applied sciences

MDPI

Article

Detection of Software Security Weaknesses Using Cross-Language Source Code Representation (CLaSCoRe)

Sergiu Zaharia [1], Traian Rebedea [1] and Stefan Trausan-Matu [1,2,*]

[1] Faculty of Automatic Control and Computers, University Politehnica of Bucharest, 060042 Bucharest, Romania; sergiuzaharia@yahoo.com (S.Z.); traian.rebedea@upb.ro (T.R.)
[2] Institute for Artificial Intelligence "Mihai Draganescu" of the Romanian Academy, 050711 Bucharest, Romania
* Correspondence: stefan.trausan@upb.ro

Abstract: The research presented in the paper aims at increasing the capacity to identify security weaknesses in programming languages that are less supported by specialized security analysis tools, based on the knowledge gathered from securing the popular ones, for which security experts, scanners, and labeled datasets are, in general, available. This goal is vital in reducing the overall exposure of software applications. We propose a solution to expand the capabilities of security gaps detection to downstream languages, influenced by their more popular "ancestors" from the programming languages' evolutionary tree, using language keyword tokenization and clustering based on word embedding techniques. We show that after training a machine learning algorithm on C, C++, and Java applications developed by a community of programmers with similar behavior of writing code, we can detect, with acceptable accuracy, similar vulnerabilities in C# source code written by the same community. To achieve this, we propose a core cross-language representation of source code, optimized for security weaknesses classifiers, named CLaSCoRe. Using this method, we can achieve zero-shot vulnerability detection—in our case, without using any training data with C# source code.

Keywords: software security engineering; machine learning; code embeddings; common weakness enumeration; zero-shot classification

Citation: Zaharia, S.; Rebedea, T.; Trausan-Matu, S. Detection of Software Security Weaknesses Using Cross-Language Source Code Representation (CLaSCoRe). *Appl. Sci.* **2023**, *13*, 7871. https://doi.org/10.3390/app13137871

Academic Editors: Irina Trubitsyna and Reza Shahbazian

Received: 30 May 2023
Revised: 1 July 2023
Accepted: 3 July 2023
Published: 4 July 2023

1. Introduction

Cybersecurity risks are moving gradually to the top of both business and global concerns lists, according to The Global Risks Report 2023 issued by the World Economic Forum [1] and to many other risk professionals. One of the main channels used in hacking attacks is software applications, with billions of lines written in various programming languages, each of them being a potential vehicle for security vulnerabilities. The existing technologies and skills to detect security flaws in source code, early in the software development process, are limited to a subset of programming languages and security vulnerabilities. Increasing the capacity to identify security weaknesses in languages that are not well supported by specialized security scanners and analysts, based on the knowledge gathered from securing the popular ones, is vital in reducing the overall exposure of software applications. We propose a solution to expand the capabilities of security gaps detection to downstream languages, influenced by their more popular ancestors from the programming languages' evolutionary tree, using language keyword tokenization and clustering based on word embedding techniques.

We show that after training a machine learning algorithm on C, C++, and Java applications developed by a community of programmers with similar behavior of writing code, we can detect similar vulnerabilities in C# source code written by the same community, using a core cross language representation of source code, optimized for security weaknesses classifiers, named CLaSCoRe.

1.1. Problem Statement and Research Objectives

Early detection of security vulnerabilities in source code is the most economical way to increase software application resilience. The number and complexity of weaknesses that can be introduced in source code by programmers with insufficient experience in building secure software, or as a result of the pressure to deliver their products in a very short timeframe, are increasing each year. Automated solutions to scan the source code for security vulnerabilities are available to the communities of programmers, as commercial or freeware tools. They are known, in general, as Static Application Security Testing (SAST) technologies. SAST technologies may cover one or multiple programming languages, especially the most popular, like C/C++, Java, JavaScript, or PHP. The most professional tools can identify more than 1000 security weaknesses in source code written in those languages. However, the majority of less popular languages do not benefit from automated scanning solutions, as the market potential is less attractive for developing SAST solutions.

Based on various sources, there are between 250 and 700 active programming languages in use, and about 8,945 programming languages in history, according to the Online Historical Encyclopedia of Programming Languages [2]. The Tiobe Index [3] continuously updates the top 50 popularity scores for the 278 programming languages in the inventory. Wikipedia [4] presents a list of 687 programming languages per name, category, chronologically, or per generation. An important part of the source code written globally remains exposed to security exploitation, as SAST technology coverage remains limited to a subset of less than 50 programming languages. As the access to software security experts able to manually identify flaws in less popular languages is limited, the embedded cyber resilience of the corresponding software applications is questionable.

As an alternative to SAST technologies, there is an increasing trend in using machine learning-based algorithms to identify security vulnerabilities in source code or within the binary format of software components. Most of them are developed using specially designed datasets, with software modules labeled as containing a "vulnerable" or "not vulnerable" piece of code, per each supported security weakness listed in public repositories like NVD (National Vulnerability Database) [5] or MITRE CWE (Common Weakness Enumeration) [6]. As labeled source code is available only for a limited subset of programming languages, these solutions complement SAST technologies for the most popular languages, especially C and Java, and are not able to detect security weaknesses in languages not supported with labeled datasets.

The objective of the current research is to identify the optimal representation of the source code and the zero-shot classification model able to detect security vulnerabilities in source code written in a different programming language than those used for training.

1.2. Main Contributions

We propose a cross-language source code representation, able to preserve the security vulnerabilities patterns of the source code developed in various programming languages, and show that common machine learning algorithms (like Support Vector Machines) with default configurations can detect these patterns in code written in other languages which were not part of the training (as long as on the evolutionary tree they are influenced by the ones used in the learning phase). This approach can support the communities of developers, mainly those who follow similar strategies in writing code, to identify security flaws in software developed using programming languages that are not well supported by static analysis tools or software security reviewers.

The representation is based on the concept of splitting the control flow and data flow sections of source code into two loosely coupled vectors and applying word embeddings and clustering of multiple programming languages' keywords to the resulting control flow elements, a method to retain the patterns of security vulnerabilities and at the same time to remain less dependent on the lexical and semantical structure of the programming language used to write them. The word embedding techniques are used to identify semantical relations between keywords across different programming languages (e.g., C, C++, Java,

C#). Clustering techniques are then used to make the representation less dependent on the lexical structure of the programming language by replacing multiple keywords with unique values per cluster. In this way, lexically different keywords, but close from a semantical perspective to C, C++, Java, or C#, are considered identical, reducing the dependency of source code representation from the original language used to write it.

We use the proposed source code representation to train popular machine learning algorithms on datasets written in C, C++, and Java, and test their performance on datasets written in C#. All datasets are synthetic and provided by the National Institute of Standards and Technology (NIST) [7] for software security scanners' evaluations. We show that security vulnerabilities can be detected with good precision in software developed using less popular programming languages, as long as datasets are defined for programming languages that are positioned as their influencers in the evolutionary tree.

The novelty of our approach is given by the following techniques:

- Applying word embedding techniques to a corpus of source code developed in multiple languages by a community of programmers with similar behavior of programming—an evolutionary step of our previous research [8].
- Applying iterative clustering techniques to source code relevant keywords that have semantical similarities when used in source code written in different languages by a community of developers.
- Developing a source code representation that is able to preserve security weaknesses patterns between different programming languages with close syntactical or semantical structures, as a result of the influence the mainstream languages have on the more recent ones.

The next sections of this document are structured as follows:

- Section 2—Related Works: describes the state of the art and positions our approach in the overall context of machine learning-based security weaknesses detection in source code.
- Section 3—The Proposed Architecture for the Cross Language Source Code Representation: describes our approach to transforming source code written in C, C++, Java, or C# into an abstract representation that is able to preserve security weaknesses patterns and is less dependent on the original language lexical structure.
- Section 4—Experimental Results: describes our tests on the source code dataset provided by NIST, exemplifying the outcome of learning in two main scenarios, "train in C#, test on C#" and "train in C, C++ and Java, test on C#".
- Section 5—Conclusions: presents our conclusions and the next steps related to the cross-language CWE detection approach.

2. Related Works

During the last decade, security researchers and experts in machine learning and natural language processing joined their efforts and worked on solutions able to detect software security weaknesses in source code using less deterministic methods, as is the case with SAST solutions. These new models are not intended to replace static analysis tools, as their role in securing software remains essential; however, the models based on machine learning and natural language processing are performing better from year to year, threatening the supremacy of deterministic SAST technologies. Providers of traditional SAST solutions have also embraced these new models, making their solutions more accurate in determining software security vulnerabilities. We present below relevant software security and quality scanners, based on machine learning and natural language processing techniques, developed in the last decade.

Yamaguchi et al. [9] propose Chucky, a model designed to support software security analysts in identifying vulnerabilities caused by improper or missing validations of the input data, which may lead to unnecessary permissions and access allowed to legitimate users (e.g., user access to folders for which they do not have a "need to know") or to buffer overflow scenarios where the computer memory allocated to the application during

runtime is not protected against being written with malicious instructions by external processes, resulting in application disruption, illegitimate remote control, or sensitive data access. The model verifies how data being introduced as input to an application is validated during its entire lifecycle until its transfer to an instruction or process. Chucky identifies in source code the patterns for validation of input data that is propagated through the source code, using Abstract Syntax Trees to keep track of this flow of data, and word embeddings to understand the context around the code functions. Experimental tests have been executed for specific CVE detection in open-source code libraries.

Yamaguchi et al. [10] introduced the concept of Code Property Graph (CPG), a data structure combining various properties of Abstract Syntax Tree (AST), Control Flow Graphs (CGF), and Program Dependence Graphs (PDG). The functions in source code are transformed into Code Property Graphs at the individual level and consequently linked together in the entire code base (e.g., Linux kernel). The authors show that this combination of control flow, data flow, and syntactical information is able to preserve patterns of CVE vulnerabilities, which are identified by comparing CVE-labeled graph traversals with those resulting from the transformation of the source code under analysis. New, unpublished security vulnerabilities, like buffer overflows, have been identified in Linux distributions using this approach.

Suneja et al. [11] consider the CPG representation of source code at the function level written in C, which is then vectorized through word embeddings via the Word2Vec method proposed by Mikolov et al. [12], as to preserve its semantic information, and applying Graph Neural Networks [13] to identify relationships between the nodes and edges of source code graphs. Their model performed better on synthetic datasets like NIST Juliet than freeware static analysis solutions.

Russell et al. [14] compiled a list of more than 1 million C/C++ functions from open sources (NIST SARD suite, GitHub, Debian Linux distributions) and developed a labeled dataset with both synthetic (NIST) and natural code, with the support of static analysis tools and security professionals. The authors used Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) for features extraction and applied the "Bag-of-Words" Embedding Model to the tokenized code, training the solution with the Random Forest ensemble classifier for CWEs identification. The authors showed that results on synthetic datasets are better than those on natural code.

Li et al. [15] propose the SySeVR (Syntax-based, Semantics-based, and Vector Representations) format for C/C++ source code, which includes word embeddings techniques (Word2Vec), intended to preserve the lexical and semantical information with relevance to security vulnerabilities. The security flaws contextual information is obtained with a method inspired by the region proposal [16] concept familiar for image processing. This representation is proven to be highly effective for security weaknesses classification by deep neural networks. Xuan et al. [17] apply the SySeVR framework to parse C/C++ samples from the NIST SARD dataset and uses word embeddings (Word2Vec) to normalize the code so that its length remains constant when fed to machine learning classifiers, out of which Random Forest provided the best results. The model has been supported by a professional static analysis tool in mapping vulnerable code functions to categories of security flaws like arithmetic expression, pointer usage, and array usage vulnerability.

Saha et al. [18] developed a method to automatically capture patterns including sequences of instructions from CWEs published in databases like NIST SARD, each CWE being enriched with the information collected from CVE repositories. They propose the ML-FEED (Machine Learning Framework for Efficient Exploit Detection), which extracts individual instructions from the newly developed exploit database and label them with the CWEs that these instructions may trigger. The instructions are transformed to vectors via word embeddings using FastText [19], and a multi-label classifier designed as a Feedforward Neural Network (FNN) is used to predict all exploits that a specific instruction can execute. Instructions which are not linked to any CWE are white-listed and are bypassed in the classification process, contributing to increased performance.

Wang et al. [20] developed the DeepVulSeeker architecture based on the pre-trained semantic model named UniXcoder [21] built on transformers and efficient in encoding natural language to program languages. The pre-trained semantic model is applied to the structural representation of the source code via AST, CGF, and Data Flow Graphs (DFG); the CWE patterns are identified by CNN and FNN types of neural networks.

New trends consider using Large Language Models as security scanners, secure code generators, and pedagogical solutions for software developers or software security analysts. The models are able to generate source code in different programming languages to explain in detail how to fix the software [22] and hardware [23] security weaknesses or quality bugs [24] within the products under development and to develop new versions with improved security. This characteristic of "self-healing" has to be well managed by data scientists and security professionals, as to obtain a strong "security by immunity" of Artificial Intelligence generated source code in the future. An important fact related to ChatGPT-based models of security scanners referred to within this section is that the training dataset, despite its huge size and coverage, does not include information about vulnerabilities discovered in the last 2 years, being limited to data generated in 2021 at the latest. On the other hand, datasets like NIST SARD Suite are even older, its C# knowledge base being developed in 2016, the Java dataset updated in 2017, with the latest updates being provided in 2022 for the PHP and C/C++ datasets.

Similar approaches are used in code quality scanners, for software bug detection. Software quality bugs are detected using methods similar to those applied for detecting security vulnerabilities. ChatGPT identifies quality bugs with a success rate of 77.5% and is able to recommend fixes [24] using human interaction and follow-up in dialogue-based communication. On a conceptual level, methods used to detect quality bugs may be used to detect security vulnerabilities as well, because these models look in similar ways for lexical and semantical patterns within the source code. The accuracy of Large Language Models is not always the intended one, as these models are not trained specifically on security and quality detection activities. As an example, using CodeBERT, the particular Bidirectional Encoder Representations from Transformers (BERT) model adapted for source code, software defects are identified by Pan et al. [25] with newly trained or pre-trained models with average F1 values between 0.5 and 0.6. The various models in the literature for detecting security and quality patterns in source code with their main characteristics are presented in Table 1.

A zero-shot, cross-language scanner leveraging the similarities among programming languages by training the model to identify CWEs in Java and detecting vulnerabilities in C#, has been proposed by Chauhan [26]. The solution applies per each language in scope (Java and C#) the code2vec [27] embeddings method, to capture the semantic properties of source code, converting the code snippets in fixed lengths feature vectors. The best results on the NIST SARD suite, using RNN-based, zero-shot, classifiers, are F1 = 0.55 for the "C# to Java" scenario and 0.74 for the "Java to Java" scenario.

The proposed Cross Language Source Code Representation (CLaSCoRe) model is positioned as a cross-language CWE scanner, built on word embeddings and clustering techniques to define security weaknesses patterns within source code which are detected by supervised machine learning classifiers like Support Vector Machines. We believe that our zero-shot, cross-language model is filling the state-of-the-art gap in CWE detection within source code, using a model easy to implement by the communities of software developers and security analysts.

Table 1. State of the Art in Machine Learning and Natural Language Processing based Source Code Security Scanners.

Model Name & Paper Date	Functionality & Language Agility							Based on					Datasets	
	Security Scanner CWE	Security Scanner CVE	Code Quality Scanner	Pedagogical Tool	Code Generator	Language Agnostic	Cross Language	ML/NLP Model	Transformers	Word /Code Embeddings	Clustering	Graphs	NIST SARD	Other Datasets
Chucky (2013)		x		x				Clustering		x	x	x		LibTIFF, Pidgin
CPG (2014)		x						Static Analysis				x		Linux kernel
Russel et al. (2018)	x					x		RF		x		x	x	Draper
AI4VA (2020)	x							GNN		x		x	x	Draper, s-bAbI
Chauhan (2020)	x						x	RNN		x		x	x	
SySeVR (2020)	x	x						RNN /CNN		x			x	CVE NVD
Do Xuan et al. (2022)	x							RF		x			x	
Zaharia et al. (2022)	x					x		RF, DT		x	x		x	
ML-FEED (2022)	x							FNN		x		x	x	CVE NVD
DeepVulSeeker (2023)	x							CNN/FNN	x			x	x	QEMU, FFMPEG
ChatGPT 3.5 (2023)	x			x	x	x		LLM	x					Large data until 2021
ChatGPT 3 (2023)			x	x	x	x		LLM	x					QuixBugs & Large data until 2021
CodeBERT (2023)			x					LLM	x					PROMISE
CLaSCoRe (2023)	x						x	SVM		x	x		x	

3. The Proposed Architecture for the Cross-Language Source Code Representation

We propose the architecture presented in Figure 1, to transform source code written in various programming languages into the CLaSCoRe representation. This concept builds on our previous research related to machine-learning-based security pattern recognition techniques [8], able to detect security weaknesses in C/C++ and Java using a mix of keywords (tokens) from both languages. The CLaSCoRe representation maintains the minimal, core traits of source code needed to classify it as "vulnerable" or "not vulnerable", although it is less dependent on the lexical and semantical structure of the original programming language.

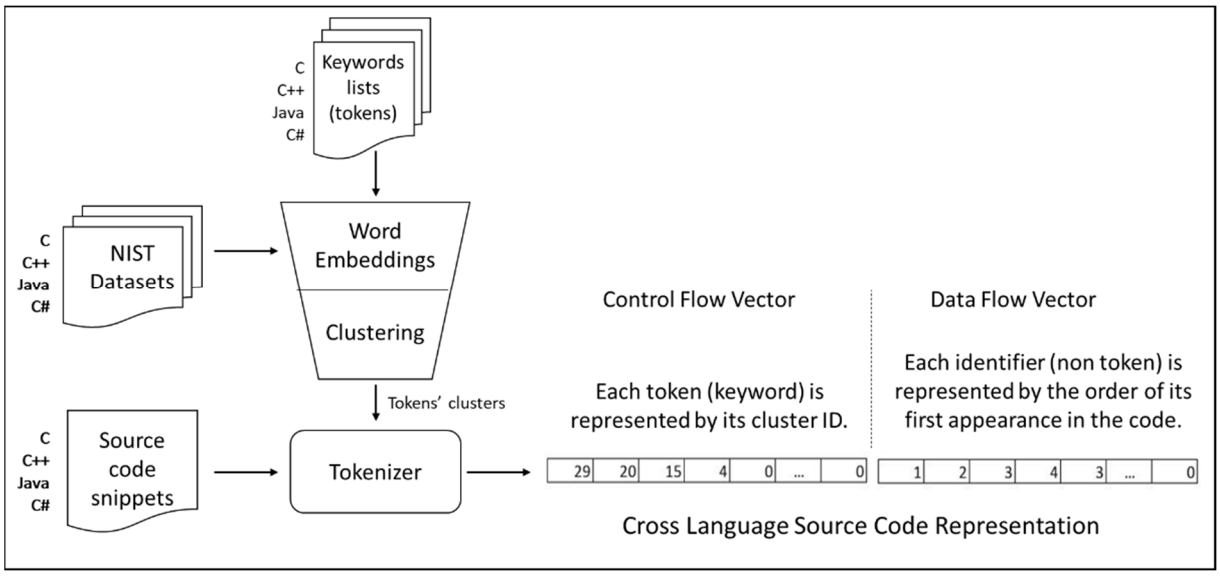

Figure 1. Cross Language Source Code Representation development architecture.

Examples of code snippets from NIST datasets, with vulnerable functions written in Java and C#, input to the Tokenizer, are displayed in Table 2. The steps for building the ClaSCoRe representation are described in more detail below.

Table 2. CWE 78 (OS Command Injection using Use environment variable) vulnerability in Java and C# code.

Java Vulnerable Function	C# Vulnerable Function

public void bad() throws Throwable
{
String data;
if (PRIVATE_STATIC_FINAL_FIVE = = 5)
{
data = System.getenv(“ADD”);
}
else
{
data = null;
}
 	
String osCommand;
if (System.getProperty(“os.name”).toLowerCase().
indexOf(“win”) >= 0)
{
osCommand = “c:\\WINDOWS\\
SYSTEM32\\cmd.exe /c dir “;
}
else
{
osCommand = “/bin/ls “;
}
Process process = Runtime.getRuntime().
exec(osCommand + data);
process.waitFor();
}
	public override void Bad()
	{
	string data;
	if (PRIVATE_CONST_TRUE)
	{
	data = Environment.
	GetEnvironmentVariable(“ADD”);
	}
	else
	{
	data = null;
	}
	String osCommand;
	if (RuntimeInformation.
	IsOSPlatform(OSPlatform.Windows))
	{
	osCommand = “c:\\WINDOWS\\
	SYSTEM32\\cmd.exe /c dir “;
	}
	else
	{
	osCommand = “/bin/ls “;
	}
	Process process = Process.
	Start(osCommand + data);
	process.WaitForExit();
	}

3.1. Multi-Language Tokens Inventory

The proposed solution is used to train machine learning algorithms for the detection of vulnerable code written in C, C++, and Java and to later detect the same vulnerabilities in C#, so we collect all possible keywords we may find in all these programming languages. We named these keywords tokens, by convention, as they will contribute to the tokenization of source code into the CLaSCoRe format.

The keywords included in the inventory are the following:

- the reserved keywords for C++98 and C++11, which include the C keywords;
- the Portable Operating System Interface POSIX.1-2017 identifiers and functions from The Open Group [28];
- Linux or Windows functions which we may find in source code written in C/C++
- the reserved keywords for Java;
- Java predefined classes, methods, variables, operators, constructors, or interfaces [29];
- the reserved and contextual identifiers for C# [30].

We have identified a total of 1782 tokens specific to C/C++ programs, 19.775 tokens specific to Java code, and 120 C# tokens. Some of the tokens are specific to more than one programming language. As a result, after removing all duplicates, the total number of tokens present in the source code written in C, C++, Java, or C# languages is 21,464. We selected from the full list of tokens the ones which are present in the NIST datasets for the programming languages in the scope of our research to optimize the speed of processing. We have identified 920 tokens that appear in the NIST dataset at least once.

3.2. Identifying Semantic Similarities between Tokens from Multiple Languages

We identify semantic similarities between the 920 tokens in our inventory using as a corpus the entire dataset of "vulnerable" and "not vulnerable" source code provided by NIST for C, C++, Java, and C# programming languages. An important characteristic of the dataset provided by NIST is that the code is synthetic, and it follows similar characteristics of code writing, as in the case of communities of programmers having a similar approach and behavior in writing software programs. One of the main research goals is to support these communities in the identification of vulnerable source code written in programming languages they use to develop applications (like C# in our case), once they have the technical and human ability to detect and label vulnerabilities in other, more popular languages (C, C++, Java).

To semantically group the tokens, we use word embedding techniques like Word2Vec, supported by Natural Language Processing concepts, implemented by NLTK (Natural Language Toolkit) [31]. The resulting word vectors, specific to every keyword in the inventory, preserve the contextual similarity between tokens from multiple programming languages.

New techniques aimed at identifying semantic similarities between tokens from different languages are being developed, like the multilingual, cross-lingual, or polylingual language models [32]. The cross-lingual models are, in general, pre-trained on text written in multiple languages, reducing the influence of the semantics culturally embedded within the source code created by developers which are part of a stable community. On the other hand, these models typically require large amounts of training data and processing power, with an increased complexity, which may not be "developer-friendly". Additionally, the cross-lingual models strongly rely on very well-aligned data between languages, which is a challenge when defining a dataset with vulnerable code in Java, C, C++, and C#. However, our future work will consider testing the impact of the multilingual and cross-lingual models on CLaSCoRe representation.

3.3. Clustering Tokens from Multiple Languages, Based on Semantic Similarities

The word vectors are used to develop clusters of tokens from different programming languages, all tokens in a cluster being replaced in the next phases with the cluster identification number. As a result, the lexical and semantical differences between programming languages are reduced and the code representation is less dependent on the language.

We propose the K-Means clustering algorithm for the word embedding vectors. Based on our previous research, other clustering algorithms, like Hierarchical Agglomerative Clustering (HAC), behave in a similar way.

The word vectors resulted from the NIST dataset corpus, and the 920 tokens vocabulary are grouped in two main clusters, which can be observed visually in Figure 2. The elbow score, calculated and presented in Figure 3 using the KElbowVisualizer [33], suggests a number of 17 clusters, each containing an average of 54 tokens. This may lead to a very abstract and generic representation of the source code, and being unable to preserve the security weaknesses patterns. We have chosen to apply iterative clustering and obtain clusters with 10 tokens or less, balancing the desired independence on the programming language with an acceptable level of abstraction that still captures the security vulnerability patterns. We have generated with this approach 167 clusters of tokens. Examples of tokens' clusters are presented in Table 3.

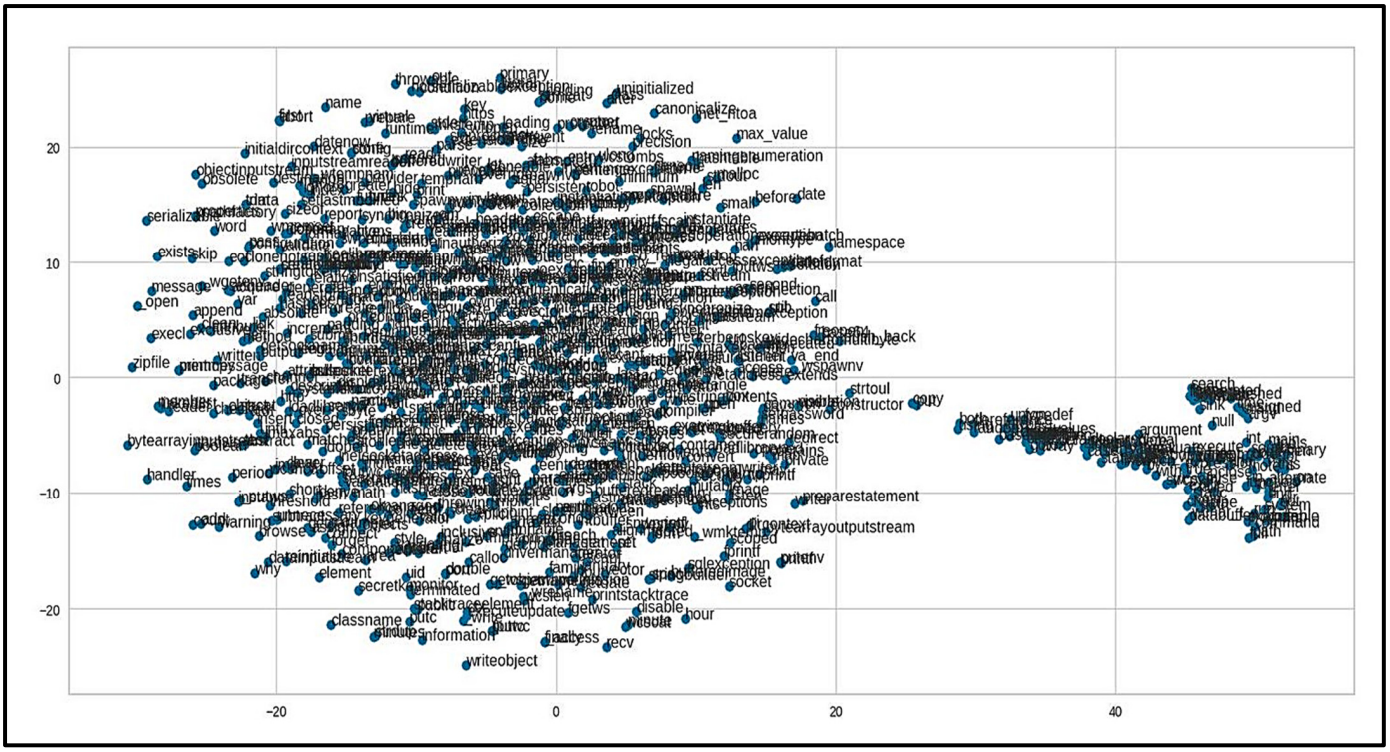

Figure 2. Bi-dimensional view of NIST dataset tokens' word vectors.

Table 3. Examples of clusters containing mixed tokens from C/C++, Java, and C# languages.

Cluster	Token	Token Found in:			Cluster	Token	Token Found in:		
		C/C++	Java	C#			C/C++	Java	C#
65	args			x	1	command		x	
	assign		x			executable		x	
	connect	x	x			program		x	
	pass		x			run		x	

Table 3. *Cont.*

Cluster	Token	Token Found in:			Cluster	Token	Token Found in:		
		C/C++	Java	C#			C/C++	Java	C#
	char	x	x	x		_stat	x		
	data		x			attempted		x	
	databuffer		x			hour		x	
8	pipe	x	x		156	increment		x	
	static	x	x	x		inputsource		x	
	void	x	x	x		process		x	x
	wchar_t	x				runtime		x	
	_wfopen	x				case	x	x	x
	_wgetenv	x				declared		x	
	abstract		x	x		else	x	x	x
	eof		x			fgets	x		
26	invoke		x		17	flags		x	
	sum		x			hit		x	
	test		x			rand	x		
	var		x	x		statement		x	
	verify		x			string		x	x

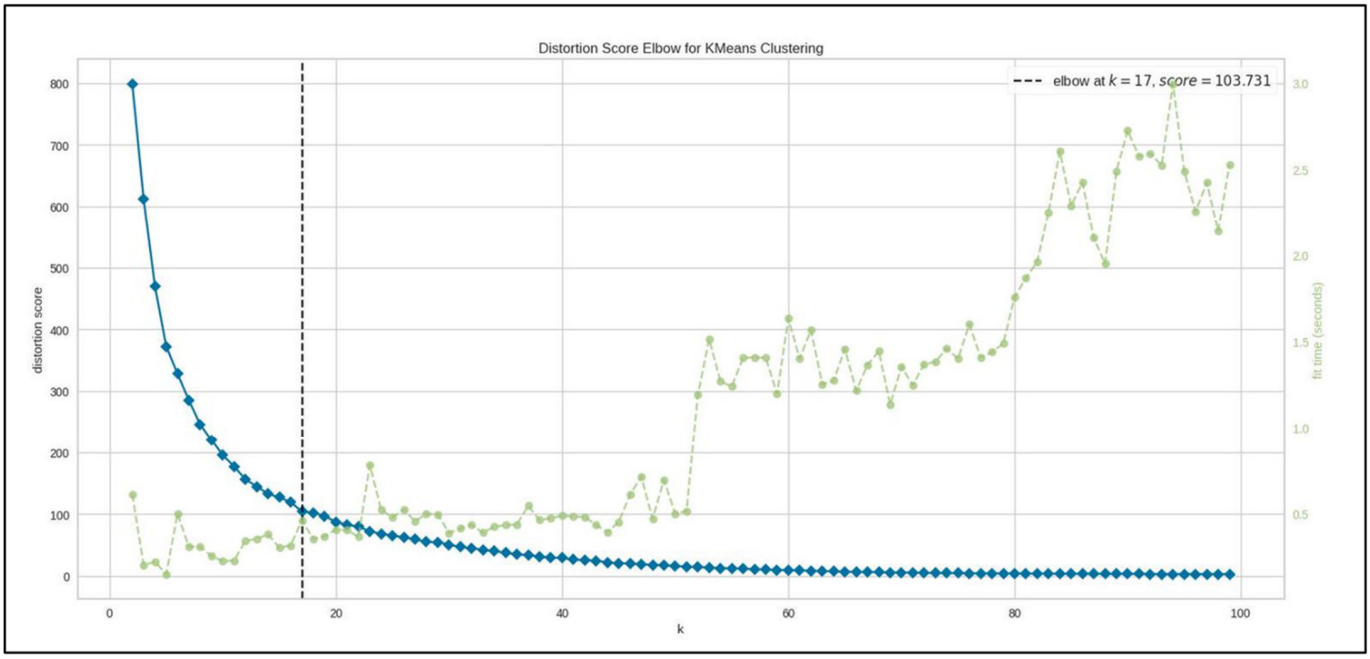

Figure 3. The Elbow method suggests a number of 17 clusters of tokens.

3.4. Source Code Tokenization

Source code snippets are tokenized using a specially designed parser, named Tokenizer, available at sergiuzaharia/CWE_Scanner [34]. The parser translates the source code snippets with "vulnerable" and "not vulnerable" code into the CLaSCoRe representation.

Before tokenization, all comments and pre-processing directives are removed from the code. The parser transforms the code in two different vectors: control flow and data flow. The rationale of this split is to capture the programming behavior embedded in the

source code as a specific succession of reserved keywords and instructions. The data flow section includes all other keywords (like identifiers, functions, and methods' given names) in a codified format, as defined in Section 3.4.2. The model is an evolutionary step of the Intermediate representation composed of the loosely coupled control and data flows, defined in [35] and improved in [36] and [8].

3.4.1. The Control Flow Vector

For the tokenization process, we use the vocabulary that was developed based on the method defined in Section 3.1. The vocabulary consists of the 920 keywords specific to C, C++, Java, and C# programming languages, which were identified as having at least one appearance in the NIST dataset for these languages. Source code tokens are checked against the vocabulary and, when identified, they are replaced with their respective cluster identification number, resulting from the clustering step defined in Section 3.3. For example, the token "else", which we can identify in source code written in C/C++, Java, or C#, is replaced with the integer value 17, according to Table 3.

3.4.2. The Data Flow Vector

The data flow vector includes all tokens which are not part of the vocabulary, like names given by the code writers to various functions, classes, variables, etc. As these identifiers may be named in infinite ways, depending on programmers' imagination or the software program's roles, the identifiers are transformed into a generic value, an integer capturing the order of appearance of the identifier in the source code. If one identifier appears more than once in the code snippet, it maintains the initial value, precisely the order of its first appearance in the code.

The pair of the control flow and data flow vectors represents the Cross-Language Source Code Representation (CLaSCoRe), being used as input to machine-learning-based classifiers as a core representation of the code.

4. Experimental Results

4.1. Learning Scenarios

The NIST dataset with "vulnerable" and "not vulnerable" code is used in two scenarios, illustrated in Table 4.

Table 4. Training scenarios on the NIST dataset with "vulnerable" and "not vulnerable" code.

Scenario	Training Dataset	Testing Dataset
C# to C#	C# code snippets	C# code snippets
Other to C#	C, C++, Java code snippets	C# code snippets

The "C# to C#" scenario considers training various machine learning algorithms on the C# dataset with "vulnerable" and "not vulnerable" code, to assess the quality of the CLaSCoRe representation. The C# dataset with code samples is split into training and testing sets with both "vulnerable" and "not vulnerable" code.

The "Other to C#" scenario is basically a zero-shot classifier for C#. It considers the training on datasets written in C, C++, and Java languages in order to detect security vulnerabilities (CWEs) in code written in C#. In this scenario, no C# code snippet is used during the training process.

4.2. Dataset Processing

The NIST dataset with "vulnerable" and "not vulnerable" code written in C, C++, Java, and C# was sanitized, and all comments and pre-processing directives were removed. We isolated "vulnerable" functions and "not vulnerable" functions per CWE, for the four programming languages, making use of NIST developers' convention to name "vulnerable" functions with "GOOD" and "not vulnerable" functions with "BAD". Examples

of "vulnerable" functions written in Java and C# are displayed in Table 2. We observe similar programming behavior in writing two pieces of code in C# and Java, specifically in the order of operations and usage of instructions, despite the lexical differences between these two programming languages. The processing units in the learning process represent functions or methods, as they are defined to include the entire context of vulnerabilities, a characteristic of the NIST suite.

In addition, the clustering of the word vectors resulting from the Word2Vec word embedding process adds a layer of abstraction to the source code written in different programming languages, preserving the semantic meaning. We observe in Table 2 that the token name "Runtime" in the Java code snippets and the token named "Process" in the C# code of a CWE 78 vulnerable function, are part of the same cluster, which means they will have the same value in the CLaSCoRe representation, as they would represent the same keyword. The clustering of the word vector is, in this way, reducing to some extent the dependency on the programming language.

We have identified 50 CWEs, out of the 178 CWEs in the NIST Software Assurance Reference Dataset (SARD) suite, supported with code in all four programming languages, and only 8 CWEs for which we can sum up a number of more than 1500 samples written in C, C++, and Java, with at least 250 samples per each of the four programming languages. The CWE 134 vulnerability samples have also been excluded, as they behave as outliers for all previous [8,35,36] and current experiments, producing low-performance results. The list of remaining CWEs included in the experiments and their associated numbers of vulnerable samples are illustrated in Table 5.

Table 5. The number of vulnerable code samples in C, C++, Java, and C# per CWE.

	Vulnerable Code Samples				
CWE Name	**C#**	**C++**	**C**	**Java**	**Total Samples Written in C, C++, and Java**
CWE190	6039	1404	5040	7015	13,459
CWE191	4026	858	3864	5612	10,334
CWE78	600	2200	5600	720	8520
CWE369	2562	468	1008	3050	4526
CWE789	2177	1080	560	2537	4177
CWE400	2013	390	840	2396	3626
CWE197	8100	396	1008	1980	3384
CWE606	610	260	560	732	1552

For each programming language, we selected the methods and functions named "GOOD" in the NIST dataset and labeled them as "not vulnerable". The number of "not vulnerable" samples, after being represented in a CLaSCoRe format and removing duplicated entries, are represented in Table 6. The "not vulnerable" functions contain secure versions of all the vulnerable code snippets, meaning that, in the training dataset used for a specific CWE, we included the "vulnerable" code snippets to the respective CWE, and secured code snippets that were highly relevant to the respective CWE (fixing that vulnerability) and code which were not related to the CWE, but were free of other CWEs present in the NIST suite.

For the learning scenarios presented below, we retained a randomized 80% of the dataset samples for learning and the rest of the 20% for testing. Cross-validation using 5 groups of samples, with similar sizes, from the training set was enabled to support the calculation of accuracy means and standard deviations in both scenarios.

Table 6. The number of "not vulnerable" code samples in the NIST suite, per programming language.

Programming Language	"Not Vulnerable" Code Samples
C#	14,323
C++	13,341
C	19,457
Java	15,171

4.3. Learning Scenario "C# to C#"

The NIST dataset with C# code has been used in training the machine learning algorithms included in the Scikit-learn platform, to detect CWE patterns in source code. The results are similar in quality to our previous experiments on C, C++, and Java, which are illustrated in Table 7. We observe the very good performance of all machine learning algorithms to detect CWE patterns in the CLaSCoRe representation of the NIST dataset code snippets which are written in C#.

Table 7. F1 Score Average per machine learning algorithm, for the 8 CWEs in scope, in the "C# to C#" and, respectively, "Other to C#" learning scenarios.

	F1 Score Average			
Machine Learning Algorithm	**K-NN**	**Decision Tree (DT)**	**Random Forests (RF)**	**Support Vector Machines (SVM)**
Scenario "C# to C#"	0.97	0.98	0.99	0.90
Scenario "Other to C#"	0.63	0.45	0.43	0.72

The machine-learning-based classifiers have been configured as below:

- SVM (SVC): class_weight = None, kernel = rbf;
- Decision Tree: splitter = best, criterion = gini;
- KNeighborsClassifier: n_neighbors = 7, weights = uniform, algorithm = auto;
- Random Forests: n_estimators = 100, criterion = gini.

4.4. Learning Scenario "Other to C#"

The "Darwinian" evolution of programming languages resulted in lexical and semantical similarities between them. As an example, the C# programming language has been developed starting from C++ and Java languages. Before, C++ was influenced by C and Algol, the last one being influenced by Fortran [37]. We derive from here that C# is directly influenced by C, C++, and Java, and indirectly influenced by Algol and its predecessor, Fortran. We intend to make use of C# inherited characteristics from C, C++, and Java, in detecting security weaknesses in C# source code by learning their patterns in code written in other languages, like C, C++, and Java.

As a result, we trained the same machine learning algorithms used for the "C# to C#" scenario to detect CWEs on NIST code written in C, C++, and Java, and tested them on NIST code written in C#. As seen in Table 2, the code snippets of the NIST dataset have some similarities in the order of operations and in the overall business logic, as they are created by the same community of programmers. Despite some differences between the lexical and semantic structures, they are also visible. Our intention is to observe how these two properties of the code, the influence of their ancestor languages and the similar behavior of writing code within a community of programmers, may help security analysts detect weaknesses when they lack static analysis tools, skills, or datasets for the downstream or exotic languages used by the developers in their organizations.

According to the results displayed in Table 7, the Support Vector Machines (SVM) algorithm is able to detect CWE patterns in source code written in C#, when no C# code has been used in training if we use code written in C, C++, or Java during the training phase. Other machine learning algorithms did not perform well enough.

The results should be positioned and valued in the following context:

- Not all possible C# code tokens were included in the vocabulary during the experiments; for example, the Windows API tokens (e.g., the "GetEnvironmentVariable" token observed in the C# code section of Table 2) or the .NET tokens.
- The word embeddings were applied to the NIST corpus with C, C++, Java, and C# synthetic code, with word similarities being identified in this limited context. In the next stages, the word similarities will be generated on a much larger corpus of code from public sources, which will lead to new and more relevant clusters per each subset of tokens.

The detailed results for the Support Vector Machines classifier applied to the 8 CWEs in scope are presented in Table 8.

Table 8. F1, recall, precision, and accuracy scores of the Support Vector Machines classifier applied to the 8 CWEs in scope, in the "C# to C#" and, respectively, "Other to C#" learning scenarios.

CWE Name	Scenario "Other to C#"				Scenario "C# to C#"			
	F1 (SVM)	Recall (SVM)	Precision (SVM)	Accuracy Mean (SVM)	F1 (SVM)	Recall (SVM)	Precision (SVM)	Accuracy Mean (SVM)
CWE190	0.75	0.82	0.69	0.91	0.95	0.94	0.96	0.97
CWE191	0.73	0.82	0.67	0.91	0.93	0.94	0.93	0.97
CWE78	0.72	0.80	0.65	0.91	0.82	0.99	0.70	0.98
CWE369	0.75	0.78	0.71	0.86	0.83	0.95	0.74	0.95
CWE789	0.68	0.78	0.61	0.95	0.91	0.94	0.89	0.97
CWE400	0.70	0.82	0.61	0.83	0.94	0.98	0.90	0.98
CWE197	0.69	0.80	0.60	0.87	0.96	0.94	0.99	0.96
CWE606	0.77	0.75	0.79	0.88	0.82	0.96	0.72	0.98

We can conclude that our method allows security analysts to identify security weaknesses in C# source code with acceptable accuracy when they lack a C# dataset to train the machine learning algorithms, but they have access to labeled code written in C, C++, and Java, from the same community of programmers. The accuracy averages at 0.89 in the "Other to C#" scenario with a standard deviation averaging at 0.054 for all CWEs in scope. For the "C# to C#" scenario, the accuracy mean is 0.97 with a standard deviation averaging at 0.017.

We obtained an average recall score equal to 0.80 in the "Other to C#" scenario for SVM and a recall score equal to 0.95 in the "C# to C#" scenario, for the same machine learning algorithm. This result can be interpreted in the following way: if we are able to identify 95% of the vulnerable code in a C# dataset, using a method based on CLaSCoRe representation and a C# dataset for training, in the same way, we are able to identify 80% of the vulnerable code in the C# dataset after training the algorithm to detect security vulnerabilities in C, C++, and Java code written by programmers within the same community of developers.

The performance of CLaSCoRe representation in the "Other to C#" scenario, as zero-shot classification, has been in average F1 = 0.80 on the NIST SARD dataset, and 0.95 in the "C# to C#" scenario, better than in similar approaches [26]. The advantage of learning from multiple languages that are linked together by a lexical and semantical inheritance is essential for zero-shot software security scanners. More, the loosely coupled control and data flow vectors make the ClaSCoRe representation partially language agnostic and, at the same time, able to capture the patterns of security vulnerabilities (CWEs) which are shaped by both reserved keywords, methods, functions, and developers' defined variables being processed by them. The model is able to partially fill the gap between the structure,

lexical, and semantic differences between programming languages, especially when they are strongly interconnected on the language influence map.

The inheritance at lexical and semantical levels, transmitted "genetically" to newer programming languages from their "ancestors", is valuable to security analysts when datasets and skills are limited for new, not-yet-popular languages. Programs written in downstream languages, influenced by more popular ones, do not benefit, in general, from the same number of software security experts and technologies to detect security weaknesses (CWEs) in the early versions of applications developed by the same community of programmers. Using the CLaSCoRe representation of source code, we were able to detect security vulnerability patterns in software written in a different language than the ones used for training, as long as the code was written in a downstream language, and we had enough samples in upstream languages.

We observe that most graph-based representations are not identified as cross-language or language agnostic, their dependency on the programming language being higher than in our model. We group the tokens in clusters based on their semantic similarity, making the ClaSCoRe representation less dependent on the lexical structure of programming languages and portable between them.

The ClaSCoRe representation performs very well in the "C# to C#" scenario, similar to or better than most of the models described in Section 2, with an average F1 score of 0.96 for all CWEs in scope and all machine learning algorithms tested, which indicates that the semantical representation and clustering of tokens included in the multi-language source code provided by the community of developers are beneficial as well in single language models.

5. Conclusions

The ability to identify security vulnerabilities within source code written in programming languages that are not well supported with security scanning tools, software security analysis, and labeled datasets, using the collective knowledge acquired during the process of securing more traditional applications developed in popular languages, increases the overall resilience of the global software surface. Communities of developers with similar behavior in writing source code may use this characteristic to identify vulnerabilities in modules and functions written in programming languages less supported by security tools, experts, and datasets, as long as they have developed similar modules in popular languages supported by static analysis technology and are well connected on the languages influence map. Identifying 80% of security flaws implies reducing the attack surface significantly, considering also that new and less popular languages are also targeted by fewer cyber criminals, the know-how being limited on both sides.

We conclude that ClaSCoRe, a core representation of source code for machine=learning-based classifiers, is less dependent on the programming language than the original code and still able to preserve the lexical and semantical representation of the security weaknesses which may be part of it.

The potential limitations and challenges faced by our proposed representation are:

- The availability of large, labeled datasets with vulnerable code for multiple CWEs within the communities of developers.
- The level of source code abstraction, which may reduce the influence of behavioral patterns of vulnerable and clean code writing.

Following our vision, we plan to advance our research in the future in two main directions:

- To include more C, C++, Java, and C# code in the word embedding process to create more realistic word similarities between the programming language tokens.
- To include more tokens in the vocabulary, from all major software development frameworks and libraries used by programmers around the world, increasing the ClaSCoRe representation footprint.

- To expand the zero-shot model to other programming languages that are more different from the training languages (from a lexical and semantical perspective) since, in our case, C# is quite close to C/C++ and Java languages.

Author Contributions: Conceptualization, S.Z., T.R. and S.T.-M.; data curation, S.Z. and T.R.; formal analysis, S.Z. and T.R.; investigation, S.Z. and T.R.; methodology, S.Z., T.R. and S.T.-M.; project administration, T.R. and S.Z.; resources, T.R. and S.T.-M.; software, S.Z.; supervision, T.R. and S.T.-M.; validation, S.Z., T.R. and S.T.-M.; visualization, S.Z.; writing—original draft, S.Z.; writing—review and editing, S.Z., T.R. and S.T.-M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. The Global Risks Report 2023, by the World Economic Forum. Available online: https://www.weforum.org/reports/global-risks-report-2023 (accessed on 20 June 2023).
2. Online Historical Encyclopaedia of Programming Languages. Available online: https://hopl.info/ (accessed on 20 June 2023).
3. TIOBE Index. Available online: https://www.tiobe.com/tiobe-index/ (accessed on 20 June 2023).
4. Wikipedia, List of Programming Languages. Available online: https://en.wikipedia.org/wiki/List_of_programming_languages (accessed on 20 June 2023).
5. National Vulnerability Database, NIST. Available online: https://nvd.nist.gov/vuln (accessed on 20 June 2023).
6. MITRE List of Security Weaknesses. Available online: https://cwe.mitre.org/data/index.html (accessed on 20 June 2023).
7. NIST datasets for C, C++, Java, C#. Available online: https://samate.nist.gov/SARD/test-suites (accessed on 20 June 2023).
8. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. Machine Learning-Based Security Pattern Recognition Techniques for Code Developers. *Appl. Sci.* **2022**, *12*, 12463. [CrossRef]
9. Yamaguchi, F.; Wressnegger, C.; Gascon, H.; Rieck, K. Chucky. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security—CCS '13, Berlin, Germany, 26 December 2013. [CrossRef]
10. 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 May 2014; pp. 590–604. [CrossRef]
11. Suneja, S.; Zheng, Y.; Zhuang, Y.; Laredo, J.; Morari, A. Learning to map source code to software vulnerability using code-as-a-graph. *arXiv* **2020**, arXiv:2006.08614. [CrossRef]
12. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781. [CrossRef]
13. Scarselli, F.; Gori, M.; Tsoi, A.C.; Hagenbuchner, M.; Monfardini, G. The graph neural network model. *IEEE Trans. Neural Netw.* **2008**, *20*, 61–80. [CrossRef] [PubMed]
14. 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 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018. [CrossRef]
15. 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]
16. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *39*, 1137–1149. [CrossRef] [PubMed]
17. Xuan, C.D.; Son, V.N.; Duc, D. Automatically detect software security vulnerabilities based on natural language processing techniques and machine learning algorithms. *J. ICT Res. Appl.* **2022**, *16*, 70–87. [CrossRef]
18. Saha, T.; Al-Rahat, T.; Aaraj, N.; Tian, Y.; Jha, N.K. ML-FEED: Machine Learning Framework for Efficient Exploit Detection (Extended version). *arXiv* **2023**, arXiv:2301.04314. [CrossRef]
19. Joulin, A.; Grave, E.; Bojanowski, P.; Douze, M.; Jégou, H.; Mikolov, T. Fasttext. Zip: Compressing text classification models. *arXiv* **2016**, arXiv:1612.03651. [CrossRef]
20. Wang, J.; Xiao, H.; Zhong, S.; Xiao, Y. DeepVulSeeker: A Novel Vulnerability Identification Framework via Code Graph Structure and Pre-training Mechanism. *Future Gener. Computer Systems. arXiv* **2022**, arXiv:2211.13097. [CrossRef]
21. Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; Yin, J. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv* **2022**, arXiv:2203.03850. [CrossRef]

22. Khoury, R.; Avila, A.R.; Brunelle, J.; Camara, B.M. How Secure is Code Generated by ChatGPT? *arXiv* **2023**, arXiv:2304.09655. [CrossRef]
23. Nair, M.; Sadhukhan, R.; Mukhopadhyay, D. Generating Secure Hardware Using ChatGPT Resistant to CWEs. Cryptology ePrint Archive. 2023. Available online: https://eprint.iacr.org/2023/212 (accessed on 20 June 2023).
24. Sobania, D.; Briesch, M.; Hanna, C.; Petke, J. An analysis of the automatic bug fixing performance of ChatGPT. *arXiv* **2023**, arXiv:2301.08653. [CrossRef]
25. Pan, C.; Lu, M.; Xu, B. An Empirical Study on Software Defect Prediction Using CodeBERT Model. *Appl. Sci.* **2021**, *11*, 4793. [CrossRef]
26. Chauhan, A. Machine Learning Based Cross-Language Vulnerability Detection: How Far Are We. The University of Texas at Dallas. 2020. Available online: https://utd-ir.tdl.org/handle/10735.1/8810 (accessed on 18 June 2023).
27. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. Code2Vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages, 3(POPL)*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1–29. [CrossRef]
28. The Open Group. Available online: https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html (accessed on 20 June 2023).
29. Java™ Platform, Standard Edition 8. Available online: https://docs.oracle.com/javase/8/docs/api/index-files/index-1.html (accessed on 20 June 2023).
30. Microsoft C# Guide/Language Reference. Available online: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ (accessed on 20 June 2023).
31. Natural Language Toolkit. Available online: https://www.nltk.org/ (accessed on 20 June 2023).
32. Mimno, D.; Wallach, H.; Naradowsky, J.; Smith, D.A.; McCallum, A. Polylingual topic models. In Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, Singapore, 6–7 August 2009; pp. 880–889. [CrossRef]
33. Bengfort, B.; Gray, L.; Bilbro, R.; Prema, R.; Patrick, D.; McIntyre, K.; Morrison, M.; Ojeda, A.; Schmierer, E.; Morris, A. Yellowbrick v1.5., Visualizing the Scikit-Learn Model Selection Process. *OpenAIRE* **2022**. [CrossRef]
34. GitHub Repository of ClaSCoRe Source Code. Available online: https://github.com/sergiuzaharia/CWE_Scanner (accessed on 20 June 2023).
35. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. Source code vulnerabilities detection using loosely coupled data and control flows. In Proceedings of the 2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, 4–7 September 2019. [CrossRef]
36. Zaharia, S.; Rebedea, T.; Trausan-Matu, S. CWE pattern identification using semantical clustering of programming language keywords. In Proceedings of the 23rd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 26–28 May 2021. [CrossRef]
37. Influence of Programming Languages. Available online: https://rigaux.org/language-study/diagram.html (accessed on 20 June 2023).