Received July 28, 2020, accepted August 9, 2020, date of publication August 14, 2020, date of current version August 26, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3016774

Vulnerability Prediction From Source Code Using Machine Learning

ZEKI BILGIN, (Member, IEEE), MEHMET AKIF ERSOY, ELIF USTUNDAG SOYKAN, EMRAH TOMUR, PINAR ÇOMAK, AND LEYLI KARAÇAY
Ericsson Research, 34367 İstanbul, Turkey

Corresponding author: Zeki Bilgin (zeki.bilgin@ericsson.com)

This work was supported by the Scientific and Technological Research Council of Turkey through the 1515 Frontier Research and Development Laboratories Support Program under Project 5169902.

ABSTRACT As the role of information and communication technologies gradually increases in our lives, software security becomes a major issue to provide protection against malicious attempts and to avoid ending up with noncompensable damages to the system. With the advent of data-driven techniques, there is now a growing interest in how to leverage machine learning (ML) as a software assurance method to build trustworthy software systems. In this study, we examine how to predict software vulnerabilities from source code by employing ML prior to their release. To this end, we develop a source code representation method that enables us to perform intelligent analysis on the Abstract Syntax Tree (AST) form of source code and then investigate whether ML can distinguish vulnerable and nonvulnerable code fragments. To make a comprehensive performance evaluation, we use a public dataset that contains a large amount of function-level real source code parts mined from open-source projects and carefully labeled according to the type of vulnerability if they have any. We show the effectiveness of our proposed method for vulnerability prediction from source code by carrying out exhaustive and realistic experiments under different regimes in comparison with state-of-art methods.

INDEX TERMS AST, machine learning, source code, vulnerability prediction.

I. INTRODUCTION

Software vulnerabilities have widespread impact and cause substantial economic and reputational damage to both companies and people, developing or using software products [1], [2]. Therefore, it is highly critical to detect and eliminate potential vulnerabilities as early as possible. A vulnerability is defined as a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source [3], whereas a flaw or bug is a defect in a system that may (or may not) lead to a vulnerability [4]. Thus, vulnerabilities are actually the subclass of software bugs that can be exploited for malicious purposes [5], [6]. Vulnerabilities require quite a different identification process than defects because they are often not realized by users or developers during the normal operation of the system while defects are more easily and naturally noticed [6]. These make the fighting against vulnerabilities much more challenging than typical defects.

There are two traditional approaches used for vulnerability detection: (i) static analysis and (ii) dynamic analysis. In static analysis, the code is examined for weaknesses without executing it. Therefore, the potential impact of the executable environment, such as the operating system and hardware, is not taken into consideration during analysis [7]. On the other hand, in dynamic analysis, the code is executed to check how the software will perform in a run-time environment, but this can only reason about the observed execution paths and not all possible program paths [7]. Hence, both static and dynamic code analyses have some problems on their own. Some of the tools used as source code security analyzer are listed[1] along with their basic capabilities by National Institute of Standards and Technology (NIST) in the scope of Software Assurance Metrics And Tool Evaluation (SAMATE) project.[2] Considering that both static and dynamic analysis may be ineffective in detecting some

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

[1]https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

[2]https://samate.nist.gov/Main_Page.html

vulnerabilities in certain situations [4], [8], the SAMATE project provides very useful overview, evaluations and test results about effectiveness of several static code analysis tools based on a public dataset that includes real and synthetic test cases with a set of known security flaws [4].

The recent advances in computing power, availability of data, and new algorithms have led to major breakthroughs in artificial intelligence (AI) and machine learning (ML) in the last decade. Many applications of AI/ML have become ubiquitous in everyday life, ranging from automation and natural language processing to forecasting and privacy issues [9], [10]. They are increasingly exploited in the sectors as diverse as industry, government and commerce [11]. In this sense, yet another promising application area for AI/ML is automated and intelligent software analysis with various objectives such as vulnerability prediction [6], code summarization [12], [13], code classification and clone detection [14]. Employing ML for software security analysis not only reduces the dependence on domain experts for the hand-crafted pattern or feature extraction [15], but also helps to simplify and automate processes that are required for the current security analysis techniques [7].

Leveraging data-driven techniques (e.g., AI/ML) for performing automated intelligent analysis directly on source code requires to solve some challenges such as representing source code in a proper form to enable further analysis in ML algorithms and localizing detected vulnerabilities on source code. In this study, we first seek solutions to these challenges and propose methods to tackle them. Then, we model vulnerability prediction task as a binary classification problem for each targeted vulnerability class such that our ML model takes a source code fragment as input and decides whether it is *vulnerable* (i.e. containing the targeted vulnerability) or *non-vulnerable*. For experimental analysis and performance evaluation, we implement our methodology and conduct a set of experiments on the different subsets of the public Draper VDISC Dataset[3] [16] that contains a large amount of labeled real source code components mined from several open-source projects such as Debian Linux distribution [17] and public Git repositories on GitHub [18] and the codes from SATE IV Juliet Test Suite [19]. To the best of our knowledge, this is the first study performing AST-based vulnerability prediction analysis on the Draper VDISC Dataset.

The main contributions of this article include:

- An ML-based vulnerability prediction method that is implemented and experimentally evaluated with a publicly available vulnerability dataset containing a great number of *real* source code fragments, which were carefully labeled according to findings from several static analyzers,
- A method for vectorial representation of source code while preserving syntactic and semantic relations contained in the source code fragments,
- The experimental analysis showing that even the partial Abstract Syntax Tree (AST) representation of the source code can be useful for vulnerability prediction when it is not feasible to extract or process whole AST of the corresponding source code, and
- Adapting a prior state-of-art source code representation method [20] to the vulnerability prediction problem for performance comparison.

The rest of the paper is organized as follows. In Section II, we provide a review of some related noteworthy works. Then, in Section III, we give background information of the proposed source code representation technique that is explained in the subsequent Section IV. Later on, in Section V, we present our vulnerability prediction approach along with a set of comparative experimental analysis. Afterward, in Section VI, we discuss some additional examinations we have done related to the proposed method. Section VII concludes the paper.

II. RELATED WORK

One of the earliest studies in vulnerability prediction belongs to Neuhaus and Zimmermann [21], who observed that the software components that had similar imports or function calls were likely to be vulnerable to the same vulnerability. They developed their vulnerability prediction model based on this observation and validated it in the Mozilla project. Neuhaus *et al.* [22] later extended this idea to the analysis of dependencies between packages in the Red Hat.

A. SOFTWARE METRICS

Some studies [2], [23], [24] investigate whether software metrics obtained from source code and development history are discriminative and predictive of vulnerable code locations. For example, Shin *et al.* [2] examined the applicability of three types of software metrics (complexity, code churn, and developer activity) to build vulnerability prediction models. They performed empirical analyses on two open-source projects, the Mozilla Firefox and the Red Hat Enterprise Linux kernel, and found that 24 of the 28 metrics collected are discriminative of vulnerabilities for both projects. Another work [24] demonstrated that some trivial software metrics such as character diversity, string entropy, function length and nesting depth could be useful indicators for vulnerability detection.

B. TEXT MINING

It is claimed [25] that the prediction techniques, in intelligent software analysis, based on text mining perform better than the prediction techniques based on software metrics. The former approach treats the source code as regular text in general and leverages the natural language processing (NLP) techniques for code representation and feature extraction. The naturalness hypothesis of software approaches the subject in a similar way and asserts that although programming languages, in theory, are complex, flexible and powerful,

[3]https://osf.io/d45bw/

the code fragments that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks [26]. For example, following the word embeddings concept in NLP, the authors of [27] generated a set of general-purpose models pre-trained over large amounts of code. Although the authors claimed that their models could be used to assist a number of information retrieval tasks, including identifying semantic errors, they did not provide any experimental results for these tasks. Another noteworthy study [16] adopting NLP based approaches for vulnerability detection belongs to Russell *et al.*, who used a method that was initially developed for sentence sentiment classification in NLP. They adapted this method to classify ''vulnerable'' and ''not vulnerable'' source code components by employing deep feature representation learning over one-hot encoding of the tokens obtained from the lexed source code. For performance evaluation, they built their own dataset, called the Draper VDISC Dataset, containing millions of function-level C and C++ source code fragments mined from open-source projects and labeled as ''vulnerable'' or ''not vulnerable'' according to findings from several static analyzers. In the presented study, we use a subset of this Draper VDISC Dataset for experimental analysis; however, our study differs from [16] especially in the following aspects: (i) our analysis is based on AST representation of source code rather than identifiers in the source code, and (ii) we implement a categorical encoding for the identified tokens rather than one-hot encoding.

C. AST-BASED ANALYSIS

Treating the source code as natural language has some limitations in capturing comprehensive program semantics to characterize vulnerabilities of high diversity and complexity in real source code, because source code is actually more structural and logical than natural languages and has various aspects of representation such as AST, data flow and control flow [14], [15]. For this reason, some studies [12]–[14], [20] seek to investigate alternative approaches for a more efficient representation of source code in ML-based software analysis. Aiming to reduce information loss in the process of representation learning, Zhou *et al.* [15] presented a vulnerability identification model that encodes a function-level source-code fragment into a joint graph structure from multiple syntax and semantic representations and then leverage the composite graph representation to learn how to discover vulnerable code. Another noteworthy study in this category belongs to Alon *et al.* [20], who presented a neural model for representing snippets of code as continuous distributed vectors (i.e. ''code embeddings''), based on collection of paths in its AST. The authors demonstrated the effectiveness of their approach by using it to predict a method's name from the vector representation of its body. We make performance comparison of our proposed method with [20] and provide detailed comparative analysis in Section V-E3.

D. CHALLENGES

A recent thesis [6] evaluates the effectiveness of vulnerability prediction methods and mentions the challenges in vulnerability prediction research, such as having a lack of reliable vulnerability dataset and lack of replication framework for comparative analysis of existing methods. In complying with this, it is shown in [28] that sufficient and accurately labeled data has a great impact on the performance of ML-based vulnerability prediction methods. Another difficulty in vulnerability prediction is the class imbalance problem, arising from the fact that the number of vulnerable code samples is far less than the number of healthy code samples, which makes it hard to perform good prediction performance without giving too many false alarms. In this sense, the study [29] deals with the class imbalance problem in ML-based vulnerability detection approaches, and proposes a fuzzy oversampling method to balance the training data by generating synthetic samples for the minority class (i.e. vulnerable code samples). However, notice that an effective and realistic vulnerability detection model should be able to perform well even against the highly imbalanced cases because original distribution of the data in nature will remain same. Therefore, in our presented work, we evaluated the performance of our proposed vulnerability prediction method against the original distribution of vulnerable and not vulnerable code fragments, which is highly imbalanced.

III. PRELIMINARIES

Source code is mostly written in high-level programming languages such as C/C++, Java and Python, which comprise text-based words and phrases from natural language. Some compilers or interpreters process and translate the source code into low-level programming languages (e.g., assembly language, object code, or machine code) that are more appropriate to be executed by the instructions in computer architecture. In this process of compiling source codes, there are several intermediate steps where source codes are subject to different procedures, such as lexical analysis, parsing, AST representation, etc. [30].

In the lexical analysis, the source code is transformed into a series of tokens, by discarding any whitespace or comments included in the code. For example, in C language, a line of code given below:

```
int a = 5; // This is a comment!
```

which produces the following sequence of tokens:

```
int (keyword), a (identifier),
= (operator), 5 (constant),; (symbol)
```

In the parsing process, the tokens generated in the lexical analysis are converted into a data structure – mostly a kind of parse tree or other hierarchical structure, giving a structural representation of the input while checking for correct syntax based on the rules of a context-free grammar (CFG). This step usually yields AST representation of the given source code, which is a tree type data structure based

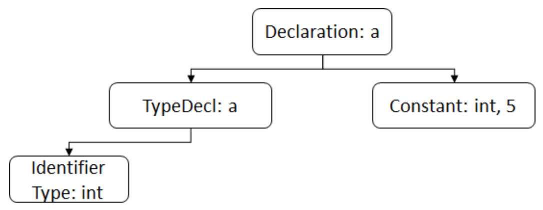

FIGURE 1. AST of the code "int a = 5; // This is a comment!".

on the tokens extracted in lexical analysis. In the context of the same example, AST of the code given above is depicted in Figure 1.

The fact that AST contains both structural and semantic information related to the code gives rise to the trend of developing AST-based intelligent analysis of source code [13].

IV. SOURCE CODE REPRESENTATION

In this study, we aim to develop an ML model for vulnerability prediction based on the AST representation of source code. A significant challenge to achieving this goal is to extract useful features from the code to be analyzed. This obstacle, as briefly mentioned in Section II, is often tackled by adopting some techniques developed initially for NLP purposes. Unlike NLP-based approaches, in this work, we present a novel method that allows us to directly give the AST representation of source code into ML algorithms as input. More specifically, we translate the AST representation of source code into a one-dimensional numerical array, where each element can be treated as a feature in an ML model. To preserve structural and semantic information contained in the source code during this translation process, we apply clever techniques both while converting the tree-type AST structure to array form and mapping the identified tokens to categorical numeric values. Thus, the presented method has potential to leverage data-driven techniques (e.g., AI/ML) for performing automated intelligent analysis directly on source code, such as vulnerability prediction, similarity analysis, code completion, and more, which helps to reduce the need for domain expert and manual analysis to carry out these tasks.

More specifically, as depicted in Figure 2, we initially split source code into smaller parts to allow more granular analysis. Then, we generate and extract AST for each departed code component, which also includes a tokenization process via a lexer. Later on, we convert the extracted AST into the complete binary tree that has a deterministic shape where it is specified how many nodes are located at each level of the tree. Afterward, we encode each token in the complete binary AST to pre-defined numerical tuples and finally obtain a one-dimensional numerical array representation of the corresponding function-level source code by concatenating the assigned numerical tuples from the root node to leaves in order. We justify each step in detail in the following parts, along with examples.

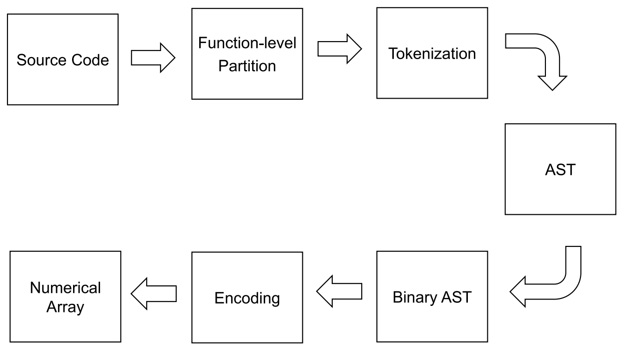

FIGURE 2. Steps of the proposed source code representation method.

Step-1 (Source Code): In this step, the source code to be processed is taken as an input. The source code can be written in any high-level programming language, such as C, C++, Java, and Python, provided that its AST can be obtained with the help of an appropriate parser. From this perspective, the proposed method is actually language-agnostic, that is, it can be applied to different languages.

Step-2 (Function-level partition): The source code of a program or an application can be arbitrarily long composed of a lot of components, functions, and lines. Instead of dealing with the source code as a whole, it is a good practice to split it into sub-parts and handle each sub-part separately, which would increase granularity. For this purpose, we prefer to use function-level source code because it is the lowest level of granularity, capturing the overall flow of a subroutine [13], [15], [16]. This is also good for more precise localization of the predicted vulnerabilities. From now on, the term "source code" is used in the sense of function-level source code. To make the subsequent steps more concrete and understandable, we provide a sample function-level source code written in C language given below, which will be referenced in the subsequent steps with the corresponding examples.

```
int main()
{
int a = 5, b = 2;
printf(a+b);
}
```

Step-3 (Tokenization): In this step, firstly, the source code is cleaned by removing its unnecessary elements such as comments, whitespaces, tabs, newlines, etc. Then, the remaining part is converted into a series of tokens, where a token is a sequence of characters that can be treated as a unit in the grammar of the corresponding programming language. This can be achieved by using a lexer developed explicitly for the language of the source code. Some tokens extracted from the *main* function given above is as follows:

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

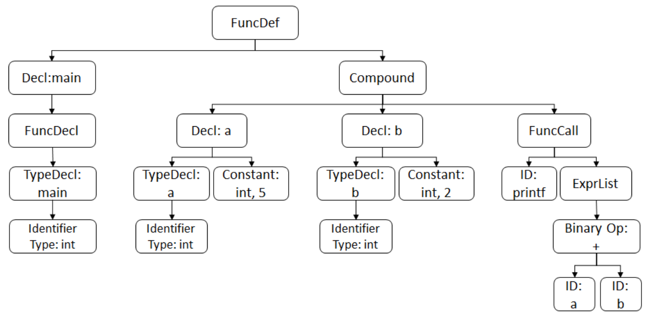

FIGURE 3. AST of the main function in Step-2 of Section IV.

Step-4 (AST Generation): In this step, AST of the source code is generated, which can be achieved by using a parser developed specifically for the language of the code. Different parsers may yield slightly different ASTs even for the same code depending on their implementation. Yet this would not be an issue in our case as long as the same parser is used for all samples. AST contains syntax and semantic information about the source code, and therefore it is highly useful for further analysis. Figure 3 shows AST of the *main* function given in Step 2. The structural relations (e.g., parent-child) are important in the AST and could be useful for vulnerability identification. Therefore, these relational pieces of information should not be lost during the transformations in source code representation stages. This is a bit challenging given that a regular AST is a kind of *m-ary* tree where there is no restriction on the values that *m* can take, which means each node may have an arbitrary number of child nodes that makes the structural shape of the AST unpredictable. To overcome this issue, we apply the next step to the AST.

Step-5 (Conversion to Complete Binary AST): AST is a tree type data structure and we need to convert it into an array format to feed an ML algorithm. Suppose for a moment that we convert an AST into an array by placing nodes side by side starting from root level to deeper levels, from leftmost node to rightmost node at each level. Several problems would occur in such a conversion. First of all, structural relations among AST nodes such as parent-child relations would be lost in the resulting array because a parent node may have arbitrary number of children. Second, the resulting arrays would be in different lengths. However, it is highly important to preserve structural relations among AST nodes while mapping them into a one-dimensional array because both it contains some semantic information about the code and neural network based models use such spatial information to extract hidden patterns.

As a solution, in this step, we convert a regular AST to the corresponding complete binary AST, where all leaves have the same depth, and all internal nodes have exactly two children. This can be accomplished in a variety of ways. We perform the following rules for doing this [31]:

- The leftmost child of node-x in the *m-ary* tree is assigned as the right-child of node-x in the corresponding complete binary tree (single child is treated as the left-most child here),
- The right sibling of node-x in the *m-ary* tree is assigned as left-child of node-x in the corresponding complete binary tree, and
- If node-x has no children, then its right-child becomes NULL, and if node-x is the rightmost child of its parent, then its left-child becomes NULL.

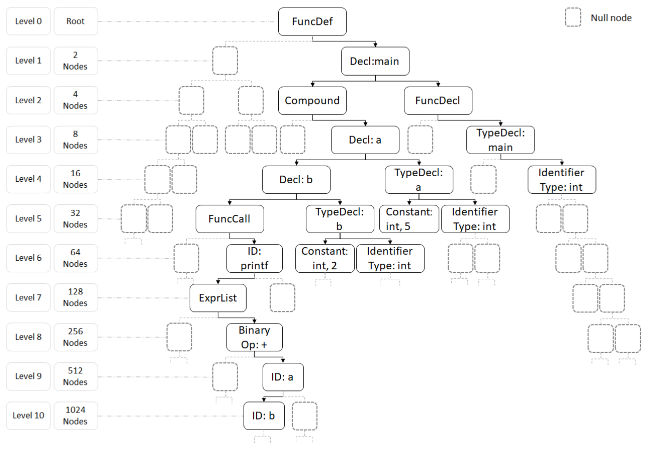

FIGURE 4. Complete binary AST of the regular AST in Figure 3.

Algorithm 1 Conversion From M-Ary Tree to Binary Tree

```
1:  procedure Encode(rootNode)
2:      if $rootNode = NULL$ then return $false$
3:      end if
4:      $(TreeNode)\ result \leftarrow TreeNode\ (rootNode.value)$
5:      if $children[rootNode] \neq NULL$ then
6:          $rightchild[result] \leftarrow ENCODE(leftmost[rootNode])$
7:      end if
8:      $(TreeNode)\ currNode \leftarrow rightchild[result]$
9:      for all $c \in rootNode$ do
10:         $leftchild[currNode] \leftarrow ENCODE(c)$
11:         $currNode \leftarrow leftchild[currNode]$
12:     end for
13: end procedure
```

Figure 3 and Figure 4 depict the regular AST and the corresponding complete binary version generated based on the above-mentioned rules, respectively, for the *main* function given in Step-2. In order to have a complete tree, NULL children nodes are added to NULL nodes until there is a level where all nodes are NULL.

Step-6 (Encoding to Numerical Tuples): In the obtained complete binary AST, nodes are named with words or strings such as "FuncDef", "Decl", "TypeDecl", "Constant", "ID", and so on. These names need to be encoded into categorical numeric values to allow them to be processed by ML algorithms. Therefore, we map them into predetermined numerical values, as exemplified in Table 1.

TABLE 1. Sample encoding from tokens to numerical tuples.

Token	Encoded Numerical Tuple
FuncDef	2.0, 0.0, 0.0
Decl	3.0, 0.0, 0.0
FuncDecl	4.0, 0.0, 0.0
TypeDecl	5.0, 0.0, 0.0
IdentifierType: int	6.0, 0.0, 103.0
Compound	7.0, 0.0, 0.0
Constant: int, value	8.0, 103.0, float(value)
FuncCall	9.0, 0.0, 0.0
ID: printf	10.0, 262.0, 0.0
ID: other	10.0, 0.0, 0.0
ExprList	46.0, 0.0, 0.0
BinaryOp (+)	14.0, 127.0, 0.0
"NULL node"	0.0, 0.0, 0.0

In this encoding, the first number in the encoded numerical tuple represents the type of token, while second and third numbers can be used to keep auxiliary information that may exist at nodes. For example, the token "*Constant: int, 100*" is encoded to (8.0, 103.0, 100.0). This can be seen as a three-dimensional data structure where each dimension keeps a value related to an associated token. Notice that the numeric values in the encoding are chosen arbitrarily and can be changed as long as different categories take different values. The encoding given in Table 1 corresponds to the tokens in Figure 4 where the NULL nodes in the AST are encoded to the tuple of 0.0, 0.0, 0.0.

Step-7 (Array Representation): In this final step, a numeric array is generated based on the complete binary AST such that the encoded numerical tuples are mapped to locations in the numeric array according to respective locations of the token nodes and NULL nodes in the complete binary AST. We prefer to use the Breadth-first search (BFS) approach for doing this, i.e., starting at the tree root and traversing all of the neighbor nodes at the present level prior to moving on to the nodes at the next depth level as illustrated in Figure 5.

As an example, following all steps described in this section, the *main* function given earlier is converted to the following array:

```
[2.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 3.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 7.0, 0.0, 0.0, 4.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 3.0, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 3.0,
 0.0, 0.0, 5.0, 0.0, 0.0,.......]
```

Notice that, in the presented approach, structural relations among tokens in an AST are preserved due to the respective positional relations between elements in the array.

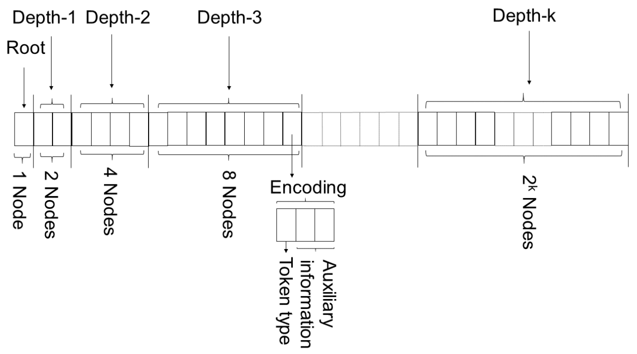

FIGURE 5. Converting complete binary AST structure to array form.

Thus, together with rich categorical encoding, as explained in the earlier steps, this numeric array corresponds to vector representation of the related AST and thus retains semantic information embedded in the AST. In addition to these useful properties, another significant advantage of the presented numeric array representation is that each particular element in the resulting array acts as a feature, and therefore can be directly inputted into an ML algorithm, which enables performing automated AST-based intelligent code analysis.

V. VULNERABILITY PREDICTION

We model the vulnerability prediction problem as a binary classification task such that our ML model takes the source code as input and decides whether it contains specific predefined vulnerabilities or not. For experimental analysis and performance evaluation, we used the publicly available dataset given in [16] containing a large amount of function-level source code collected from various open-source projects, where the details are given in the following subsections.

A. DATASET

The public Draper VDISC Dataset published at [16] contains a vast number of function-level source codes collected from several open-source projects such as Debian Linux distribution [17], public git repositories on GitHub [18] and SATE IV Juilet Test Suite [19] of NIST's Samate project. Unlike the first two, the SATE IV Juilet Test Suite contains synthetic codes, but it constitutes only about 1 percent of the entire dataset as indicated in [16]. The authors of [16] stated that they carefully labeled these function-level codes according to findings from three different static analyzers indicating potential exploits. They categorized these functions under 5 different groups of CWE vulnerabilities as indicated in Table 2 such that the functions flagged by the static code analyzers were labelled as vulnerable for the category of interest, while the others were labelled as non-vulnerable functions. The authors of [16] also mentioned that they split the whole dataset into three subparts as training (80%), validation (10%) and test (10%) sets, while paying attention to not include a duplicate of training sample in the test dataset.

To preserve the disjointness of the training and the test sets, we intentionally avoided k-fold cross validation, and instead, used the dataset in the original splitting in our experiments.

The Draper VDISC Dataset is highly imbalanced as the number of positive (i.e. vulnerable) samples is far less than the number of negative (non-vulnerable) samples due to fact that they are collected from real world projects and thus reflect the natural distribution of the targeted vulnerabilities. We keep this imbalance in our experiments to measure the effectiveness of our proposed method fairly. Also, we build balanced subsets in some of our experiments to measure the detectability of different vulnerability categories on equal terms.

The referenced dataset contains functions written in C and C++ languages. From this dataset, we selected only the functions that are written in C language and parseable by the parser, namely Pycparser [32], we used in our implementation. Thus, we obtained several subsets from the original training, validation, and test datasets.

B. IMPLEMENTATION DETAILS

In the implementation phase of the proposed source code representation method, we used the Pycparser library, which is a parser for the C language (C99), for generating ASTs of the source codes. Also, to convert a regular AST into a complete binary AST, we adapted some open-source implementations [33] to our case. To encode nodes of an AST into numeric values, we identified 48 different essential token types based on the grammar of C language and assigned unique values for each token type (the first number in the numerical tuple as exemplified in Table 1), as well as determined different values for encoding auxiliary information (e.g., *char, int, float, short, signed*, $+$, $-$, $==$, $>$, $<$, *etc.*) of tokens when required (the second and the third numbers in the numerical tuple). For the ML implementations, we used both the Scikit-learn [34] and the TensorFlow [35] libraries. In the Scikit-learn, we implemented and executed Multi-layer Perceptron (MLP) algorithm, while in the TensorFlow, we implemented and ran Convolutional Neural Network (CNN) algorithm.

C. IMPACT OF AST DEPTH

In a complete binary tree, the number of nodes that can exist at each level is certain and doubled in each subsequent depth level. Thus, at level-k, there are 2^k nodes. In total, for a complete binary AST with depth k, there are $\sum_{i=0}^{k} 2^i$ nodes. Because each node (token) is encoded to a tuple consisting of 3 numeric values, the resulting array size becomes $\sum_{i=0}^{k} 3.2^i$. However, all input data must be in the same size for all samples while feeding them into an ML algorithm, which may not be the case because the size of the resulting arrays may be different. To have an identical dimension in the resulting numeric array representation for all functions, we can either apply padding on the shorter arrays or cut the binary AST at a certain pre-determined level for all functions. We preferred the latter case because using the partial binary tree above a certain level might be more efficient and feasible than the padding which could increase the resulting array size unnecessarily. To figure out the optimal depth to be taken for this purpose, we investigated its impact on vulnerability prediction performance and conducted a set of experiments as follows.

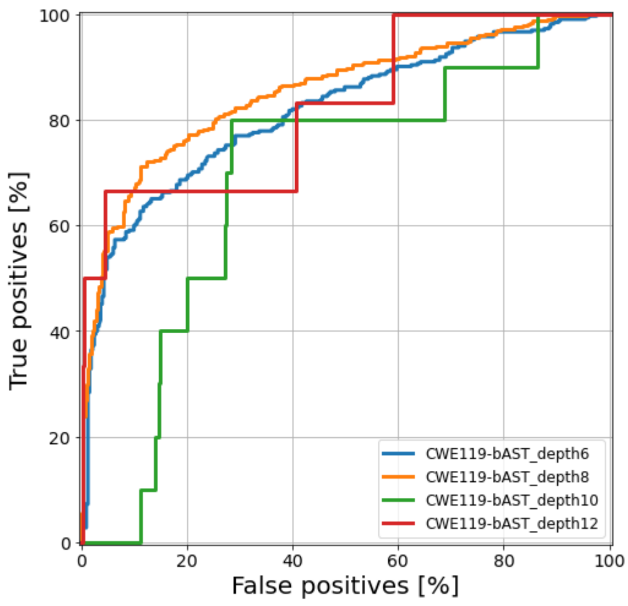

FIGURE 6. ROC curves for varying binary AST depths.

Extracting a subset of the dataset for the vulnerability type CWE-119, we generated corresponding numerical representation of the source code up to the binary AST depth of 6, 8, 10 and 12. Notice that the resulting numerical array size is 381, 1533, 6141 and 24573 for the depth of 6, 8, 10 and 12 respectively. In this set of experiments, the training dataset is comprised of 2684 "vulnerable" and 2684 "not vulnerable" functions, whereas the test dataset contains 335 "vulnerable" and 335 "not vulnerable" functions. Then we built a CNN model in the TensorFlow environment with the training dataset and evaluated it with the test dataset. Since the constructed dataset is balanced, including the same number of samples for both "vulnerable" and "not vulnerable" classes, we generated ROC (Receiver Operating Characteristic) curves for performance evaluation. Figure 6 shows our result as the true positive and false positive ratio for different binary AST depths that are used as a threshold to cut complete binary ASTs. As seen in Figure 6, our vulnerability prediction model performs better for AST depth of 8, while its performance degrades as the depth decreases (e.g. for depth 6) or increases (e.g., for depth 10 and 12). The corresponding AUC (Area Under Curve) values for these curves are 0.816, 0.854, 0.687 and 0.825 respectively for depth 6, 8, 10 and 12. This behaviour we observed may seem counter-intuitive as one might expect better results for the highest AST depths (e.g. depth 10 and 12) since they carry more information with respect to relatively lower depths (e.g. depth 6 and 8). However, in machine learning, there is a phenomena known

TABLE 2. The types of vulnerabilities investigated in the experimental work.

CWE ID	CWE Description
119	Improper Restriction of Operators within the Bounds of a Memory Buffer
120/121/122	Buffer Overflow
469	Use of Pointer Subtraction to Determine Size
476	NULL Pointer Dereference
20,457,805 etc	Improper Input Validation, Use of Uninitialized Variable, Buffer Access with Incorrect Length Value etc.

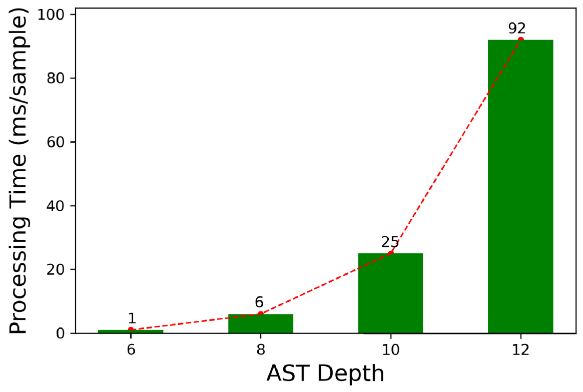

FIGURE 7. Processing time per sample during training for the AST depths of 6,8,10 and 12 resulting in the numerical array size of 381, 1533, 6141, and 24573 respectively.

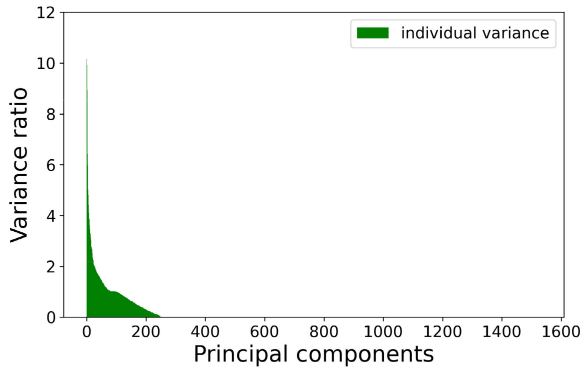

FIGURE 8. Feature importance based on principal component analysis.

as the *curse of dimensionality* which requires to increase the size of the training set exponentially with the data dimension to preserve predictive power of the model [36]. Otherwise, with a fixed number of training samples, the predictive power of a model initially increases as the data dimension (i.e. the number of features) grows, but then begins to decrease when the dimension reaches at a specific point [37]. Therefore, the exponential growth in the dimension of input data is the major reason for degradation in ML performance for higher AST depths because the learning becomes more difficult for the model and requires more samples as the input size grows. From this, we conclude that it is feasible to cut a complete binary AST at certain depth instead of considering the whole tree in some cases.

We also observed the impact of AST depth on training time as depicted in Figure 7. For the depth of 6, 8, 10 and 12, the corresponding processing time during training is 1, 6, 25, and 92 ms/sample respectively, on a 64-bit Windows-10 PC with an Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz processor and 32.0 GB RAM. The processing time grows exponentially as the AST depth increases, which is an expected outcome since the size of numeric array representation of code snippets grows exponentially as well.

D. DIMENSION REDUCTION

As illustrated for the *main* function in Step-7 in Section IV, there may be too many 0 (zeros) in numeric array representation due to NULL nodes in the complete binary AST. Considering the whole dataset, if the resulting array representations of most functions contain zero or the same value at the same columns, then these columns will have a less or negligible impact on the ML model. To reveal the most important features and eliminate ineffective features, we measured variance for each column and determined principal components. Figure 8 shows principal components for the depth of 8. As seen in this figure, around 250 features out of 1533 have a considerable effect on the ML model. From this, we can reduce the array size from 1533 to 250, which provides an advantage for the training time of the ML model. According to our experimental analysis, this process reduces training time by about 68%.

E. PERFORMANCE EVALUATION

As justified in Section V-C, we take partial binary AST up to depth 8 in the experiments presented in this part. We prepared two different experimental settings, as explained in the following parts.

1) IMBALANCED DATASET

In this set of experiments, we used the dataset given in Table 3. As seen in this table, there is a massive difference between the number of positive (i.e., vulnerable) and negative (i.e., not vulnerable) samples, which makes the dataset highly imbalanced. It is inherently more challenging to achieve good performance results with highly imbalanced datasets in ML applications. For this reason, in performance evaluation, we have to take into consideration the ratio between the number of positive and negative samples, which determines the baseline.

We realized our implementation using the Scikit-learn [34] library to built a multi-label classifier. Thus, when our ML model tests a source code fragment, it simultaneously analyses the code for all categories of vulnerabilities. In other words, there is no need to develop and train a separate

TABLE 3. The number of positive and negative samples in imbalanced dataset that are obtained by under-sampling the original dataset.

Class	Training (# of samples:46093)		Validation(# of samples:5736)		Test(# of samples:5765)	
	Vulnerable	Not Vulnerable	Vulnerable	Not Vulnerable	Vulnerable	Not Vulnerable
CWE119	2684 (5.82%)	43409 (94.18%)	335 (5.84%)	5401 (94.16%)	355 (6.16%)	5410 (93.84%)
CWE120+	5119 (11.10%)	40974 (88.89%)	641 (11.18%)	5095 (88.82%)	684 (11.86%)	5081 (88.14%)
CWE469	323 (0.70%)	45770 (99.30%)	36 (0.63%)	5700 (99.37%)	32 (0.56%)	5733 (99.44%)
CWE476	1160 (2.52%)	44933 (97.48%)	146 (2.55%)	5590 (97.45%)	140 (2.43%)	5625 (97.57%)
CWEOther	3294 (7.15%)	42799 (92.85%)	419 (7.30%)	5317 (92.70%)	399 (6.92%)	5366 (93.08%)

ML model for different vulnerability categories, which is an advantage in terms of training time, processing power, and memory requirements.

We preferred to use the Multi-Layer Perceptron (MLP) algorithm because it has multi-label classification capability and yielded better results with respect to other ML algorithms. We obtained many results by changing hyperparameters of the model, such as by setting hidden layer size to 5, 10, 20, 50, and 100. We observed that the ML model showed similar performance for the hidden layer size 5 and 10, whereas its performance degraded when the hidden layer size was set to 20, 50, or 100. This indicates that the model becomes overfitted for higher hidden layer sizes. Figure 9 shows individual Precision-Recall (P-R) curves for each vulnerability class when the hidden layer size is 5. From the figure, we can infer that our model performs better for certain vulnerabilities than others. For example, it is seen that CWE476 shows the best performance as almost reaching the F1=0.6 curve when the precision and the recall are around 0.701 and 0.521, respectively, resulting in the area under the curve (AUC) of 0.528. On the other hand, CWE469 demonstrates the worst performance yielding the F1=0.090 with AUC of 0.031. The P-R curves for the other three classes (i.e., CWE119, CWE 120+, CWEother) lay between CWE476 and CWE469 reaching the F1 scores of 0.509, 0.427 and 0.270 respectively. The area under curve for the micro-average P-R curve (i.e., the average performance for all categories) is 0.377 as passing the F1=0.4 curve. There is a similar trend in CWE-based performance evaluation in [16]. Notice that, while interpreting this figure, we must take the ratio of positive and negative samples into consideration because it determines the baseline. Therefore, predicting vulnerabilities for CWE469 is much more challenging because it has a much higher imbalance in the dataset with respect to other categories, as seen in Table 3, which is the main reason for its poor performance. Regarding the performance of CWEother, considering that the category CWEother is comprised of multiple different vulnerability types such as CWE-20, CWE-457, CWE-805 and more, as in the original dataset, it may become more challenging for the ML model to build a common classifier for all these different vulnerability types, and this could be one reason for its relatively worse performance.

2) BALANCED DATASET

In this set of experiments, we undersampled the imbalanced dataset given in the previous part to generate a balanced subset that allows us to make a comparison with the ROC

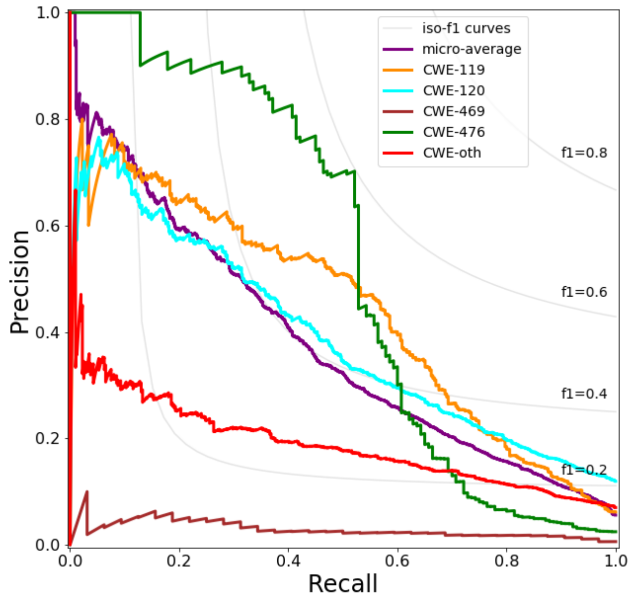

FIGURE 9. P-R curves for different classes for the imbalanced dataset.

curves given in [16]. Performing performance comparison on balanced dataset is important as it gives insight to what extent the targeted vulnerabilities can be detected under similar conditions. To undersample the dataset, we selected the same number of "vulnerable" and "not vulnerable" functions for each particular class. For example, as seen in Table 3, there are 2684, 335, and 355 "vulnerable" functions for the vulnerability type of CWE119 in the training, validation, and test datasets respectively. To generate a balanced set for this class, we kept all of the "vulnerable" functions and randomly selected the same amount of "not vulnerable" functions (e.g. yielded a training dataset containing 2684 "vulnerable" and 2684 "not vulnerable" functions for CWE119 class). A similar procedure was applied for other classes as well, and balanced datasets were obtained that contained the same number of "vulnerable" and "not vulnerable" functions.

We performed experiments using CNN algorithm on the Tensorflow with the hyperparameters given at Table 4. For each vulnerability type, we trained separate ML model on the associated balanced dataset. Figure 10 shows ROC curves when binary AST is taken up to depth level 8 for different vulnerability types. As it can be seen from the Figure 10, the trained models perform well compared to "No skill" (random decision) for all classes. These results show that the proposed source code representation method is capable of

TABLE 4. The hyper-parameters of the CNN algorithm in the balanced dataset experiment.

Hyperparameter	Value
Optimizer	adam
Number of filters	32
Kernel size	9
Number of hidden nodes	200
Dropout Rate	0.5
Batch Size	250
Activation	relu
Loss	sparse categorical crossentropy
Number of epochs	10

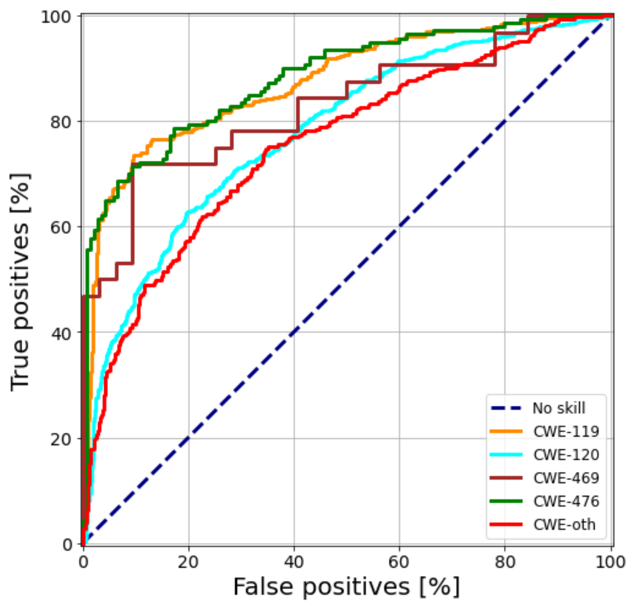

FIGURE 10. ROC curves for different classes for the balanced dataset.

characterizing source code for vulnerability prediction. The corresponding AUC values for the curves in Figure 10 are 0.874, 0.778, 0.829, 0.882 and 0.755 for vulnerability types of CWE-119, CWE-120, CWE-469, CWE-476 and CWE-other respectively. According to Figure 10, the vulnerability prediction for CWE-119 and CWE-476 classes perform better than the remaining classes, which implies that detecting these two vulnerability categories is easier than the other categories, yet the performance of other classes is encouraging as well. This also matches the experimental results obtained in the previous section where the model gave the best results for CWE-119 and CWE-476 classes. It is interesting to observe that CWE-120 and CWE-other, the two vulnerability classes that actually contain more than one vulnerability type as explained in Table 2, perform worse than the other three categories. As described in Table 2, CWE120+ vulnerability data consists of CWE120, CWE121 and CWE122 vulnerability functions and similarly CWEother data contains CWE20, CWE457, CWE805 and some other vulnerability types. Containing multiple vulnerability types may be a reason for their relatively bad performance because constructing a common classifier for all of them should be more difficult.

In comparison with the results in [16], the ROC curves in [16] may seem to outperform our results; however, notice that the ROC curves in [16] are obtained from the SATE IV dataset, which contains simple synthetic functions. On the other hand, our ROC curves are based on real-world functions collected from open-source projects as described in Section V-A, and therefore our case is more challenging with respect to the situation in [16].

3) COMPARISON WITH THE Code2vec

We also compared our presented method with the code2vec [20], which is a state-of-art code representation method. The code2vec uses a neural network model to represent a code snippet as a single fixed-length code vector, which can be used to predict semantic properties of the snippet. To this end, the method first decomposes the code into a collection of paths in its AST, called path contexts, and then the network learns the atomic representation of each path contexts while simultaneously learning how to aggregate a set of them. The authors demonstrate the effectiveness of the code2vec by using it to predict a method's (i.e. function's) name from the vector representation of its body.

One major difference between the code2vec and our presented method is that a *distributed* representation technique with neural networks is employed in the code2vec to generate vector representation of code snippets, whereas a kind of deterministic rule-based representation is used in our method. Thus, the generated vector representation of code fragments in our proposed method becomes *static* in the sense that it does neither change under varying conditions nor depend on other code samples in a given dataset, while the resulting codevector in the code2vec approach is *dynamic* meaning that it may change under different settings such as being considered with different datasets for different objectives. Another significant difference between the two approaches is that the code2vec method assigns different weights (i.e. attention) to certain subpaths of AST depending on their importance in prediction, on contrary to our presented approach which takes a certain part of AST into consideration depending on depth and assigns equal importance to its elements.

To make performance comparison with the code2vec for vulnerability prediction, we used two open source implementations called *code2vec*[4] and *astminer*[5] [38]. The former requires the latter to extract path context of the C codes considered in our own work, as the publicised implementation of the code2vec currently supports only Java and *C#* as the input languages. Because the public implementation of the code2vec is designed to predict function names, we needed to convert the problem of name prediction to the problem of vulnerability prediction. For this purpose, we tried alternative experiment settings as follows: (i) **(Code2vec multilabel)** We replaced original function names with new multi-word names (i.e. a tuple of 5 different words corresponding to

[4]https://github.com/tech-srl/code2vec

[5]https://github.com/JetBrains-Research/astminer

TABLE 5. Performance comparison of the presented method with the code2vec.

Category \ Method	Code2vec			Code2vec+MLP			Proposed Method		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
CWE119	0.053	0.040	0.046	0.065	0.873	0.120	0.504	0.515	0.509
CWE120+	0.110	0.064	0.081	0.131	0.790	0.225	0.415	0.440	0.427
CWE469	0.0	0.0	0.0	0.009	0.188	0.017	0.060	0.187	0.090
CWE476	0.043	0.037	0.040	0.024	1.0	0.047	0.701	0.521	0.598
CWEOther	0.108	0.077	0.090	0.085	0.547	0.148	0.218	0.353	0.270

5 different vulnerability types) such that each element in the tuple can take one of the two possible words, one of which indicates the existence of a specific vulnerability type and the other implies the opposite. Thus, after replacing the function names in this way, predicting functions' names by using the code2vec corresponds to the prediction of vulnerabilities likewise our multi-label binary classification task. (ii) **(Code2vec single label)** We replaced original function names with a new single-word label that is chosen differently for each vulnerability category and takes one of the two possible words depending on whether the function is vulnerable or non-vulnerable. In this case, we performed prediction task separately by using the code2vec for each vulnerability category in disjoint processes. (iii) **(Code2vec + MLP)** After applying item (i), we extracted the corresponding code vector representation of the functions according to the code2vec, which is a numeric array containing 384 elements, and then used them in an additional ML algorithm (i.e. MLP) for vulnerability prediction in a similar manner to our imbalanced dataset experiment.

We performed performance comparison of these approaches experimentally with our own method on the prediction of vulnerabilities on the imbalanced dataset shown in Table 3. Since we observed that (i) and (ii) performed similarly, we included only the experimental results of (i) and (iii) in comparison with our presented method. Table 5 contains performance evaluation results for Approach (i) (i.e. code2vec), Approach (iii) (i.e. code2vec+MLP) and our own method. According to the results of code2vec, the best $F1$ value was obtained as 0.09 for category CWEOther, while all other F1 values are less than this value. It is even zero for category CWE469, which means the algorithm did not find any one of the vulnerable functions. On the other hand, it seems that applying MLP on code vectors (code2vec+MLP) improved performance to some extend, resulting in F1 values between 0.017 (CWE469) and 0.225 (CWE120+). Finally, it is seen from Table 5 that our method performs better than the code2vec approaches, resulting in F1 values between 0.093 (CWE469) and 0.593 (CWE476).

One reason for the poor performance of the code2vec might be that it is not designed to deal with binary classification problems on highly imbalanced datasets, which require to accurately detect positive test samples constituting a very small portion of the whole test samples, while not raising false alarms for negative samples. For example, in the code2vec approach, it is considered a success to some extend when a method's name is predicted accurately at second rank, which would be a complete false prediction in a binary classification case. On the other hand, the reason for superiority of our presented method is probably the fact that the resulting vector representation in our presented approach retains information about structural relations among AST nodes thanks to both categorical encoding of AST tokens and their consistent mapping to final vector correspondence, which helps the used ML algorithm to reveal hidden patterns indicating targeted vulnerabilities.

Our method also advantageous compared to code2vec method with respect to run-time of algorithms. It takes on average 30 seconds for our proposed method to train the model while each training epoch of code2vec takes 147 seconds on average. All the experiments for our proposed method are performed on a 64-bit Windows-10 PC with an Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz processor and 32.0 GB RAM. Since code2vec developers provided their instructions for Ubuntu systems, we tested code2vec on a Ubuntu 18.04 installed on a virtual machine that utilizes 24022 MB base memory and 4 processors and runs on the same windows computer described here.

VI. DISCUSSION

In our existing implementation for encoding AST tokens to numerical tuples, we used a tuple of 3 numeric values for each different token, where the first number corresponds to token type while second and third numbers in the tuple are reserved for embedding auxiliary information related to the token as described in Section IV. In doing so, we aimed to transfer as much information as possible from AST to numeric array representation. However, using 3 numbers for one token makes the resulting array size 3 times more extended with respect to 1:1 mapping, which increases space and time requirements in ML operations. An alternative approach might be using tuples of 2 numeric values or using only 1 number in a greater range covering all possible variants of tokens. This would result in relatively shorter numeric arrays with respect to an existing case; however, it might also be less effective in preserving information embedded into AST. To shed light on this issue, we also implemented 1:1 encoding (i.e. mapping 1 token to 1 numeric value) in addition to existing 1:3 encoding (i.e. mapping 1 token to a tuple of 3 numeric values) and performed similar experiments. We did not observe any significant performance differences when implementing 1:1 mapping in the encoding step.

Another topic we examined is whether employing sparse matrix representation in resulting numerical array is useful or not. More specifically, instead of encoding NULL nodes in a complete binary AST, only extracted tokens could be encoded with additional location indexes to take into consideration of their respective locations in the tree. This could significantly reduce the size of the resulting numerical array representation. Our experimental investigations revealed that sparse matrix representation has some negative side effects, which degrade the performance of the ML model in vulnerability prediction. This may be due to the fact that inserting extra parameters in the encoding like the respective location index of AST tokens results in disruptive effect, but this issue needs more elaboration.

VII. CONCLUSION AND FUTURE WORK

Automated intelligent software security analysis is an important topic that gathered great interest lately. In this context, we first proposed a source code representation method that is capable of characterizing source code into a proper format for further processes in ML algorithms. The presented method extracts and then converts AST of a given source code fragment into a numerical array representation while preserving structural and semantic information contained in the source code. Thus, it enables us to perform ML-based analysis on source code through resulting numeric array representation. We implemented our approach and experimentally investigated the problem of vulnerability prediction from source code using ML as benefiting from a public vulnerability dataset containing a large amount of real function-level source code mined from several open-source projects and carefully labeled according to examinations of multiple static code analysis tools. We conducted many experiments under different settings with the objective of predicting 5 different predetermined vulnerability types and achieved promising and encouraging results compared to state-of-art methods.

As a future work, it would be interesting to examine the presented source code representation technique for different objectives rather than vulnerability prediction, such as similarity analysis and code completion. Another direction could be to research how to improve localization and interpretation aspects of the vulnerability prediction (i.e. where exactly is a detected vulnerability in a function-level code and why is it detected as a vulnerability). Furthermore, it would be of interest to explore possible ways of leveraging transfer learning techniques with the presented method to apply a model trained on a certain language to other languages.

REFERENCES

[1] R. Telang and S. Wattal, "An empirical analysis of the impact of software vulnerability announcements on firm stock price," *IEEE Trans. Softw. Eng.*, vol. 33, no. 8, pp. 544–557, Aug. 2007.

[2] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011.

[3] R. S. Ross, "Information security," Joint Task Force Transformation Initiative, Guide Conducting Risk Assessments, NIST Special Publication, Gaithersburg, MD, USA, Tech. Rep. 800-30 Revision 1, 2012.

[4] A. M. Delaitre, B. C. Stivalet, P. E. Black, V. Okun, T. S. Cohen, and A. Ribeiro, "Sate V report: Ten years of static analysis tool expositions," NIST, Gaithersburg, MD, USA, Tech. Rep. SP-500-326, 2018.

[5] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Reading, MA, USA: Addison-Wesley, 2006.

[6] M. Jimenez, "Evaluating vulnerability prediction models," Ph.D. dissertation, Dept. Sci., Technol. Commun., Univ. Luxembourg, Rue Mercier, Luxembourg, Oct. 2018.

[7] T. Abraham and O. de Vel, "A review of machine learning in software vulnerability research," Cyber Electron. Warfare Division, Dept. Defense, Austral. Government, Edinburgh, SA, Australia, Tech. Rep. DST-Group-GD-0979, 2017.

[8] B. McCorkendale, X. F. Tian, S. Gong, X. Zhu, J. Mao, Q. Meng, G. H. Huang, and W. G. E. Hu, "Systems and methods for combining static and dynamic code analysis," U.S. Patent 8,726,392, May 13, 2014.

[9] E. Ustundag Soykan, Z. Bilgin, M. A. Ersoy, and E. Tomur, "Differentially private deep learning for load forecasting on smart grid," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, Dec. 2019, pp. 1–6.

[10] Z. Bilgin, E. Tomur, M. A. Ersoy, and E. U. Soykan, "Statistical appliance inference in the smart grid by machine learning," in *Proc. IEEE 30th Int. Symp. Pers., Indoor Mobile Radio Commun. (PIMRC Workshops)*, Sep. 2019, pp. 1–7.

[11] M. Craglia, Ed., A. Annoni, P. Benczur, P. Bertoldi, P. Delipetrev, G. De Prato, C. Feijoo, E. F. Macias, E. Gomez, M. Iglesias, H. Junklewitz, M. L. Cobo, B. Martens, S. Nascimento, S. Nativi, A. Polvora, I. Sanchez, S. Tolan, I. Tuomi, and L. V. Alujevic, "Artificial intelligence—A European perspective," Publications Office, Rue Mercier, Luxembourg, Tech. Rep. EUR 29425 EN, 2018.

[12] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," 2018, *arXiv:1808.01400*. [Online]. Available: http://arxiv.org/abs/1808.01400

[13] L. Chen, W. Ye, and S. Zhang, "Capturing source code semantics via tree-based convolution over API-enhanced AST," in *Proc. 16th ACM Int. Conf. Comput. Frontiers*, Apr. 2019, pp. 174–182.

[14] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 783–794.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, New York, NY, USA: Curran Associates, 2019, pp. 10197–10207. [Online]. Available: http://papers.nips.cc/paper/9209-devign-effective-vulnerability-identification-by-learning-comprehensive-program-semantics-via-graph-neural-networks.pdf

[16] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.

[17] *Debian—The Universal Operating System*. Accessed: Jul. 4, 2020. [Online]. Available: https://www.debian.org/

[18] *Github—Distributed Version Control Software*. Accessed: Jul. 4, 2020. [Online]. Available: https://github.com/

[19] P. E. Black and P. E. Black, *Juliet 1.3 Test Suite: Changes From 1.2*. Gaithersburg, MD, USA: US Department of Commerce, National Institute of Standards and Technology, 2018.

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," in *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019, pp. 1–29, doi: 10.1145/3290353.

[21] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th ACM Conf. Comput. Commun. Secur. CCS*, 2007, pp. 529–540.

[22] S. Neuhaus and T. Zimmermann, "The beauty and the beast: Vulnerabilities in red hat's packages," in *Proc. USENIX Annu. Tech. Conf.*, 2009, pp. 1–14.

[23] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. FSE*, 2014, pp. 257–268, doi: 10.1145/2635868.2635880.

[24] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," in *Proc. 4th ACM Int. Workshop Secur. Privacy Analytics IWSPA*, 2018, pp. 31–39.

[25] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 23–33.

[26] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 837–847.

[27] V. Efstathiou and D. Spinellis, "Semantic source code models using identifier embeddings," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 29–33.

[28] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. ESEC/FSE*, 2019, pp. 695–705, doi: 10.1145/3338906.3338941.

[29] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, "DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection," *IEEE Trans. Fuzzy Syst.*, vol. 28, no. 7, pp. 1329–1343, Jul. 2020.

[30] B. Chess and J. West, *Secure Programming With Static Analysis*. London, U.K.: Pearson, 2007.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.

[32] *Pycparser—Parser for the C Language*. Accessed: Jul. 4, 2020. [Online]. Available: https://github.com/eliben/pycparser

[33] *Algodaily—Algorithm for Encoding M-Ary Tree to Binary Tree*. Accessed: Jul. 4, 2020. [Online]. Available: https://github.com/calvinchankf/AlgoDaily

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[35] M. Abadi *et al.* (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software. [Online]. Available: https://www.tensorflow.org/

[36] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2014.

[37] G. V. Trunk, "A problem of dimensionality: A simple example," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-1, no. 3, pp. 306–307, Jul. 1979.

[38] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, "PathMiner: A library for mining of path-based representations of code," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 13–17.

ZEKI BILGIN (Member, IEEE) received the B.S. and M.S. degrees in electrical and electronics engineering from Gazi University, Ankara, and the Ph.D. degree in computer science from The City University of New York, NY, USA. He is currently working as an Experienced Security Researcher at Ericsson Research, Turkey. He was involved in many industrial and academic research projects related to telecommunications, the Internet of Things (IoT), 5G, cybersecurity, social computing, computer vision, smart grids, and machine learning, and authored many scientific articles and inventions in these domains.

MEHMET AKIF ERSOY received the B.S. and M.S. degrees in computer engineering from Boğaziçi University, in 2012 and 2015, respectively, where he is currently pursuing the Ph.D. degree. He had worked as a Researcher at The Scientific and Technological Research Council, National Cryptology Institute, for three years. He is currently working as an Experienced Security Researcher at Ericsson Research, İstanbul, Turkey, where he joined, in October 2018. He has several years of experience as a Software Developer. He has authored some international conference papers related to networks, parallel programming, and security.

ELIF USTUNDAG SOYKAN received the M.S. degree in computational science and engineering from Istanbul Technical University, in 2005, where she is currently pursuing the Ph.D. degree. She had worked at The Scientific and Technological Research Council, National Cryptology Institute, for 13 years in security domain. She joined Ericsson Research, in 2018, as a Senior Security Researcher. She has published several papers in international conferences, mostly on information security and privacy. Her research interests include ML/AI security, privacy enhancing technologies, and the Internet of Things (IoT) security.

EMRAH TOMUR received the B.S. and M.Sc. degrees in electronics engineering from Bilkent University, in 1999 and 2001, respectively, and the Ph.D. degree in information systems from Middle East Technical University, in 2008. He has been working as a master's Researcher at Ericsson, since January 2019, where he is currently leading the research team in the area of security. Before joining Ericsson, he worked as a Research and Development Manager in private sector companies and the Technology Transfer Manager in universities, where he gave courses and served as a Graduate Thesis Advisor. He has several scientific research papers published in journals and conference proceedings in the area of security. He also worked in numerous various national and international research and development projects funded by EU or national agencies. His technical expertise is on security of the Internet of Things, M2M, and wireless sensor networks.

PINAR ÇOMAK was born in Ankara, Turkey. She received the B.Sc. degree in mathematics and the M.Sc. and Ph.D. degrees in cryptography from Middle East Technical University (METU), in 2010, 2013, and 2020, respectively. She had worked as a Research Assistant at METU for eight years. She joined Ericsson Research, İstanbul, Turkey, in September 2019, where she has been working as an Experienced Security Researcher. She has authored some international conference papers related to coding theory, computational algebra, and cryptography.

LEYLI KARAÇAY was born in Tabriz, Iran, in 1983. She received the B.Sc. degree in software engineering from IAU, Qazvin, Iran, in 2006, and the M.Sc. and Ph.D. degrees in computer science and engineering from Sabanci University, in 2012 and 2020, respectively. She had worked as a Teaching Assistant at Sabanci University, for seven years. She joined Ericsson Research, İstanbul, Turkey, in October 2019, where she has been working as a Security Researcher. She has some scientific research papers published in journals and conference proceeding in the area of security.