information

Article

There Are Infinite Ways to Formulate Code: How to Mitigate the Resulting Problems for Better Software Vulnerability Detection †

Jinghua Groppe [1,*], Sven Groppe [1,*], Daniel Senf [2] and Ralf Möller [1]

[1] Institute of Information Systems (IFIS), University of Lübeck, Ratzeburger Allee 160, 23562 Lübeck, Germany
[2] Lufthansa Industry Solutions AS GmbH, Schützenwall 1, 22844 Norderstedt, Germany
* Correspondence: jinghua.groppe@uni-luebeck.de (J.G.); sven.groppe@uni-luebeck.de (S.G.)
† This article is an extended version of the paper entitled "Variables are a Curse in Software Vulnerability Prediction" presented at the 34th International Conference on Database and Expert Systems Applications (DEXA 2023), Panang, Malaysia, 28–30 August 2023.

Abstract: Given a set of software programs, each being labeled either as vulnerable or benign, deep learning technology can be used to automatically build a software vulnerability detector. A challenge in this context is that there are countless equivalent ways to implement a particular functionality in a program. For instance, the naming of variables is often a matter of the personal style of programmers, and thus, the detection of vulnerability patterns in programs is made difficult. Current deep learning approaches to software vulnerability detection rely on the raw text of a program and exploit general natural language processing capabilities to address the problem of dealing with different naming schemes in instances of vulnerability patterns. Relying on natural language processing, and learning how to reveal variable reference structures from the raw text, is often too high a burden, however. Thus, approaches based on deep learning still exhibit problems generating a detector with decent generalization properties due to the naming or, more generally formulated, the vocabulary explosion problem. In this work, we propose techniques to mitigate this problem by making the *r*eferential structure of variable references explicit in input representations for deep learning approaches. Evaluation results show that deep learning models based on techniques presented in this article outperform raw text approaches for vulnerability detection. In addition, the new techniques also induce a very small main memory footprint. The efficiency gain of memory usage can be up to four orders of magnitude compared to existing methods as our experiments indicate.

Keywords: software security; software vulnerability; deep learning; 3-property encoding; variable name dependence; abstract syntax graph

Citation: Groppe, J.; Groppe, S.; Senf, D.; Möller, R. There Are Infinite Ways to Formulate Code: How to Mitigate the Resulting Problems for Better Software Vulnerability Detection. *Information* **2024**, *15*, 216. https://doi.org/10.3390/info15040216

Academic Editor: Aneta Poniszewska-Maranda

Received: 11 March 2024
Revised: 5 April 2024
Accepted: 7 April 2024
Published: 11 April 2024

1. Introduction

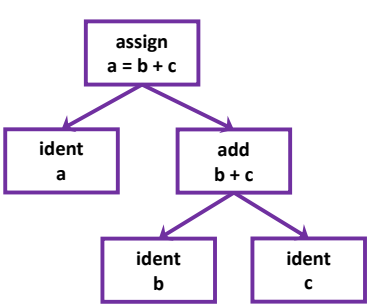
Software security is an important factor in facing cyber-attacks. However, detecting code vulnerabilities to guarantee software security is still a daunting task. Traditional approaches [1–9] to vulnerability detection, for instance, rely on manually designed vulnerability rules, which are typically highly complex and also very difficult to define due to the diversity and complexity of vulnerabilities in application programs. In contrast, deep learning (DL) technology can learn complicated patterns hidden in data, in this case programs, and therefore provides a promising solution to automatically learn vulnerability rules.

A significant amount of research work has been dedicated to applying DL to predict the vulnerabilities of software code. Despite these efforts, DL-based approaches have not achieved a final breakthrough in this field and still have a limited capability to distinguish vulnerable from non-vulnerable code [10]. We could perhaps explain the lack of success

like this: A program has complicated structures that contain various syntactical and semantic dependencies. If it contains a vulnerability, the vulnerability only involves a tiny part of the complex program structure. Consequently, it is very challenging to learn vulnerability features, which are submerged in a plethora of dependencies. However, deep learning is well known for its ability to learn complex patterns, and we have witnessed its great success in natural language processing [11], speech recognition [12] and computer vision [13]. Therefore, we have to ask ourselves why DL-based approaches have not yet learned vulnerability patterns well.

To answer this question, we must investigate exactly how DL approaches work. Currently, DL approaches, either graph-based [10,14] or unstructured ones [15–18], borrow methods used in natural language processing to define the semantics of the textual program code or nodes in a graph of the code. The full code or a piece of the code is considered as plain text like a natural language and it is first split into a sequence of tokens, and each token is represented by a real-valued vector (called embedding representation). Unstructured approaches learn the representation of the code based solely on the sequence of the tokens. The sophisticated graph-based approaches learn a presentation based on the tokens appearing in each node and the relations between nodes.

Let us consider a very simple example: We want to program the summation of two variables, and we can write the code as $a = b + c$. From the text of this code, we can create its graph structure, tokens and their embeddings as illustrated in Figure 1. A representation of the code can be learned based on these embeddings and the relations of nodes. We can use other variable names to express the same functionality, such as $x = y + z$, $c = e + f$, or $a = a1 + a2$. Since different names have different embeddings, DL models, which learn based on the raw code text, could only find a representation that is specific to the code text with the used variable names. It is hard for them to capture the intrinsic functionality beyond the diversity of code expression using different variable names and to generalize to new code pieces given name-specific encoding of variables.



Token	Embedding
a	1.288, ...
b	1.449, ...
c	0.066, ...
=	-0.764, ...
+	-1.092, ...
assign	0.031, ...
ident	-1.022, ...
add	-1.436, ...

Figure 1. The graph structure and embeddings of the summation functionality coded as $a = b + c$, from which a representation specific to the code text will be learned.

In fact, one can use any combination of letters, digits, and other symbols to name a variable as long as the combination conforms to the syntax rules. This means that there are numerous possibilities to code text representations of equivalent expressions. It would be very difficult for DL models to capture patterns hidden in an infinite number of equivalent expressions, and, hence, we have to solve the problem of the infinity. We can think about two solutions to this issue:

Collecting enough training data. DL models have the ability to capture the intrinsic patterns hidden under the variety of equivalent expressions if we provide them with enough training data, which covers all the text representations of equivalent expressions. The question is how we can collect data that covers an unlimited number of text representations. Apart from the fact that different variable names could express the same semantics, the same variable names could express different semantics. What makes things worse is that it may be random whether two different variables express the same semantics in different contexts, or whether two identical variable names have different semantics. Although

Motzkin [19] and Ramsey theory [20] have pointed out that pure randomness is impossible and this is especially true in the choice of variable names, generally we can consider variable names as a random event. The occurrence of random events does not show any patterns, and a DL model will, therefore, not be able to discover a pattern from a random event.

Mitigating the infinity and randomness. We could not obtain a well-generalized model in the presence of an infinite number and randomness of text code of the same functionality, and we, therefore, need solutions to address these challenges. Ideal solutions should transform an infinite number of text representations into a finite number, and unfortunately, we have not found such a solution. The solution we propose in this work is to mitigate the infinity and randomness by removing the names of variables, which are a major cause of this problem. Concretely, we suggest a new edge type of name dependence and a type of abstract syntax graph (ASG) extending the abstract syntax tree (AST) with the edges of name dependence and develop a 3-property node encoding scheme based on the ASG. These techniques allow us to remove variable names from code but retain the semantics of the code, and thus greatly mitigate the semantic uncertainty of variables and the diversity of text code of a functionality. The empirical evidence presented later shows that our techniques help DL models to better learn the intrinsic functionality of the software and improve their prediction performance. The evidence is especially strong over the extremely imbalanced training dataset Chromium+Debian, which contains only 592 (6.92%) samples with vulnerability.

The rest of this paper is organized as follows: We first review related work in Section 2 and present our techniques of how to deal with variable names in program code in Section 3. These techniques are evaluated and compared with existing approaches in Section 4. Section 5 summarizes and concludes this work.

2. Related Work

This work is based on the work [21] and extends it with at least 60% new content, where we conduct a comprehensive experimental evaluation, perform an extensive and in-depth analysis of the techniques and evaluation results, investigate the problem of vocabulary explosion, and explore the quality of the data and the representations learned by the techniques and models proposed in this work.

Deep learning-based software vulnerability detection, both unstructured [15–18] and structure-based [10,14,22–24], currently adopt the raw text representations of code to describe the semantics of code. Approaches to unstructured data treat source code as plain text, and the text is split into a sequence of tokens, which is used as the representation of the code. This representation only describes a flat structure, which does not directly reflect the syntactical and semantic dependencies. The structure-based approaches address the limitation using the structure information of program code. Among them, graph-based models [10,14] are more sophisticated because they can directly work with graph data. However, since they still use pieces of raw code as the feature of nodes, these models could learn only a representation specific to the code text, not its intrinsic functionality. Models based on a representation specific to the original code text require much more training data to abstract away variable names in code.

Inspired by the success of pre-trained models in natural language processing (NLP), especially BERT [25], several pre-trained models like RoBERTa [26], CuBERT [27], CodeBERT [28], GraphCodeBERT [29] and SyncoBERT [30] were recently developed to learn representations of code. The first three approaches treat source code as plain text and do not support syntactic and semantic dependencies, whereas GraphCodeBERT and SyncoBERT are structure-based ones, which additionally introduce a variable relation graph and an AST, respectively. Specifically, the unstructured models use source code and text documents (code, document) as input data, while the GraphCodeBERT and SyncoBERT require three components (code, document, variable relation graph) and (code, document, AST), respectively, as input data to learn code representations. The variable relation graph is created by first parsing the source code to an AST and then extracting a variable sequence from

the AST. GraphCodeBERT outperformed other models for the downstream tasks of clone detection, code translation and code refinement. SyncoBERT exhibited better performance in code search.

As discussed earlier, DL models that use original code text as input require much more training data to abstract away variable names in code. For example, GraphCodeBert is pre-trained over six million code samples and does show such potential. It can be fine-tuned for specific tasks after pre-training requiring less training data. Nevertheless, for each supported programming language, the data for pre-training are immense. A big advantage of our techniques is that they require small quantities of training data even for the pre-training (and do not need fine-tuning): There are only a few real-world vulnerability datasets available, which contain very few vulnerable code examples from less than 1000 to 32,000, which are all that we require for pre-training DL models based on our techniques. Furthermore, our techniques have a very low memory footprint, as shown in the evaluation part. The work [31] extends GraphCodeBERT with the two mechanisms of k-nearest neighbor and contrastive learning and is evaluated over QEMU+FFmpeg code samples using the F1 metric. The evaluation results show that [31] achieved slightly better F1-scores (69.43% for the single FFmpeg dataset and 72.47% for the single QEMU dataset) than ours (62.99% for the original QEMU+FFmpeg dataset that is a collection of QEMU and FFmpeg code samples). However, the QEMU and FFmpeg datasets used in [31] have 9116 vulnerable samples while ours contains only 5865 vulnerable samples. Therefore, we have reason to believe that our techniques are more efficient in comparison to GraphCodeBERT and its variants.

Because of the wide variety of possibilities of naming identifiers, DL models of representation learning [32] also face the problem of vocabulary explosion. In order to address this issue, refs. [15,33] define a set of keywords, such as VAR1, VAR2, FUNC1, FUNC2, and use them to replace the variable and user-defined function names. This method can indeed reduce the number of unique tokens, but at the same time, it could mislead models since the same keywords could have different semantics in different settings.

The code graphs in our work are generated based on the tool developed in [34] and the AST+ used in our work is equivalent to its code property graph (CPG), i.e., adding flow dependence, data dependence, and control flow into AST. CPG was first used in the non-learning approaches to find software vulnerabilities [34,35] and later on was adopted in deep learning-based approaches [10,14] as the input of graph encoding networks. Two widely used libraries for graph encoding networks are PyG [36] and DGL [37], and in this work, we use the latter to implement our models.

In this work, we suggest a new kind of edge of name dependence, and a type of ASG, which adds the edges of name dependence to AST and removes the names of variables from it without changing the semantics of the code. Abstract syntax graphs are not a new concept and many studies have suggested different ASG graphs from AST for different purposes [38–42], and named them differently, either abstract syntax graph, abstract semantic graph or term graph. They are mainly applied in software engineering for graph transformation [38], program optimization [39,40] and code refactoring [41].

3. Mitigating the Infinity and Randomness

In order to help DL models of software vulnerability detection improve their generalization ability, in this section, we propose techniques for how to mitigate the infinity and randomness of text representations of code by getting rid of one of its main causes, i.e., the names of variables.

3.1. Feasibility Analysis

We aim at removing the names of variables in the program code but still maintain its semantics. Let us first analyze the feasibility of this goal by comparing programming languages with natural languages. Each natural language contains a vocabulary, which is a set of words pre-defined in terms of their spelling and meaning. A sentence consists

of words from the vocabulary, and its semantics is determined by its words. When we replace a word with another with a different meaning, the semantics of the sentence will be changed. The opposite should be true in a programming language. The meaning of a variable is determined by the code where it is in, not determined by its name. This means that we can use many other names to replace this variable. This actually implies that the concrete text representations, i.e., the names of variables, are not important. Therefore, we have reason to believe that it is theoretically possible to describe the semantics of code without using the concrete names of variables. Now, we need to find a practical solution to do this.

3.2. Name Dependence for Removing Variables

We will use a type of abstract syntax graph (ASG) to get rid of variable names. Before we present this ASG, we first examine why existing graph representations of code, abstract syntax tree (AST), control flow graph (CFG), data dependence graph (DDG) and control dependence graph (CDG), are not enough to maintain the semantics of a program under the absence of the variable names. Once again, let us use a very simple piece of code presented in Figure 2.

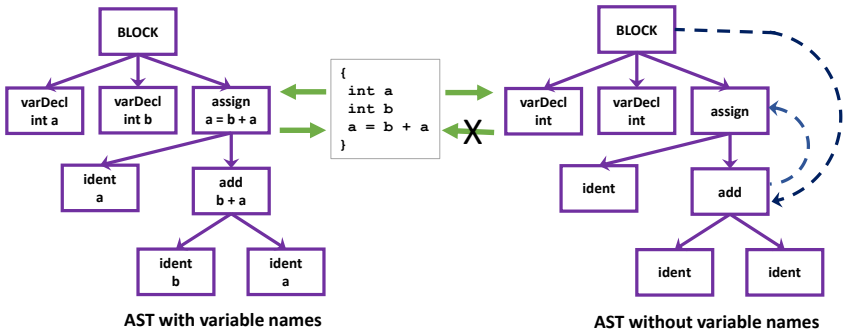


Figure 2. Abstract syntax trees with and without variable names.

The left tree in Figure 2 is a standard AST of the code given in the middle part, which contains complete information about the code. Given this AST, we can revert to its original code exactly. However, we want to mitigate the infinity and randomness of the possible text representations, so we should create an AST without variable names, like the tree on the right side. Unfortunately, the AST will not be able to maintain the semantics of the original code. Adding control flow edges and data dependence as indicated by the dotted lines would not work either. We need a solution and so our abstract syntax graph comes into play.

In programming languages, a variable is related to its declaration (which is either explicitly given or implied). We can determine this relation by the name of the variable. Software engineering uses the term 'dependence' to describe the relations between two components, like data dependence and control dependence. To align with it, we define a new kind of dependence called *Name Dependence* (see Definition 1), to express the relation between a variable and its declaration. With regard to the definition, the name dependence can, hence, be statically determined for programming languages with static name resolution like C, C++, Erlang, Haskell, Java, Pascal, Scheme, and Smalltalk, and only partially for programming languages with dynamic name resolution like some Lisp dialects, Perl, PHP, Python, REBOL and Tcl. In an AST with full information, the name dependence between two nodes can be inferred by the names of variables and identifiers and their lexical scope. When we remove the names of variables and identifiers from the AST, we lose the information on name dependence. Without the information, we will not be able to restore the semantics of the original code.

Definition 1 (Name Dependence). *In a computer program, a variable declaration has a name dependence with an identifier in a statement if the identifier matches the variable declaration according to the name resolution rules of the specific programming language.*

So, we need a way to express the name dependence when names are absent. A solution is to add an edge of name dependence between two related nodes. After adding such edges, the tree structure turns into a graph structure as illustrated in Figure 3, which we call an *abstract syntax graph*. From the graph, we can construct a fragment of code with the exact semantics of the original code, but perhaps with a different text representation, which would not be a problem at all for the task of vulnerability detection.

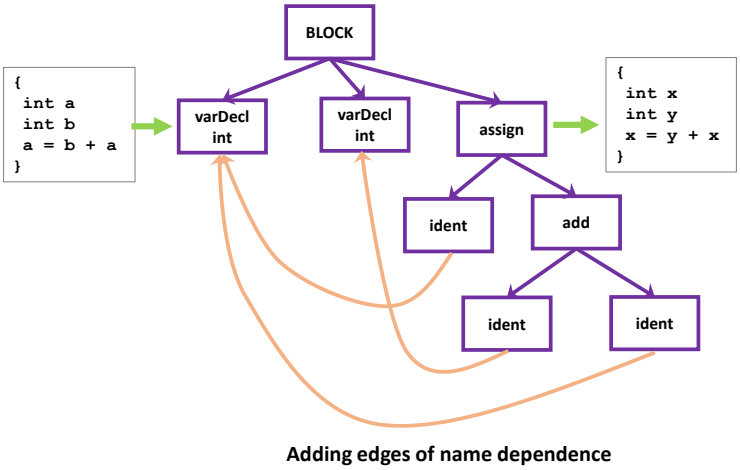


Figure 3. Abstract syntax graph, which expresses the semantics of code without using its variable names.

We can also extend the ASG with the control flow edges and data dependence edges. The extended graph will integrate more information about the behavior of code but also becomes much more complicated. It is obvious that processing more complicated graphs will need more computing resources and time. What we are not sure of is if the complicated graphs can bring more benefits to vulnerability prediction. We know that a code flaw typically involves only a very small part of the graph of code. When the whole code graph becomes more complicated, the ratio of the flaw part to it could become smaller. This means that it would be more difficult to detect it. In our work, we use the ASG and extended ASG to train DL models, and the training results are reported in later sections.

ASTs and ASGs describe the structure information of program code that cannot be reflected in the raw text representations of code. However, since there is not a standardized AST model, different tools would construct different ASTs for the same program code, and so ASGs extended from the ASTs will also be different. Interesting research topics include investigations about how different AST models and corresponding ASGs impact the performance of software vulnerability detection and what is the best AST model for this task.

3.3. Property-Based Node Encoding Scheme

Our ASG provides software programs with a graph representation independent of their text formulation of variable names, and this will help a DL model to learn the 'intrinsic' functionality of code. Apart from the ASG, we further suggest a method to efficiently represent the semantics of nodes in a code graph, *3-property encoding*, which provides a consistent description of the features of nodes and allows DL models to infer the commonalities and differences between nodes easily. This 3-property encoding is developed in the context of our ASG but it can be applied to other code graphs, and it is also programming languages agnostic.

In a code graph, every node represents an executable syntactic construct in code, which can be an expression or a statement or its constituent parts, like variables and constants

(which are, of course, also executable). Currently, the piece of code that consists of the construct (with or without a notation to the construct like 'varDecl' and 'add') is used as the feature of the node. The feature is encoded by first splitting the piece of code into tokens and then averaging the embeddings of all the tokens. The code-based encoding uses the original piece of code to present the feature of a node, and at the same time, the result of encoding blurs the semantics of the original code because of the averaging operation. Our 3-property encoding avoids these two issues by introducing additional information related to the language constructs.

Each language construct has its properties, which may not explicitly appear in the raw code text. Independent of specific programming languages, we found that it is enough to use three properties to describe different constructs: *class*, *name* and *type* of data if any, and each value of the properties will be represented by a unique token. Table 1 demonstrates several common language constructs and their representations with the three properties. With this property-based approach, we can encode all nodes in a consistent way, and this is a very valuable characteristic for many applications. So far, this 3-property encoding has not removed the diversity of text representations and we will further normalize this encoding scheme to mitigate the diversity as much as possible based on the name dependence and ASG.

Table 1. 3-property encoding scheme.

	3-prop. Encoding With Variable Names			**3-prop. Encoding Without Variable Names**		
Construct	**Class**	**Name**	**Type**	**Class**	**Name**	**Type**
int x	varDecl	x	int	varDecl	-	int
if (x $\geq$ 0)	control	if	-	control	if	-
x·0.05	mathOp	mul	-	mathOp	mul	-
fputs(x, stdout)	call	fputs	-	call	fputs	-
x	ident	x	int	ident	var	int
stdout	ident	stdout	-	ident	stdout	-
{...}	block	-	-	block	-	-
0.05	literal	0.05	float	literal	-	float
'Hello'	literal	'Hello'	str	literal	-	str
char[6] y	varDecl	y	char[6]	varDecl	-	char[N]

Variables are one of the main factors that lead to the enormous number of text formulations of the same functionality. Thanks to the edges of name dependence, we can remove variable names from code. Besides the variable names, there are also other constructs in code, which can have any values. One of them is literals, e.g., 0.01, 'Hello', which will cause similar issues as variable names, so we will also remove the concrete value of a literal. Another construct is array declarations with size, e.g., $char[8]$, $char[1024]$. We will normalize them as $char[N]$. A more refined solution could be to create several normalized data types, e.g., $char[uint8]$, $char[uint16]$, $char[uint32]$, and normalize the data type of arrays according to their sizes. For instance, any char arrays with sizes between 0 and 256 could be normalized to $char[uint8]$. Table 1 also provides examples of normalized representations. The definition of the classes of language constructs and the normalized tokens could vary depending on the implementation of applications and the tool for generating code graphs.

4. Evaluation

In order to evaluate our techniques, we have generated four types of code graphs, AST, AST+, ASG, and ASG+, for training DL models of software vulnerability detection. AST+ is AST extended with data dependence, control dependence, and control flow. ASG is AST with the edges of name dependence and variable names removed, and ASG+ is ASG with control dependence, data dependence, and control flow.

4.1. Models

We develop two models (3propASG and 3propASG+) for our techniques and two baselines (codeAST and codeAST+) based on the existing approaches [10,14]. 3propASG and 3propASG+ adopt our graph structures and 3-property node encoding scheme. codeAST and codeAST+ use the common graph structures and the code-based encoding scheme presented in Section 3, which is currently adopted by existing models. In the code-based encoding scheme, a piece of code is used to learn the feature of a node. The piece of code is split into a sequence of tokens and the average of the embeddings of tokens in the sequence is considered as the feature of the node. In comparison, in our property-based encoding scheme, all nodes in the code graph are described by three tokens and the concatenation of the embeddings of the three tokens in a node represents the node's feature.

In order to better evaluate and compare our techniques to the code-based encoding scheme, we follow the design of existing graph-based models for software vulnerabilities [10,14] and use the gated graph recurrent unit (GGRU) [43] as the graph convolution module to learn a graph representation. All models share the following architecture: The graph data of code is first delivered to an embedding layer to learn token embeddings. The learned embeddings and the graph structure are fed into the module of GGRU with one time step, the least expensive option, to perform the operation of graph convolution. The output of GGRU is sent to each of three 1D convolution (Conv1d) layers with 128 filters each and perceptive fields of 1, 2 and 3, respectively, and one 1D max pooling (MaxPool1d) is applied over the output of each Conv1D to downsample. The results of the MaxPool1d layers are concatenated together and sent to a linear hidden layer with 128 neurons, and a 25% dropout is applied to the output of the convolution and hidden layers as a regularization mechanism. We apply ReLU [44] for non-linear transformation because of its computational efficiency and the reduced likelihood of gradient vanishing and use embeddings with 100 dimensions to encode tokens. The architecture is illustrated in Figure 4.

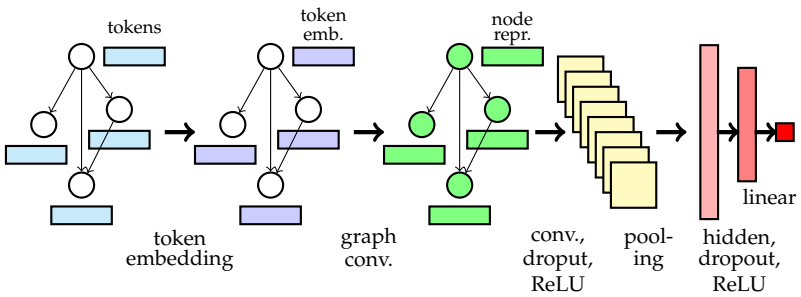


Figure 4. Architecture of models. In our 3-property encoding, the number of tokens of each node is 3. In code-based encoding, each node has the number of tokens of the node that has the most tokens.

Currently, we do not perform a pre-training of token embeddings; instead, we use the standard normal distribution $\mathcal{N}(0,1)$ to initialize the embeddings and test several different initializations. There is empirical evidence [45] which has shown in some cases that pre-trained token embeddings are not necessarily better than random initializations with the standard normal distribution. Future work could investigate if performing a pre-training of token embeddings will significantly improve the performance of DL models in this area.

4.2. Datasets

We use several real-world datasets from different open-source projects, Chromium +Debian [10], FFmpeg+Quemu [14] and VDISC [16], and adopt the open-source tool Joern (https://github.com/joernio/joern (accessed on 1 August 2023)) to create the AST and AST+ from the source code. Our AST+ corresponds to the code property graph of Joern.

Chromium+Debian [10] consists of source code from the Chromium and Debian projects, and code samples are labeled based on the information from their issue-tracking systems. Chromium is a popular Web browser from Google, and Debian is a widely used

Linux operating system. In the dataset of FFmpeg+Qemu [14], FFmpeg is a multimedia framework and Qemu is a hardware emulator and virtualizer. Several professional security experts and researchers labeled the code samples from FFmpeg and Qemu. VDSIC consists of the source code of 1.27 million functions from different open-source software and they are labeled by static analyzers.

Given the limitation of computational resources, it is extremely costly to process the full VDSIC dataset. Therefore, we down-sample negative samples in its training dataset to a number similar to the number of positive samples. Furthermore, we also have to remove the code with the following components: :: operators, new operators, switch, and goto statements, which we found cannot be correctly parsed by Joern. Table 2 gives an overview of the final datasets, 80% of which are used to train the models and 20% for evaluation.

Table 2. Overview of datasets.

Dataset	Total- Bad, Good	80% for Training- Bad, Good
Chromium+Debian	754 (7.05%), 9945 (92.95%)	592 (6.92%), 7967 (93.08%)
FFmpeg+Quemu	5865 (43.68%), 7563 (56.32%)	4687 (43.63%), 6055 (56.37%)
VDISC	31,723 (46.38%), 36,675 (53.62%)	25,304 (46.24%), 29,414 (53.76%)

4.3. Prediction Performance

Given historical data, a model learns its parameter settings in order to optimally generalize the patterns of positive and negative classes. Optimal model parameters are learned by minimizing the cross-entropy loss [46], which especially penalizes those predictions that are confident but wrong. The models are trained with a batch size of 32 and a learning rate of 0.001, and the Adam optimizer [47] is used to minimize the loss function. Each model is trained with 10 different seeds and the training stops after five further training epochs when the validation loss reaches its minimal value. The models are evaluated over the five metrics: accuracy, precision, recall, F1 and AUC, which measure different abilities of models. The results of the evaluation are presented in Tables 3–5. All DL models are developed in the Python programming language and executed under the platform of Linux-5.15.0-88-generic-x86_64-with-glibc2.35. The main software packages used are Python 3.6.9, torch 1.10.2 + cu102, and dgl 0.6.1.

Table 3. Performance of models over Chromium+Debian dataset.

Model	Graph	Encoding		Acc	Prec	Recall	F1	AUC
codeAST	AST	code	with best F1	92.01	44.58	22.84	30.20	60.26
			aver. of 10 trainings	92.4	10.34	2.9	4.14	51.32
3propASG	ASG	3-prop.	with best F1	**92.34**	**49.26**	**41.36**	**44.97**	**68.93**
			aver. of 10 trainings	92.52	46.87	24.75	31.88	61.41
codeAST+	AST+	code	with best F1	90.89	33.66	20.99	25.86	58.80
			aver. of 10 trainings	92.28	15.14	3.83	5.36	51.68
3propASG+	ASG+	3-prop.	with best F1	92.34	49.25	40.74	44.59	68.65
			aver. of 10 trainings	92.41	34.93	18.09	23.35	58.29

Table 4. Performance of models over FFmpeg+Qemu dataset.

Model	Graph	Encoding		Acc	Prec	Recall	F1	AUC
codeAST	AST	code	with best F1	55.36	49.35	67.49	57.01	56.69
			aver. of 10 trainings	58.14	51.32	35.07	37.61	55.61
3propASG	ASG	3-prop	with best F1	**60.35**	**53.43**	74.70	62.30	**61.92**
			aver. of 10 trainings	59.71	53.64	64.91	57.67	60.28
codeAST+	AST+	code	with best F1	58.38	53.27	41.51	46.66	56.53
			aver. of 10 trainings	58.03	52.94	18.37	25.04	53.7
3propASG+	ASG+	3-prop	with best F1	57.04	50.62	**83.36**	**62.99**	59.92
			aver. of 10 trainings	58.9	52.76	65.49	56.85	59.62

Table 5. Performance of models over VDSIC dataset.

Model	Graph	Encoding		Acc	Prec	Recall	F1	AUC
codeAST	AST	code	with best F1	77.82	78.2	73.11	75.57	77.55
			aver. of 10 trainings	77.0	78.3	70.63	74.22	76.63
3prorASG	ASG	3-prop	with best F1	**81.27**	**80.62**	79.11	**79.86**	**81.15**
			aver. of 10 trainings	80.81	79.68	79.36	79.51	80.73
codeAST+	AST+	code	with best F1	75.67	73.32	75.7	74.49	75.67
			aver. of 10 trainings	74.88	74.36	71.1	72.61	74.66
3propASG+	ASG+	3-prop	with best F1	80.94	79.85	**79.4**	79.63	80.85
			aver. of 10 trainings	80.25	78.24	80.39	79.26	80.26

4.3.1. Best Performances

From the 10 times of training, the models with the best F1 value are considered to be the best models. The evaluation results show that the DL models based on our graph structures (ASG and ASG+) and 3-property encoding scheme outperform those based on existing graph structures (AST and AST+) and code-based encoding in terms of the F1 metrics over all the datasets. Among these datasets, Chromium+Debian is extremely imbalanced and contains only 592 (6.92%) programs with vulnerability. Over this dataset, our models perform significantly well with F1: 3propASG is 14.77% better than codeAST and 3propASG+ is 18.73% better than codeAST+. These results are strong evidence that our techniques improve the ability of models to infer the functionality of code. However, we also see that when positive samples become more and more, although our techniques still outperform the existing approaches, the difference in performance becomes less and less. This observation in fact supports our discussion above—collecting more data will also help the models using code-based encoding to combat the diversity of text representations.

4.3.2. Average Performances

We also include the average values of 10 trainings in these tables, which can serve as an indicator of how sensitive a model is to the initialization of parameters. In our evaluation, we see that the models based on code-based encoding approaches are especially sensitive to the initialization of the extremely imbalanced dataset of Chromium+Debian. Out of 10 trainings, it occurred eight times that codeAST has no ability to predict any positive samples and this happens only one time with 3propASG. With the codeAST+ model, this occurs six times. Meanwhile, with 3propASG+, this occurs three times. The sensitivity to initialization gradually decreases as the training data becomes larger. The sensitivity of models to initialization can be observed from the results of the average of 10 trainings. But what does the sensitivity to initialization mean? This is further evidence that it is difficult for the models with code-based encoding to capture the functionality of code.

4.3.3. Basic Graph Structures vs. Extended Ones

The graph structures (AST+ and ASG+) extend AST and ASG with more semantic information (data dependence, control dependence, and control flow). However, these extended graphs do not bring significant benefits to the models. On the contrary, they decrease the detection ability of the models for different metrics in comparison to the basic structures. Flawed semantic dependencies between language constructs in code could cause software vulnerability. So why does the integration of this information not contribute to the detection of vulnerabilities? There might be two possible reasons that could lead to the degradation of performance after the introduction of more information: (i) AST+ and ASG+ are much more complicated than AST and ASG. It becomes more difficult for models to generalize the vulnerability patterns hidden in very complicated structures; (ii) AST and ASG are well-defined and relatively easy to build. However, identifying the data dependence, control dependence, and control flow in code is not a simple task, and constructing high-quality AST+ and ASG+ is much more difficult. Joern is a great tool for us to obtain AST+ and ASG+, but it currently has limited capabilities and may generate inaccurate code graphs.

4.4. Quality, Representation and Separability

The quality of data ultimately determines what quality a DL model can achieve. To analyze the quality of data and models, in this section, we will examine the representations learned by the models and how well these representations separate positive (vulnerable) from negative (non-vulnerable) samples. One way to do this is to visualize these representations and their separability. Two popular techniques for visualizing high-dimensional data are t-SNE [48] and UMAP [49]. In this work, we are using the latter, because it can preserve a more global structure and is computationally more efficient than t-SNE.

Figure 5 presents the representations learned and the classes detected by the four models on the evaluation parts of the three datasets, where the representation is from the hidden linear layer; the gray dots indicate correctly predicted negative samples (TN) and the blue dots the correctly detected positive samples (TP); the red dots are the negative samples which are wrongly detected as positive (FP) and the green dots are the positive samples which are wrongly detected as negative (FN).

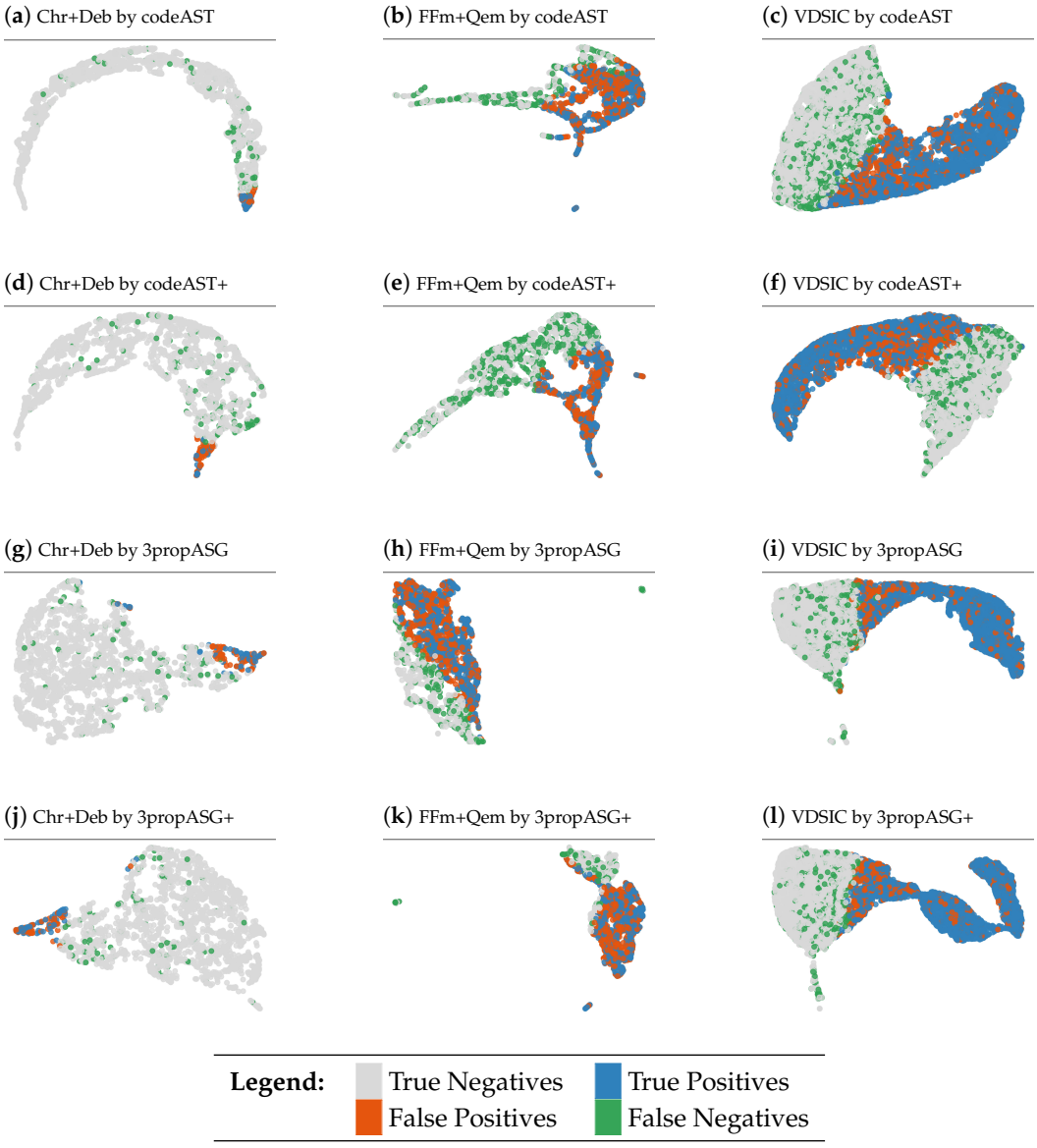


Figure 5. Representation, separability and prediction of the four models in the three datasets.

In Figure 5, we see that the representations learned by the models (3propASG, 3propASG+) based on our techniques are different from the representations learned by

the models (codeAST, codeAST+) based on existing techniques over different datasets and different representations lead to different shapes of scatter plots. However, unfortunately, from these scatter plots alone it is not obvious which representations are superior to the others and which shape is better or worse. Nevertheless, we can still observe that the models based on ASG, ASG+ and 3-property encoding produce fewer green and red points, and thus make fewer false-positive and false-negative errors than the models based on AST, AST+ and code-based encoding. This observation is supported by the concrete evaluation results presented in Tables 3–5.

What we can see from these scatter plots is that (i) positive samples (blue and green dots) interleave together with negative samples (gray and red dots), and (ii) in the areas where the number of samples of a class dominates, mostly all samples of another class are wrongly detected as the dominated class in that area. These facts mean that there is a low separability of classes and it is really hard to find reasonably good boundaries to separate them. This, in turn, indirectly shows that the representations based on our techniques are better ones because they enable DL models to make better detection.

4.5. Memory Footprint

A huge advantage of our 3-property encoding is that it has very low memory requirements and can process very large code graphs in comparison to the existing code-based encoding. In our experiments, 8 GB of memory is enough to process all data using 3-property encoding. In comparison, code-based encoding requires as much as 560 GB of memory. With the 3-property encoding, the feature of each node is represented by only three tokens. With code-based encoding, the feature of each node is represented by a piece of raw code. Although different pieces of code will create different numbers of tokens and the minimal node could contain only one token, all nodes are required to have the same number of tokens. This means that all nodes in a code graph finally consist of the maximum number of tokens.

Let $\#nodes$ be the number of nodes in a code graph, dim the dimension of token embeddings, $\#tokens$ the maximal number of tokens in the graph, and each value consumes 4 bytes. The memory needed for processing a single graph of code is computed as follows:

$$\begin{array}{ll} \text{3-property encoding:} & \#nodes \times 3 \times dim \times 4 \\ \text{code-based encoding:} & \#nodes \times \#tokens \times dim \times 4 \\ \text{code-based/3-property:} & \frac{\#nodes \times \#tokens \times dim \times 4}{\#nodes \times 3 \times dim \times 4} = \frac{\#tokens}{3} \end{array}$$

Let us use several samples from the Chromium+Debian dataset to demonstrate the efficiency of our technique. Table 6 provides the memory footprint required by our 3-property encoding and the existing code-based encoding for processing these samples. The comparison shows that our encoding scheme can be up to 32,000 times more efficient than the code-based encoding.

Table 6. Memory needs of three samples from Chromium+Debian.

Code ID	#Nodes	#Tokens	Code-Based	3-prop.	Code-Based /3-prop.
-6552851419396579257	4409	33,659	59 G	5.3 M	11,220
2388171415474875762	7012	54,157	152 G	8.4 M	18,052
5045872831385413038	12,077	96,805	468 G	14.5 M	32,268

More concretely, for the embeddings of 100 dimensions with 8 GB of main memory, our 3-property encoding can process graphs with 7 million nodes, and with 128 GB of main memory, it can handle graphs with 114 million nodes. As shown in Table 6, with the code-based encoding, 128 GB main memory is not enough for the graph with 7012 nodes. This explains why existing works [10,14] only use code samples with a number of nodes less than 500.

4.6. Alleviating Vocabulary Explosion

Since there are infinitely many ways of naming identifiers in program code, the representation learning approaches face the problem of vocabulary explosion. Our techniques remove the variable names from code and this naturally brings the benefit of a smaller vocabulary and, thus, mitigates the issue of vocabulary explosion. Table 7 exemplifies the number of vocabularies generated by the existing code-based and our 3-property node encoding schemes for the three datasets.

Table 7. Size of vocabulary generated by the code-based and our 3-property encoding schemes.

Dataset	Code-Based	3-prop.	3-prop./Code-Based
Chromium+Debian	57,027	35,416	62.10%
FFmpeg+Qume	66,791	45,795	68.56%
VDSIC	449,148	312,948	69.68%

5. Summary and Conclusions

A challenge in deep learning-based software vulnerability detection is the infinity and randomness of text representations of functionality in software code. Variable names in code are one cause of this issue. In order to better cope with them, in this work, we introduce edges of name dependence and a type of abstract syntax graph (ASG) extending AST with this new type of edges and suggest a 3-property node encoding scheme based on the ASG. These techniques not only allow us to represent the semantics of code without using its variable names but also allow us to encode all nodes in a consistent way. These characteristics will help DL models capture the commonalities and differences between nodes and learn the intrinsic functionality of code more easily.

In order to evaluate our techniques, we develop two models for our techniques and two baselines based on the existing graph structures and node encoding scheme, and all the models are built on the gated graph encoding network and convolution network. These models are trained on several real-world datasets from widely used open-source projects and libraries (Chromium, Debian, FFmpeg and Qemu), and they are evaluated using the five criteria (accuracy, precision, recall, F1 and AUC-ROC). The results of the evaluation show that the models based on our techniques outperform the ones based on existing approaches.

Apart from leading to better detection performance, our techniques have two further advantages. One of them is that they naturally lead to the mitigation of the problem of vocabulary explosion because the large variety of variable names is a key reason for this problem. The other is that they have a very low memory footprint. Given a certain dimension of embeddings, the memory need of the 3-property encoding is linearly proportional to the number of nodes in the code graph while the memory need of the code-based encoding is linearly proportional to the product of the number of nodes and the maximal number of tokens. For example, our techniques can process graphs with 7 million nodes with 8 GB memory given 100-dimension embeddings. To be able to process the datasets used in the evaluation, the code-based encoding scheme needs up to 468 GB of main memory while our techniques consume only 14.5 MB of main memory. The difference in memory need is in a dimension of 32,000 and this is really amazing. Apart from its application in software vulnerability detection, we believe that the 3-property encoding (with or without variable names) will also be a useful technique for many tasks in software analysis and software engineering.

Author Contributions: Conceptualization, J.G.; Methodology, J.G.; Software, J.G.; Validation, J.G.; Formal analysis, J.G.; Data curation, J.G.; Writing—original draft, J.G.; Writing—review & editing, S.G., D.S. and R.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Federal Ministry of Education and Research of Germany under Grant Agreement No 16KIS1337.

Data Availability Statement: The datasets analyzed during the current study are publicly available from the following links: https://drive.google.com/drive/folders/1KuIYgFcvWUXheDhT--cBALsfy1I4utOy; https://osf.io/d45bw/; https://drive.google.com/file/d/1x6hoF7G-tSYxg8AFybggypLZgMGDNHfF/view?pli=1.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Brooks, T.N. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. In Proceedings of the Science and Information Conference, Semarang, Indonesia, 7–8 August 2018; pp. 1083–1102.
2. Henzinger, T.A.; Jhala, R.; Majumdar, R.; Sutre, G. Software verification with BLAST. In Proceedings of the Workshop on Model Checking of Software, Portland, OR, USA, 9–10 May 2003; pp. 235–239.
3. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1032–1043.
4. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016; pp. 1–16.
5. Johnson, B.; Song, Y.; Murphy-Hill, E.; Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 672–681.
6. Smith, J.; Johnson, B.; Murphy-Hill, E.; Chu, B.; Lipford, H.R. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 248–259.
7. Ayewah, N.; Pugh, W.; Morgenthaler, J.D.; Penix, J.; Zhou, Y. Evaluating static analysis defect warnings on production software. In Proceedings of the 7th Acm Sigplan-Sigsoft Workshop on Program Analysis for Software Tools and Engineering, San Diego, CA, USA, 13–14 June 2007; pp. 1–8.
8. Newsome, J.; Song, D.X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. *Proc. Ndss. Citeseer* **2005**, *5*, 3–4.
9. Liu, B.; Shi, L.; Cai, Z.; Li, M. Software vulnerability discovery techniques: A survey. In Proceedings of the 2012 Fourth International Conference on Multimedia Information Networking and Security, Nanjing, China, 2–4 November 2012; pp. 152–156.
10. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep learning based vulnerability detection: Are we there yet. *IEEE Trans. Softw. Eng.* **2021**, *48*, 3280–3296. [CrossRef]
11. Collobert, R.; Weston, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In Proceedings of the 25th International Conference on Machine Learning, Helsinki, Finland, 5–9 July 2008; pp. 160–167.
12. Dahl, G.; Ranzato, M.; Mohamed, A.R.; Hinton, G.E. Phone recognition with the mean-covariance restricted Boltzmann machine. *Adv. Neural Inf. Process. Syst.* **2010**, *23*, 1–9.
13. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
14. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 1–11.
15. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv* **2018**, arXiv:1801.01681.
16. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; McConley, M. Automated vulnerability detection in source code using deep representation learning. In Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018; pp. 757–762.
17. Dam, H.K.; Tran, T.; Pham, T.; Ng, S.W.; Grundy, J.; Ghose, A. Automatic feature learning for vulnerability prediction. *arXiv* **2017**, arXiv:1708.02368.
18. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2224–2236.
19. Prömel, H.J. Complete disorder is impossible: The mathematical work of Walter Deuber. *Comb. Probab. Comput.* **2005**, *14*, 3–16. [CrossRef]
20. Graham, R.L.; Rothschild, B.L.; Spencer, J.H. *Ramsey Theory*; John Wiley & Sons: Hoboken, NJ, USA, 1991; Volume 20.
21. Groppe, J.; Groppe, S.; Möller, R. Variables are a Curse in Software Vulnerability Prediction. In Proceedings of the 34th International Conference on Database and Expert Systems Applications (DEXA 2023), Penang, Malaysia, 28–30 August 2023; Springer: Berlin/Heidelberg, Germany , 2023; pp. 1–6.
22. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 297–308.

23. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y.; De Vel, O.; Montague, P. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3289–3297. [CrossRef]
24. Pradel, M.; Sen, K. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.* **2018**, *2*, 1–25. [CrossRef]
25. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
26. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv* **2019**, arXiv:1907.11692.
27. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and Evaluating Contextual Embedding of Source Code. *arXiv* **2020**, arXiv:2001.00059.
28. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
29. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. Graphcodebert: Pre-training code representations with data flow. *arXiv* **2020**, arXiv:2009.08366.
30. Wang, X.; Wang, Y.; Mi, F.; Zhou, P.; Wan, Y.; Liu, X.; Li, L.; Wu, H.; Liu, J.; Jiang, X. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv* **2021**, arXiv:2108.04556.
31. Du, Q.; Kuang, X.; Zhao, G. Code Vulnerability Detection via Nearest Neighbor Mechanism. In Proceedings of the Findings of the Association for Computational Linguistics, Dublin, Ireland, 22–27 May 2022.
32. Bengio, Y.; Courville, A.; Vincent, P. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* **2013**, *35*, 1798–1828. [CrossRef] [PubMed]
33. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2244–2258. [CrossRef]
34. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014; pp. 590–604.
35. Yamaguchi, F.; Maier, A.; Gascon, H.; Rieck, K. Automatic inference of search patterns for taint-style vulnerabilities. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 797–812.
36. Fey, M.; Lenssen, J.E. Fast graph representation learning with PyTorch Geometric. *arXiv* **2019**, arXiv:1903.02428.
37. Wang, M.; Zheng, D.; Ye, Z.; Gan, Q.; Li, M.; Song, X.; Zhou, J.; Ma, C.; Yu, L.; Gai, Y.; et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv* **2019**, arXiv:1909.01315.
38. Ehrig, H.; Rozenberg, G.; Kreowski, H.J. *Handbook of Graph Grammars and Computing by Graph Transformation*; World Scientific: London, UK, 1999; Volume 3.
39. Garner, R. An abstract view on syntax with sharing. *J. Log. Comput.* **2012**, *22*, 1427–1452. [CrossRef]
40. Wang, Y.; Li, H. Code completion by modeling flattened abstract syntax trees as graphs. In Proceedings of the AAAI Conference on Artificial Intelligence, Virtual Event, 8 February 2021; pp. 14015–14023.
41. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Professional: Boston, MA, USA, 2018.
42. Raghavan, S.; Rohana, R.; Leon, D.; Podgurski, A.; Augustine, V. Dex: A semantic-graph differencing tool for studying changes in large code bases. In Proceedings of the 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11–17 Sepember 2004; pp. 188–197.
43. Li, Y.; Tarlow, D.; Brockschmidt, M.; Zemel, R. Gated graph sequence neural networks. *arXiv* **2015**, arXiv:1511.05493.
44. Fukushima, K. Cognitron: A self-organizing multilayered neural network. *Biol. Cybern.* **1975**, *20*, 121–136. [CrossRef] [PubMed]
45. Groppe, J.; Schlichting, R.; Groppe, S.; Möller, R. Deep Learning-based Classification of Customer Communications of a German Utility Company. In *Lecture Notes in Electrical Engineering*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 1–16.
46. Murphy, K.P. *Machine Learning: A Probabilistic Perspective*; MIT Press: Cambridge, MA, USA, 2012.
47. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
48. Van der Maaten, L.; Hinton, G. Visualizing data using t-SNE. *J. Mach. Learn. Res.* **2008**, *9*, 2579–2605.
49. McInnes, L.; Healy, J.; Melville, J. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv* **2020**, arXiv:1802.03426.