*, *sigmoid* and *rbf*).

Results: Figure 12 presents the comparison results. It appears that the algorithm has a slight impact on all scores between k-NN variants. Compared to the Neural Network classifier, the scores also fluctuated very little except the accuracy which is lower for the Neural Network classifier. Using the *rbf* or *linear* kernels, SVM is the algorithm with the best scores compared to the others. Besides, SVM with *sigmoid* kernel has the lowest results overall. Our intuition is that the use of the *sigmoid* kernel requires hyperparameters tuning whereas in our experiments, all other parameters are

[10] We use kNN with the default setting of scikit-learn where $k = 5$
[11] The NN is a simple fully-connected linear layer with bias, so in essence a linear combination

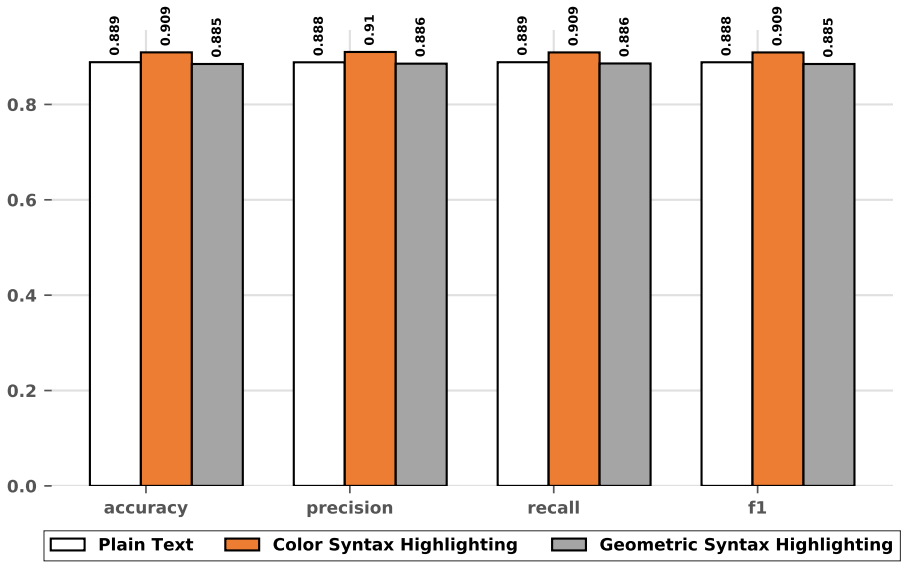


Fig. 11. Influence of visualization methods on Vulnerable code prediction performance with ResNet50 architecture

kept by default. Nevertheless, all algorithms except SVM with *sigmoid* kernel offer reasonably good performance, which suggests that the embeddings produced by the pre-trained models are effective in terms of semantic representations.

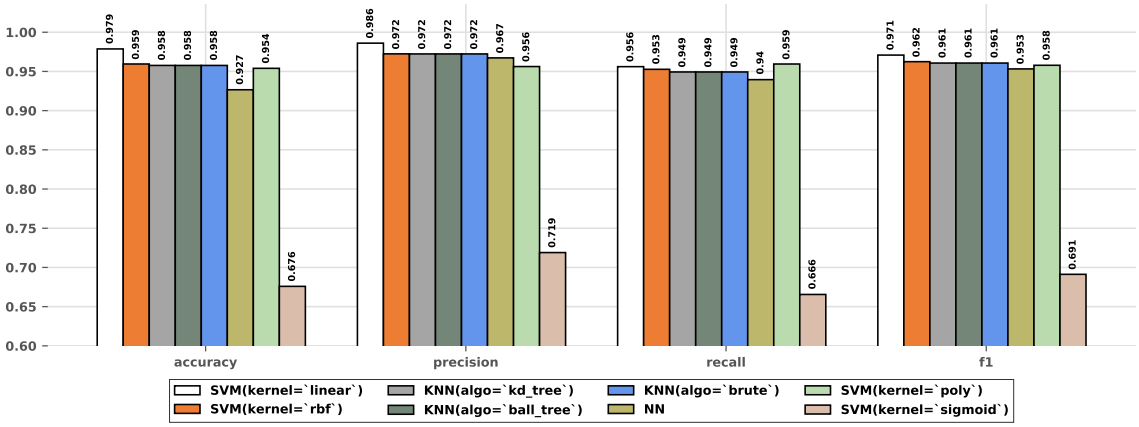


Fig. 12. Impact of classification algorithms on clone detection performance

6 DISCUSSION

Our experimental evaluation bears some threats to validity, while the approach itself has limitations that can be improved in future work.

	Approach	Accuracy	Precision	F1 score
SySeVR	(SOTA)	98.0	90.8	92.6
WYSIWIM	with ResNet50 pre-trained model	90.9	90.9	90.9
CODEGRID	with ResNet50 pre-trained model	91.2	90.8	90.9
Checkmarx	On a different dataset	72.9	30.9	36.1

Table 11. Performance Comparison for Vulnerable Code Prediction

6.1 Threats to Validity

Internal Validity - Dataset. Our dataset selections are limited in terms of size and diversity of functionalities. For the clone detection variant in particular, even though the number of clones is rather high, the number of code fragments that the clone and non-clones pairs are based on is still very small as the pairs are formed from those base code fragments by pairwise combination. This lack of diversity might negatively influence the generalizability of the evaluation results. Nevertheless, we mitigate this threat in the comparison experiment (RQ1) by using the same datasets as the state-of-the-art (i.e., ASTNN).

External Validity - Dataset. Even though the BigCloneBench is widely used throughout the literature, the judgment of whether or not a pair of code fragments form a clone remains biased and purely based on benchmark authors' intuition. Further, there is no single or precise notion of what semantic similarity is. Thus, the semantic boundaries of the functionality classes might not be consistent across all the represented functionalities. Finally, the program semantics of a code fragment might be obscured by the usage of external libraries that are not included within the dataset, in which case the decision task is technically unfeasible.

External Validity - Neural Network Architecture. In this paper we have worked with the ResNet neural network architecture (cf. Section 2.2.4). However, we have also conducted the experiments with the DenseNet [22] architecture. Although more sophisticated in its design, the results (Figure 13) show that DenseNet and ResNet50 have almost similar performance.

External Validity - Classification Algorithms. For the final learning and prediction stage of the code clone detection task, we used a simple binary classification neural network but also two (02) classical classification algorithms among the several existing ones. The experimental results for this task can then be influenced by the chosen classification algorithms. Therefore, we selected the simplest and widely used algorithms (kNN and SVM) in order to reduce the potential biases.

External Validity - Presence of clone duplicates in BCB. During development, we noticed the existence of conceptually duplicated clones in BigCloneBench. This fact showed up in the form of identical visual representations of code for different code fragment ids. It turned out that those fragments emerged from Type-1 clone pairs, which are technically the same code. When those both clone fragments are combined with another code fragment to form clone pairs, those clone pairs are conceptually duplicated. Although we cannot provide precise statistics on the extent of clone duplicates present in BigCloneBench, we can approximate, based on the code fragments that are used in Type-1 and Type-4 clone pairs, an upper-bound of approximately 30 percent clone duplicates. If we consider that Type-2 and Type-3 code snippets can also build clone duplicates, this estimation goes up to even 60 percent. In our case, during the development, we experienced drops of performance of about 10 percent, on small development examples. Hence we conclude that code clones should not be disregarded if precise and valid evaluations are desired. This conclusion is

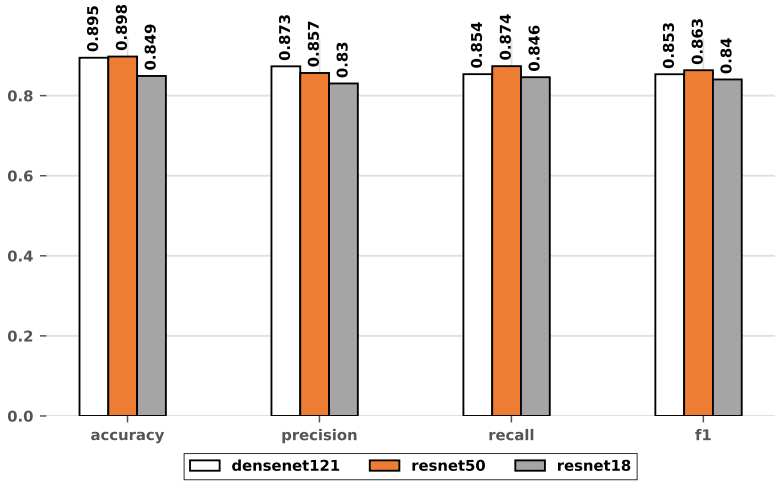


Fig. 13. Code classification performance: Densenet vs. Resnet architectures

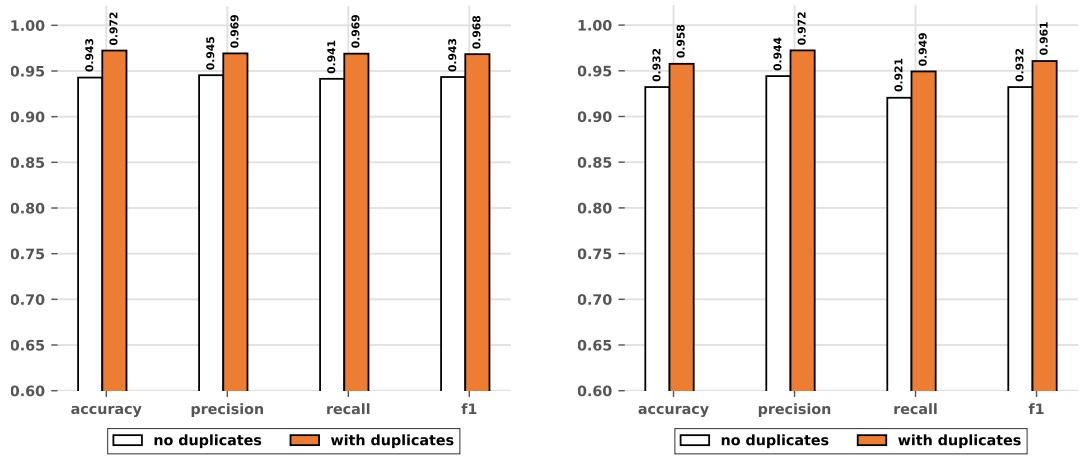
consistent with recent empirical results reported by Alamanis on the adverse effects of code duplication in machine learning models of code [1].

To explore the impact of code duplicates on the performance of WYSIWIM, we build a dataset (based on the same three functionalities and numbers of clones/non-clone pairs) where we do not use any clones that contain code fragments that are also used in Type-1 clone pairs. The results from figure 14 show that the avoidance of clone duplicates slightly degrades the overall results. This makes sense since the existence of clone duplicates makes the task easier and allows to achieve a higher score. This finding is further valid for SVM with *linear* (Figure 14 (a)) kernel and k-NN with *ball tree* algorithm (Figure 14 (b)).

Construct Validity - Dataset. A recurrent construct validity issue in the machine learning literature is related to class imbalance. In clone detection, one must ensure that all functionalities are balanced in the dataset of clone and non-clone pairs. Some approaches may overfit to specific (and largely represented) classes. To check for this issue, we build a balanced dataset (with and without duplicates) and compared the performance of WYSIWIM clone detection on this dataset as well as the imbalanced dataset provided in ASTNN artifacts. Indeed, the ASTNN dataset is randomly sampled from the BCB (using a fixed random seed) and hence -more or less- keeps the unbalancing that is present in the BCB itself. Comparison results in Figure 12 with balanced and imbalanced (i.e., ASTNN dataset) suggest that WYSIWIM keeps its promises on performance.

Dataset	Accuracy	F1 score	Precision	Recall
ASTNN	91.8	94.8	95.4	94.3
balanced	94.1	94.5	95.2	93.8
balanced w/o duplicates	92.1	92.0	94.1	90.1

Table 12. Impact of class imbalance in the dataset of code clones



(a) SVM with *linear* kernel

(b) k-NN with *ball tree* kernel

Fig. 14. Influence of clone duplicates.

Construct Validity - Geometric Visual Representation Due to the lack of sufficient geometric forms to map the vocabulary of terms in our datasets, we only replace language keywords with some geometric forms. Other terms are left as is. The mix of words and geometric forms may however lead to a deterioration of the learning performance

Construct Validity - Cross validation. We did not perform any cross validation on our approach, as our goal was to rather convey the concepts behind the approach rather than achieve high results. It is probable that the exact results vary to a certain extent on different splits of the dataset, especially since the different code fragments are probably not "semantically equally diverse" to each other, without further specifying what that could mean.

Conclusion Validity - Lack of definitions of semantic similarity. The software engineering community faces a crucial challenge for defining what semantic similarity means. Since we do not dare to explicitly define what semantic similarity means, we have to rely on the semantic value that is embedded in our dataset, respectively as it was implied by the creators of the BCB. In consequence, a specific selection of a subset of the dataset may even influence the overall semantics it carries. However, even when two approaches are applied on the same dataset, they might still view semantic similarity differently. These facts make it hard to evaluate and especially compare semantic approaches of any sort.

6.2 Limitations

Input size of ResNets. Image classification networks have technically and by construction a strong limitation on their input size. This is problematic as it introduces loss and distortion of our input data. In consequence, we may completely loose the fine-grained lexical information that is contained in the visual representations of our code fragments.

Code fragment granularity. The approach as presented is mainly designed to work with method granularity code fragments. Image classification networks are designed to assign a single most suitable label to a whole single input. This is consistent with generally accepted good coding style rules, which claim that a single method should always implement a single functionality (known as the single responsibility principle) [42]. To enlarge the scope of the granularity, our core concept of code visualization could be leveraged to full programs by applying object localization instead of detecting

what functionalities a software is composed of. This principle could also explain why our approach works slightly less well on the OJ dataset, which consists of whole programs, while the BigCloneBench rather provides method level granularities.

Colors in visualizations. Our visualizations apply colors only very sparsely, in the case of the color syntax highlighting variant, or not at all for the other visual representations. The current implementation of WySiWiM is thus not fully leveraging the potential of ResNet, which is designed to operate on all 3 color channels.

Traditional classification algorithms. For the clone detection task we apply very basic binary classification algorithms. These algorithms do probably not explore all semantics learned by the ResNet deep feature extractor.

Scope of the clone datasets. The datasets are not only a threat to validity but also a major limitation. Our hypothesis is that, due to the limited variety and size of the datasets available today, it is not possible yet to learn general semantic knowledge that can by applied to all possible data.

6.3 Lessons learned and Future work

As the current implementation of WySiWiM represents only a proof-of-concept with limited goals, it offers a lot of potential for extensions and improvements. Furthermore, the general concept of visualizing code and learning on those visual representations could be interesting also to other software engineering tasks, or could be combined with existing approaches. Beyond our approach, we identified some general current limitations on the task of semantic code clone detection, such as the lack of suitable datasets and benchmarks but also the lack of more precise and actionable definitions of semantics or semantic similarity.

Visual vs Semantics. It is rather intuitively acceptable that the visual representation of code works well for tasks such as code classification. However our experiments also show that WySiWiM neural networks yield features that help to identify semantically equivalent code fragments that are actually visually different (Type 4 clones). This shows the power of the generic features that are extracted from raw (straightforward) visual representations, which should be further investigated in other tasks that deal with semantics.

Mitigating Image classifier input limitation. As mentioned in the previous sub-section, a major limitation of our approach is the fixed input image size of the ResNet classifier. One potential way to mitigate this limitation could be to slice the image into multiple images of the required input size. Those slices could then be used to generate a larger feature vector, representing the whole image. This would allow to capture more fine-grained information as well. Of course, it might be necessary to apply also scaled versions of the images to capture large-scale structural information too.

Visualizations. As our visualizations showed, the use of colors can have a positive effect on the results. However, as our condensed AST visualization has been one to yield the best overall results, it might be interesting to further apply color coding on ASTs to make better use of the full potential of the image classification neural networks.

Datasets and Benchmarks: A future work that is important beyond our approach is the development of datasets and benchmarks that are more suitable for semantic code clone detection and semantic approaches in general. This includes a high number of different functionalities and a high number of diverse code examples per functionality. Especially sets providing a multitude of more basic functionalities that do not depend on external libraries would be desirable. They would allow to learn models the way humans learn semantics of computing languages, by starting very small.

Data augmentation: Similarly to data augmentation done in image classification via generating variant images through rotation, cropping, etc., we could envision to apply a data augmentation, although at the meta level, such as mutating the code in semantically-equivalent ways in order to increase the size of our dataset.

Actionable definitions of semantics (similarity): Another very important future work would be to make efforts towards actionable definitions of semantics, or semantic similarity. A possible approach to this could be the definition of semantics through software tests. As software tests represent an executable variant of software specifications, they give a good notion of the requirements we put into our semantics. Of course, there are a few problematic aspects in this approach. One aspect is that each application may require different abstractions of a certain functionality. Another aspect is that the code snippets for a certain functionality would all have to use test suites.

7 CONCLUSIONS

We presented a novel direction to code semantics learning based on visualization and transfer learning. WySiWiM exploits the power of pre-trained ResNets to extract deep features from visualization renderings of source code fragments. We apply this approach to two variants of clone identification, namely code classification and clone detection as well as to the task vulnerable code prediction. Experimental results on BigCloneBench and the Open Judge datasets show that our approach performs reasonably well and can keep up with the state-of-the-art within the scope of our experimental settings (which we carefully design to be comparable to literature experiments). Our experiments reveal that visualizations of AST and Color syntax highlighting yield the best overall clone detection results. We complete the paper by enumerating a list of limitations, which, if resolved, may unleash a huge potential of WySiWiM beyond clone identification tasks.

Availability: All experimental data as well as the source code of WySiWiM is open sourced in an anonymous repository:

https://github.com/wysiwim/wysiwim

ACKNOWLEDGMENTS

This work was partly supported (1) by the Luxembourg National Research Fund (FNR) - NERVE project, ref. 14591304 and CHARACTERIZE project, ref. 11693861, (2) by the Luxembourg Ministry of Foreign and European Affairs through their Digital4Development (D4D) portfolio under project LuxWAyS and (3) by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Project NATURAL - grant agreement N^o 949014).

REFERENCES

[1] Miltiadis Allamanis. 2018. The Adverse Effects of Code Duplication in Machine Learning Models of Code. *arXiv preprint arXiv:1812.06469* (2018).
[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).
[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.
[4] Ambient Software Evoluton Group. 2013. IJaDataset 2.0, http://secold.org/projects/seclone.
[5] Relja Arandjelovic and Andrew Zisserman. 2017. Look, Listen and Learn. In *The IEEE International Conference on Computer Vision (ICCV)*.
[6] B.S. Baker. 1992. A Program for Identifying Duplicated Code. In *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface*, Vol. 24. 49–57. Issue Mar.

[7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference onSoftware Maintenance*. IEEE, 368–377.

[8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.

[9] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. 2010. A theory of learning from different domains. *Machine learning* 79, 1 (2010), 151–175.

[10] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: a learnable representation of code semantics. In *Advances in Neural Information Processing Systems*. 3585–3597.

[11] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, and V. Filkov. 2020. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 578–589.

[12] Zimin Chen and Martin Monperrus. 2019. A Literature Study of Embeddings on Source Code. *arXiv preprint arXiv:1904.03061* (2019).

[13] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. *Training* 100, 101 (2017), 102.

[14] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[15] Thomas M Cover, Peter Hart, et al. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[17] FaCoY. 2017. https://github.com/facoy/facoy.

[18] Yi Gao, Shuang Liu Zan Wang, Sang Wei Lin Yang, and Yuanfang Cai. 2019. TECCD: A Tree Embedding Approach for Code Clone Detection. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[19] Simon Haykin. 1994. *Neural networks: a comprehensive foundation*. Prentice Hall PTR.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[21] Minyoung Huh, Pulkit Agrawal, and Alexei A Efros. 2016. What makes ImageNet good for transfer learning? *arXiv preprint arXiv:1608.08614* (2016).

[22] Simon Jégou, Michal Drozdzal, David Vazquez, Adriana Romero, and Yoshua Bengio. 2017. The one hundred layers tiramisu: Fully convolutional densenets for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 11–19.

[23] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[24] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 81–92.

[25] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2010. Code similarities beyond copy & paste. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 78–87.

[26] Pooja Kamavisdar, Sonam Saluja, and Sonu Agrawal. 2013. A survey on image classification approaches and techniques. *International Journal of Advanced Research in Computer and Communication Engineering* 2, 1 (2013), 1005–1009.

[27] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[28] H. Kim, Y. Jung, S. Kim, and K. Yi. 2011. MeCC: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*. IEEE, 301–310.

[29] J. Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. 301–309.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[32] D. E. Krutz and E. Shihab. 2013. CCCD: Concolic code clone detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 489–490.

[33] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 249–260.

[34] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. 2016. Measuring Code Behavioral Similarity for Programming and Software Engineering Education. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, New York, NY, USA, 501–510.

[35] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. 2020. Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning. *Applied Sciences* 10, 5 (2020), 1692.

[36] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. USENIX Association, Berkeley, CA, USA, 20–20.

[37] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2018. Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756* (2018).

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[39] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 872–881.

[40] Dengsheng Lu and Qihao Weng. 2007. A survey of image classification methods and techniques for improving classification performance. *International journal of Remote sensing* 28, 5 (2007), 823–870.

[41] Andrian Marcus and Jonathan I Maletic. 2001. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 107–114.

[42] Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.

[43] Kris McGuffie and Alex Newhouse. 2020. The radicalization risks of GPT-3 and advanced neural language models. *arXiv preprint arXiv:2009.06807* (2020).

[44] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[45] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[46] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[47] National Institute of Standards and Technology. 2018. National Vulnerability Database. http://nvd.nist.gov/.

[48] National Institute of Standards and Technology. 2018. Software Assurance Reference Dataset. https://samate.nist.gov/SRD/index.php.

[49] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[50] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch. *Computer software. Vers. 0.3* 1 (2017).

[51] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[52] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories*.

[53] Chaiyong Ragkhitwetsagul, Jens Krinke, and Bruno Marnette. 2018. A picture is worth a thousand words: Code clone detection based on image similarity. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. IEEE, 44–50.

[54] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[55] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 354–365.

[56] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 702–714.

[57] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code Relatives: Detecting Similarly Behaving Software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, 702–714.

[58] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 476–480.

[59] The MITRE Corporation. 2018. common Weakness Enumeration. https://cwe.mitre.org/.

[60] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 542–553.

[61] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[62] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 783–794.

[63] Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. 2015. Long short-term memory over recursive structures. In *International Conference on Machine Learning*. 1604–1612.

[64] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.

A WYSIWIM CODE VISUALIZATION EXAMPLES

A.1 Vulnerability Prediction

```
public CsrfConfigurer<H> ignoringAntMatchers(String... antPatterns) {
    return new IgnoreCsrfProtectionRegistry().antMatchers(antPatterns).and();
  }
```

(a) Vulnerable code fragment

```
public CsrfConfigurer<H> ignoringAntMatchers(String... antPatterns) {
    return new IgnoreCsrfProtectionRegistry(this.context).antMatchers(antPatterns)
        .and();
  }
```

(b) Non-Vulnerable code fragment

Fig. 15. Plain Text visual representation of a CWE-264 example of code fragment

```
public CsrfConfigurer ignoringAntMatchers(String... antPatterns) {
    return new IgnoreCsrfProtectionRegistry().antMatchers(antPatterns).and();
  }
```

(a) Vulnerable code fragment

```
public CsrfConfigurer ignoringAntMatchers(String... antPatterns) {
    return new IgnoreCsrfProtectionRegistry(this.context).antMatchers(antPatterns)
        .and();
  }
```

(b) Non-Vulnerable code fragment

Fig. 16. Color Syntax Highlighting visual representation of a CWE-264 example of code fragment

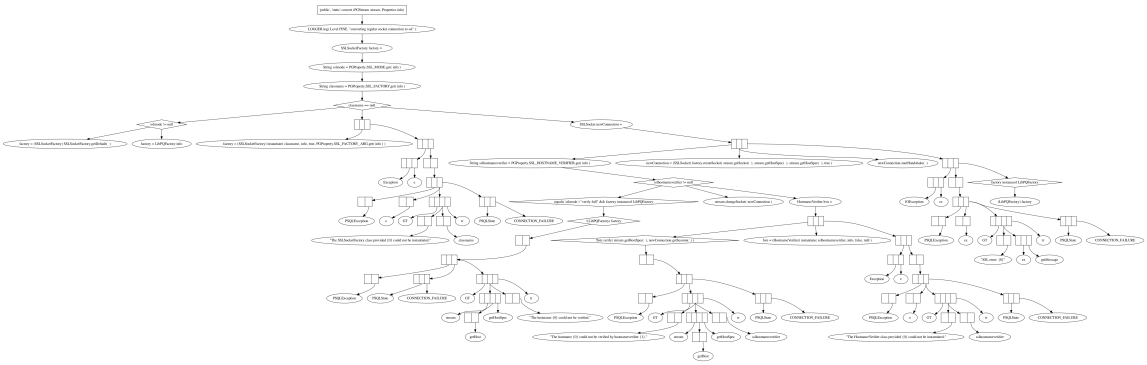
```
fun ()
{
wchar_t *data;
wchar_t *dataBadBuffer =(wchar_t *)ALLOCA (50*sizeof (wchar_t ));
wchar_t *dataGoodBuffer =(wchar_t *)ALLOCA (100*sizeof (wchar_t ));
⊕(staticTrue )
{
data =dataBadBuffer ;
data [0]=L'\0';
}
{
wchar_t source [100];
wmemset (source ,L'C',100-1);
source [100-1]=L'\0';
wcsncat (data ,source ,100);
printWLine (data);
}
}
```

(a) Vulnerable code fragment

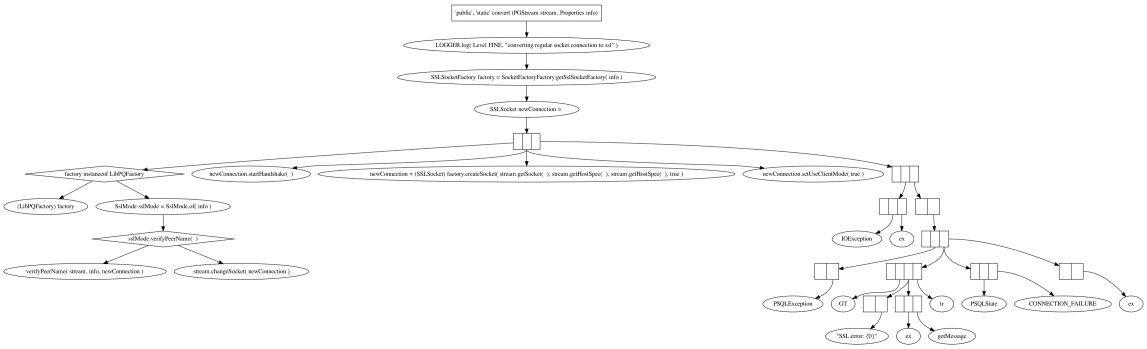
```
static fun ()
{
wchar_t *data;
wchar_t *dataBadBuffer =(wchar_t *)ALLOCA (50*sizeof (wchar_t ));
wchar_t *dataGoodBuffer =(wchar_t *)ALLOCA (100*sizeof (wchar_t ));
⊕(staticFalse )
{
printLine ("Benign, fixed string" );
}
⊖
{
data =dataGoodBuffer ;
data [0]=L'\0';
}
{
wchar_t source [100];
wmemset (source ,L'C',100-1);
source [100-1]=L'\0';
wcsncat (data ,source ,100);
printWLine (data);
}
}
```

(b) Non-Vulnerable code fragment

Fig. 17. Geometric visual representation of an example of code fragment from *API Function Call* vulnerability syntactic characteristic samples



(a) Vulnerable code fragment



(b) Non-Vulnerable code fragment

Fig. 18. AST in condensed format visual representation of a CWE-297 example of code fragment