applied sciences

MDPI

Article

Machine Learning-Based Security Pattern Recognition Techniques for Code Developers

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
[2] Institute for Artificial Intelligence "Mihai Draganescu" of the Romanian Academy, 050711 Bucharest, Romania
* Correspondence: traian.rebedea@upb.ro

Abstract: Software developers represent the bastion of application security against the overwhelming cyber-attacks which target all organizations and affect their resilience. As security weaknesses which may be introduced during the process of code writing are complex and matching different and variate skills, most applications are launched intrinsically vulnerable. We have advanced our research for a security scanner able to use automated learning techniques based on machine learning algorithms to recognize patterns of security weaknesses in source code. To make the scanner independent on the programming language, the source code is converted to a vectorial representation using natural language processing methods, which are able to retain semantical traits of the original code and at the same time to reduce the dependency on the lexical structure of the program. The security flaws detection performance is in the ranges accepted by software security professionals (recall > 0.94) even when vulnerable samples are very low represented in the dataset (e.g., less than 4% vulnerable code for a specific CWE in the dataset). No significant change or adaptation is needed to change the source code language under scrutiny. We apply this approach on detecting Common Weaknesses Enumeration (CWE) vulnerabilities in datasets provided by NIST (Test suites–NIST Software Assurance Reference Dataset).

Keywords: software security engineering; machine learning; code embeddings; common weakness enumeration

Citation: Zaharia, S.; Rebedea, T.; Trausan-Matu, S. Machine Learning-Based Security Pattern Recognition Techniques for Code Developers. *Appl. Sci.* **2022**, *12*, 12463. https://doi.org/10.3390/app122312463

Academic Editors: Andrea Prati and Paolino Di Felice

Received: 25 October 2022
Accepted: 2 December 2022
Published: 6 December 2022

1. Introduction

Source code remains one of the main entry points of hackers in business-critical applications or critical infrastructures, as result of inexperienced developers in secure coding or lack of supporting technologies in writing non-vulnerable code. The software security community uses two terms to categorize vulnerabilities which may exist in software products: Common Weakness Enumeration (CWE) and Common Vulnerabilities and Exposures (CVE).

1.1. Common Weakness Enumeration (CWE)

Technical vulnerabilities are maintained by Massachusetts Institute of Technology Research and Engineering (MITRE) (MITRE CWE repository: https://cwe.mitre.org/, last accessed on 20 August 2022) as Common Weakness Enumeration (CWE) items. Currently (in August 2022), there are 927 weaknesses in the inventory, out of which 418 weaknesses (representing 45.09%) are specific to the software development phase. These weaknesses are grouped in 40 categories, such as "Authentication Errors" or "Authorization Errors".

The CWEs are maintained with different levels of abstraction, to cover all programming languages being currently active. However, weaknesses specific to popular programming languages are grouped and labeled as such. For example, there are 82 weaknesses specific to C language, 86 are specific to C++, 77 associated with Java, and 25 with PHP.

Hardware design flaws are increasingly included into the list maintained by MITRE, currently the inventory including 100 weaknesses (representing 10.78% of the CWEs). The

current research covers only the software development weaknesses which are identifiable in source code.

1.2. Common Vulnerabilities and Exposures (CVE)

The CVEs (Common Vulnerabilities and Exposures), maintained by the same organization (MITRE CVE repository: https://cve.mitre.org/index.html, last accessed on 24 October 2022, in process of migration to https://www.cve.org/, last accessed on 20 August 2022), are related to specific instances of a vulnerability in different products or systems, being made public with various levels of technical details.

There are currently (20 August 2022) 182,763 vulnerabilities and exposures listed as CVEs, their number growing from year to year. According to MITRE metrics (MITRE CVE metrics: https://www.cve.org/About/Metrics#PublishedCVERecords, last accessed on 20 August 2022), 18,375 CVEs have been discovered in 2020, 20,161 CVE in 2021, and in the first half of 2022 a number of 12,380 have been already discovered and published. These vulnerabilities and exposures of various products and systems have different levels of severity, calculated as a Base Score with values from 0 (None/Low Severity) to 10 (Critical Severity), provided per each CVE by the National Vulnerability Database (US National Institute of Standards and Technologies (NIST) National Vulnerability Database: https://nvd.nist.gov/, last accessed on 20 August 2022) (NVD). The score depends on the level of exploitability and the impact once the vulnerability is successfully used by attackers.

The vulnerabilities and exposures identified for specific products and published as CVEs are real-time implementations of CWE technical weaknesses. One CVE may be mapped to multiple CWEs and the opposite, one CWE may be the cause of multiple products vulnerabilities (CVEs). For example, one of the most well-known weaknesses, CWE 89 (Improper Neutralization of Special Elements used in an SQL Command/'SQL Injection') corresponds to 1263 CVEs (to be read as vulnerable software products) in the NVD database, with an average severity of 8.66 (High Severity), which makes it the third security flaw of year 2022 in Top 25 CWE maintained by MITRE.

1.3. Identification of Software Security Vulnerabilities

Being able to identify and fix the CWEs early in the software development phase, before a product-specific CVE is discovered and published, is critical to achieving application resilience to cyber-attacks. On the other hand, cyber criminals have no intention to declare the identified vulnerabilities, as their exploitation will not be possible anymore, so reducing the CWE number in the source code by the programmers and security analysts remains the best and more economical solution to protect the applications against malicious actors.

Despite the lack of well-documented benchmarks regarding the number of weaknesses accepted by security professionals in business-critical or not critical applications, there are analyses such as the one run by CAST Research Labs [1] showing that the average density of well-known security flaws (only 35 CWEs checked) for NET and Java EE applications is between 30 and 100 per 10k lines of codes (LOC), with outliers, meaning "insecurely developed applications", providing more than 350 CWEs per 10k lines of code. If we consider the entire palette of CWEs managed by MITRE (418 CWEs for Software Development), there is a good assumption to consider that the density of weaknesses in software applications is higher than 300 CWEs per 10k LOC.

As manual detection of technical security flaws in source code requires a combination of highly skilled professionals in software security and heavy routine-based activities at the same time, which is always hard to find, there is a strong need for solutions being able to scan source code automatically and identify code-level security vulnerabilities early in the software development phase. Currently there are technology providers and open communities which maintain Static Analysis Security Testing (SAST) solutions. These technologies are able to browse line by line the source code written in the most popular programming languages and development frameworks and identify the technical vulnerabilities. Depending on the maturity of the solution, the weaknesses may be mapped automatically

to various standards (e.g., Top 10 OWASP, Open Web Application Security Project, Top 10 Security Weaknesses: https://owasp.org/Top10/, last accessed on 20 August 2022) and regulations (e.g., GDPR, General Data Protection Regulation (EU) 2016/679), and the mature commercial solutions are able to understand up to 20–30 programming languages.

1.4. Machine Learning Approaches in Detecting Softare Security Vulnerabilities

Organizations which develop important volumes of source code may be needed to invest important budgets for tools and services meant to identify and secure their sensitive applications and the products embedding them. However, in addition to application security, budgets at CISO's (Chief Information Security Officer) disposal have to cover a widely distributed palette of security topics, any potential for optimization being continuously explored by security professionals. They should consequently be able to complement software security scanning by commercial SAST technologies with freeware solutions able to give a good indication if a specific vulnerability is present in a new piece of code written by the programmers in a specific organization or members of a developers' community. In this context, we observe a positive trend in researching machine learning-based solutions for source code or binary code analysis, for detecting security weaknesses. Part of them [2,3] are designed to identify CVEs in source code, especially in open-source libraries, modified or unaltered, used in complex software products. Other solutions [4–10] are intended for CWE detection in specific programming languages (e.g., C/C++). These solutions are complementary to SAST technologies, which are still considered by security experts the most resilient approaches in the identification of software weaknesses in the early development stages.

1.5. Problem Statement and Research Objectives

The volume of source code developed by software companies or programmers' communities is continuously increasing as a result of technology trends. For example, the software included into a connected and autonomous car is estimated to reach more than 100 million lines of code in different programming languages and the next generation networks and cloud infrastructures are transforming into software defined networks. Securing the software is critical to build trust of people in technological evolution. There is a strong need to support software developers with tools able to identify the security flaws within their code as early as possible, independent of the programming languages used by their organizations or communities. Complementing SAST technologies with machine learning-based solutions is a cost-effective way to reduce the software related weaknesses.

The main objective of our research is to provide developers a practical solution for security weaknesses detection within source code, language agnostic, and leveraging the behavioral patterns of programmers of large organizations and communities in writing code. This solution is not meant to replace existing technologies such as SAST or FOSS (Free and Open-Source Software) scanners, but to complement them in order to reduce costs and to increase programming languages coverage.

1.6. Main Contributions

We propose a programming language agnostic solution for CWE patterns identification in source code. The concept is based on source code split in two vectorial representations: control flow and data flow. Control flow contains the possible reserved keywords, methods, functions, methods which are usually found in source code and the data flow contains all the other words which are not part of the control flow (e.g., identifiers). The control flow section is processed in two ways: by word embeddings (e.g., Word2Vec) to preserve the semantics of the code, and by clustering (e.g., K-Means) of the resulting word embedding vectors (e.g., semantically represented tokens) which lose the dependency of the lexical syntax of the programming language.

We experimented with a suite of machine learning algorithm on the resulting representation of source code to identify how CWE pattern in both C/C++ and Java can

be detected in this abstract representation, with good results in detection with low false positives rates, which makes this approach a good candidate to contribute in a holistic software security assurance process. Our approach is feasible mostly when source code is developed within communities of software developers with similar programming behavior (e.g., software hubs).

In our experiments we used the NIST Software Assurance Reference datasets for vulnerable and not-vulnerable code written in C/C++ and Java. These datasets are provided by NIST as a reference set to test the capabilities of source code static analysis tools. The code samples have a high level of abstraction but also preserve some similarities in the way the programs are written, close to what strongly connected communities of programmers in software hub have in common.

The next sections of this document are structured as follows:

- Chapter 2—The Proposed Architecture: describes the functional architecture of the proposed solution, which transforms the source code into an Intermediate Representation which preserves the CWE patterns with low dependency of the lexical structure of programing language.
- Chapter 3—Experimental Results: describes our tests on the source code dataset provided by NIST, including the dataset processing, the word embedding and clustering mechanism, the control flow and data flow representation and result obtained using well-known machine learning-based algorithms.
- Chapter 4—Discussion and related work: position our approach in the overall context of machine learning-based security weaknesses detection in source code.
- Chapter 5—Conclusions: presents our conclusions and next steps related to the multi-language CWE detection approach.

2. Proposed Architecture

For organizations with mature and well-organized software development teams, the patterns of writing source code are in general maintained from an application to another, or from a release to another release of the same application. In addition, developers follow periodic programming trainings, where they learn how to implement different functions or workflows in code.

The inherited culture of programming workflows opens an important opportunity for innovation in CWE patterns identification in source code, by creating a dataset of vulnerable and not-vulnerable pieces of software created by an organization or community of developers, which can be used to predict if newly developed code is vulnerable. When developers omit to write securely a similar piece of code, the method presented in this research is able to indicate the vulnerabilities without using SAST technologies at large scale, thus reducing the commercial tools expenses.

Highly abstract code provided by different developers are less prone to preserving programmers' cultural patterns inherited within specific communities and may need additional samples in the training datasets, to produce valuable outcomes.

We are enhancing and expanding the experiments on our multi-language CWE scanner [11,12] to the entire database with vulnerable source code in C/C++ and Java, provided by NIST, and provide the developers community a solution to identify vulnerabilities in source code written virtually in any programming language, as long as they can use their previous code as a sample dataset for training the classifiers based on machine learning algorithms.

The NIST reference datasets include vulnerable and non-vulnerable code snippets, as follows:

- 64,099 test cases for C/C++, covering 118 different CWEs,
- 28,881 test cases for Java, covering 112 CWEs.

The following architecture is intended to represent source code written in various programming languages (e.g., C/C++, Java) into an unitary conceptualized format, named Intermediate Representation (IR) by convention, less dependent on the lexical structure of each specific language. The IR must still preserve the technical vulnerability patterns

(CWEs) as to be detected by machine learning-based classifiers, despite being less dependent on the lexical and semantical structure defined for the programming languages used to write the code.

The proposed architecture for the multi-language CWE scanner, as presented in Figure 1, is based on a parser able to tokenize the source code in two sections: control flow and data flow. The control flow section maintains the reserved and predefined keywords, functions, classes, methods for the programming language (e.g., if, else, void). The data flow contains all other identifiers from the source code (e.g., names of variables).

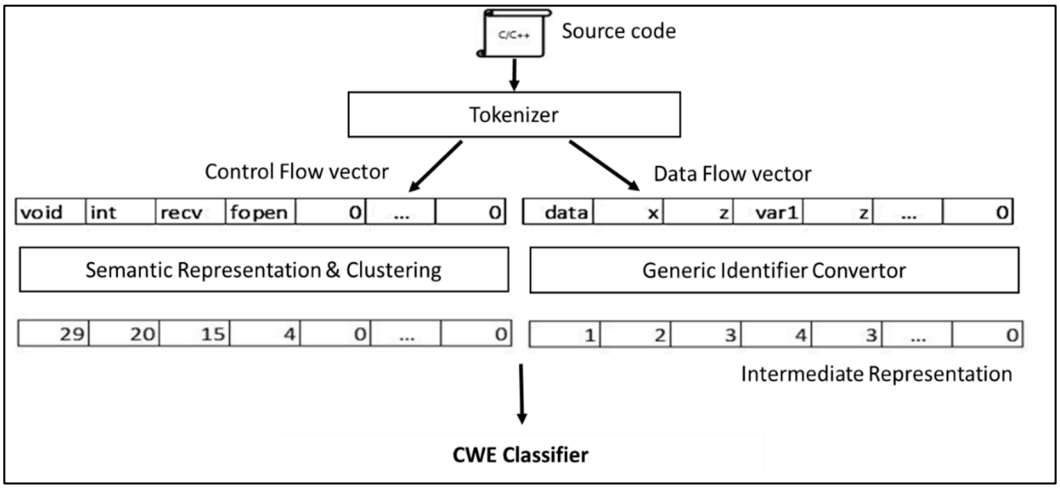

Figure 1. Architecture of the proposed multi-language CWE scanner using Intermediate Representation.

The vocabulary of control flow tokens is selected as indicated in the Section 2.1. The tokenizer creates the control flow section by selecting only the tokens included in the vocabulary, in their order of appearance. The control flow section is represented by a vector of tokens, of size 250. The remaining vector elements, after all tokens in the source code snippet were parsed by the tokenizer, are padded with "0".

The data flow section is represented by a vector of size 250, built as indicated in Section 2.3. The remaining vector elements, after all words not being part of the tokens' vocabulary in the source code snippet received by the tokenizer were parsed, are padded with "0".

2.1. The Control Flow Section of the Intermediate Representation

We have compiled an exhaustive list of potential tokens which may be found in C/C++ and Java programs, starting from various frameworks and lists of keywords or system functions. The final vocabulary for the control flow section consists of 19,775 tokens collected for Java programs and 1782 tokens collected for C/C++ programs, totaling 21,426 unique tokens. It is important to notice that the two sets of tokens have some overlap, as 131 tokens are common to both Java and C/C++.

The list of 1782 keywords which are relevant for the control flow of a C/C++ program includes the 84 keywords for C++, the POSIX.1-2017 identifiers and functions from The Open Group (https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html, last accessed on 20 August 2022), and Linux or Windows functions which we may find in source code written in C/C++.

In the same way, the 17,775 tokens which we may find in Java programs, are collected from open sources (https://docs.oracle.com/javase/8/docs/api/index-files/index-1.html, last accessed on 20 August 2022), and include reserved keywords and predefined classes, methods, variables, operators, constructors, or interfaces.

We have found only 309 tokens from our proposed vocabulary with more than one appearance in the entire NIST dataset with Java and C/C++ code. A sample of the 309 tokens for C/C++ and Java, identified in NIST datasets, is displayed in Table 1.

Table 1. A sample of C/C++ and Java Tokens identified in NIST datasets, ordered alphabetically (part of the control flow section).

No.	Token Name
1	absolute
2	abstract
3	access
4	action
5	add
6	addheader
7	addition

2.2. Word Embeddings and Semantic Clustering

We identify semantical similarities between tokens in the control flow and group them in clusters accordingly. We obtain an abstract form which is able to preserve the CWE patterns, and at the same time is less dependent on the precise lexical structure of the program.

Semantical grouping considers the Word Embeddings [13] concept widely used in Natural Language Processing in recent years. For the semantic representation of the control flow tokens, we have used the Gensim (https://pypi.org/project/gensim/, last accessed on 24 October 2022) platform to compute Word2Vec embeddings starting from source code, combined with NLTK (https://www.nltk.org/, last accessed on 24 October 2022) (Natural Language Toolkit) for tokenization, which produced similarity values for each of the 309 tokens. The list of 309 tokens represents the vocabulary for tokenization and semantical transformation of the source code. To identify the optimal Intermediate Representation of the source code in conjunction with the machine learning ability to detect security weaknesses patterns, we have used the NIST dataset tokenized with these 309 tokens and the K-Means clustering algorithm for the resulting word embedding vectors (from Word2Vec), configured as in Equation (1). The Hierarchical Agglomerative Clustering (HAC) algorithm has been tested as an alternative, with similar results.

$$KMeansClusterer(no\_of\_clusters,\ distance = nltk.cluster.util.cosine\_distance,\ repeats = 25,\ avoid\_empty\_clusters = True) \quad (1)$$

In Figure 2, we present a bi-dimensional representation for 2 (out of 50) K-Means clusters of control flow tokens. Tokens are represented by their corresponding word embeddings vectors, resulting from the Word2Vec transformation. Clusters contain between 2 and 22 tokens. One of the illustrated clusters includes the following tokens: {addition, array, bounds, decrement, if, increment, index, size, src, this}.

2.3. The Generic Identifiers (Data Flow Section of the IR)

The identifiers for the data flow are keywords which are not part of the tokens inventory used to tokenize the control flow section of the Intermediate Representation. After being identified, they are converted to natural numbers reflecting their order of appearance in the original source code. The data flow section of the Intermediate Representation simulates a symbol table under the form of (Id1, Id2, . . . , Id20), which preserves only the order of identifiers (e.g., variables names, functions names) in the source code. Identifiers which appear more than once in the source code maintain their initial value in the data flow vector. Identifiers' names are not preserved in the data flow section of the resulting Numeric Intermediate Representation.

The Intermediate Representation is input to CWE classifiers, which are trained to detect the security vulnerability patterns in source code written in C/C++, Java or any other language whose tokens are part of the inventory.

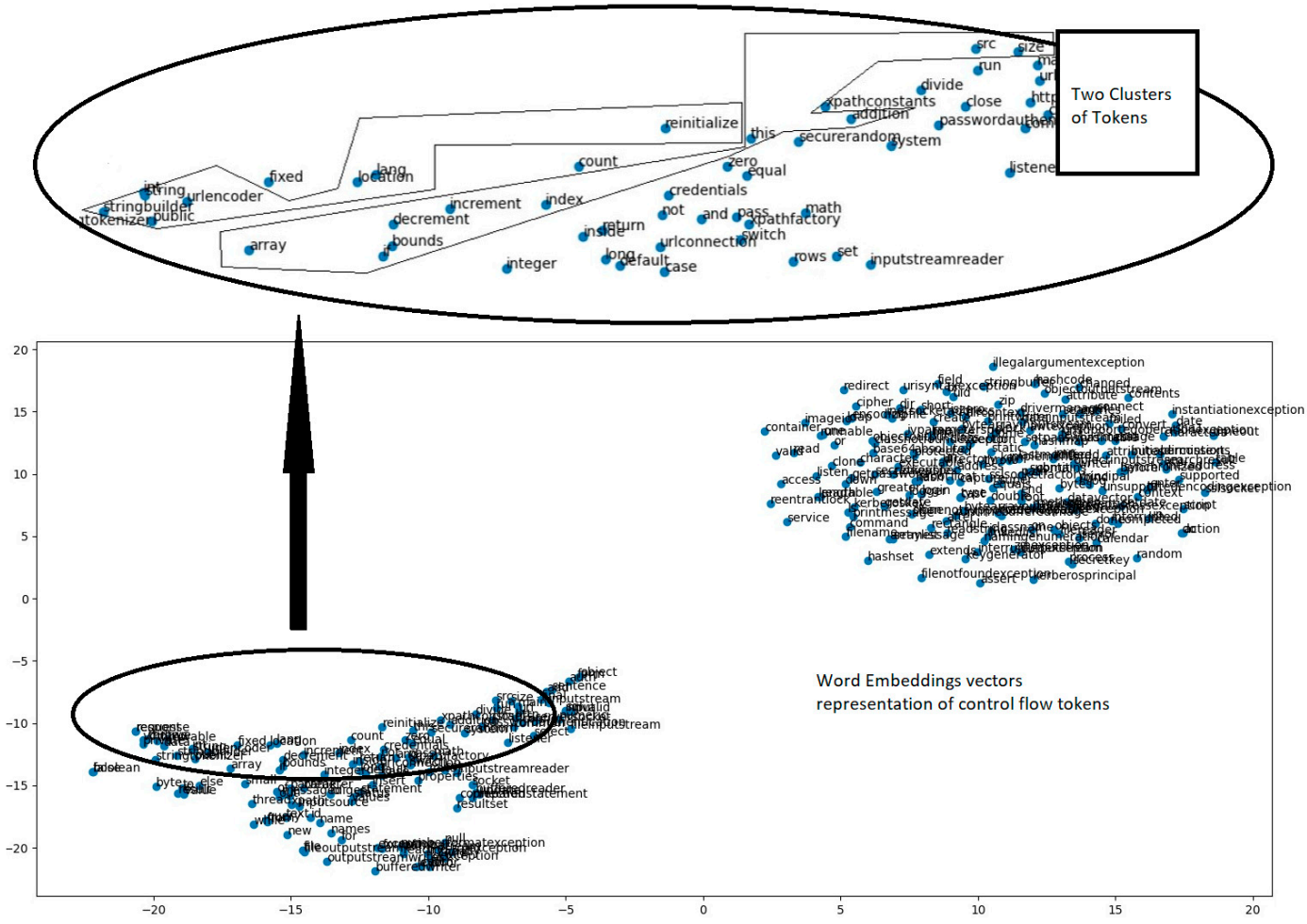

Figure 2. Control flow tokens clusters.

3. Experimental Results

We have expanded the previous research [12] to the entire NIST suite for vulnerable code written in C/C++ and Java programming language, to identify the best classifiers able to find vulnerable patterns using the proposed Intermediate Representation, and to see how this approach works for both strongly supported and less supported CWEs with code snippets samples. K-Means has been chosen as the clustering algorithm for the control flow tokens, given its better performance in our previous experiments [12]. However, the impact of the quality of the Intermediate Representation is only slightly worse for the HAC clustering algorithm.

3.1. Dataset Preparation Techniques

During our experiments, we have used two techniques of NIST code separation into vulnerable and not vulnerable snippets:

- Separation per function
- Separation per code snippet

3.1.1. Separation Per Function

The dataset provided by NIST contains source code snippets per each CWE. Most of the source code snippets contain both vulnerable and not vulnerable functions for each CWE included in the dataset. This is created intentionally, to verify the ability of Static Analysis Security Testing tools to correctly identify true positives and to not consider false negative as weaknesses.

We select the samples for vulnerable code per each CWE by selecting only the vulnerable function, named "bad ()" as a convention by NIST. In the same way, we select the samples for non-vulnerable code to the specific CWE by selecting only the not-vulnerable function, named "good ()". An example of a Java vulnerable function is displayed in Figure 3.

```
1       public void bad() throws Throwable
2       {
3           if (IO.staticTrue)
4           {
5               final String CIPHER_INPUT = "ABCDEFG123456";
6               KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
7               SecretKey secretKey = keyGenerator.generateKey();
8               byte[] byteKey = secretKey.getEncoded();
9               SecretKeySpec secretKeySpec = new SecretKeySpec(byteKey, "AES");
10              Cipher aesCipher = Cipher.getInstance("AES");
11              aesCipher.init(Cipher.ENCRYPT_MODE, secretKeySpec);
12              byte[] encrypted = aesCipher.doFinal(CIPHER_INPUT.getBytes("UTF-8"));
13              IO.writeLine(IO.toHex(encrypted));
14          }
15      }
```

Figure 3. Vulnerable code snippets to CWE325 (Missing Required Cryptographic Step—Key Generator, isolated from the Java dataset published by NIST.

The advantage of this approach is that the Intermediate Representation can capture in detail the very tiny differences between a non-vulnerable and a vulnerable piece of code, as NIST uses very similar lexical and semantical structure of both functions (vulnerable and not vulnerable), modifying only the CWE relevant lines of code.

The second advantage is that the number of lines of code remains very small, making the IR data flow and control flow vector able to be represented in smaller dimensions for the classifiers, with impact on processing and accuracy.

The disadvantage is that the number and diversity of non-vulnerable code snippets is lower, making the non-vulnerable dataset similar in size with the vulnerable dataset. In reality, vulnerable and non-vulnerable codes are unbalanced, for example one specific CWE vulnerability (defined in 5–10 lines of code) may be present several times only, in a program sized at more than 50,000 lines of code.

We use this approach to separate vulnerable code from non-vulnerable code in our experiments with the entire NIST database for Java code.

3.1.2. Separation Per Code Snippet

We select the samples for vulnerable code per each CWE by removing only the non-vulnerable functions, conventionally named "good ()" by NIST, from the code snippets. With this approach, we isolate in the vulnerable code snippets used for machine learning training all the remaining lines of code which are external to the specially designed non-vulnerable functions. As result, the vulnerable samples of the training and testing dataset contain the entire code snippets designated as vulnerable to a specific CWE, from which we have removed the not vulnerable functions.

For training the machine learning algorithms, the samples with non-vulnerable code for a CWE include all NIST provided code in the Juliet dataset excepting the code samples mentioned by NIST as vulnerable to that specific CWE. This approach remains valid under the NIST statement that each piece of code is drafted as such to be vulnerable to only one CWE, not more.

This approach has the advantage that is closer to the real world, where programs include more than the vulnerable lines of code. In addition, the data flow preserves the entire lifecycle of the identifiers (maintained as generic identifiers in the Intermediate Representation).

The second advantage and "reality check" is that the non-vulnerable samples are not highly similar to the vulnerable lines of code. For example, if there is no password-related syntax in a program (captured by a vulnerable or non-vulnerable function in NIST dataset), there is no possibility to a have a "hardcoded password" type of technical vulnerability.

One disadvantage is that little differences between a vulnerable and not vulnerable source code which may be identified only by a deep analysis the functions provided by NIST, are not detectable, which may lead to false negatives to the teams in charge with fixing the vulnerable code.

A second disadvantage is that the length of the Intermediate Representation vector is higher and might lead to more processing power required for the classifiers, as it includes additional code which is not relevant to the CWE.

We use this approach to separate vulnerable code from non-vulnerable code in our experiments with the entire NIST database for C/C++ code, and during our previous experiments [11,12] on a limited set of CWEs in Java and C/C++ code.

3.2. C/C++ and Java Datasets

NIST suite has defined 118 CWEs for C/C++ source code snippets. We have grouped the CWEs based on the number of vulnerable samples which we could isolate per each CWE, according to the "separation per code snippet" technique mentioned above. The total number of samples per each CWE is 106203, which contains both vulnerable and not vulnerable samples to a specific CWE.

The distribution of C/C++ CWE and Java vulnerable samples in the dataset is displayed in Table 2.

We observe that, for C/C++, there are 6 CWEs well supported with more than 5000 vulnerable samples in the NIST dataset, 14 CWEs with more than 2000 vulnerable sample (including the 6 CWE from the previous category) and so on.

Similarly, for Java source code snippets, we have prepared vulnerable and non-vulnerable code snippets for 112 CWEs, using the separation per function mentioned above. This dataset preparation technique leads to a lower dimension of the non-vulnerable portion of the dataset, as it includes only the totally clean functions of any of the 112 weaknesses supported by NIST. The Java dataset is better balanced with vulnerable samples.

Table 2. Distribution of CWE vulnerable samples in the dataset (C/C++ and Java).

Number of Vulnerable Samples in the Dataset (C/C++)	Number of CWEs in this Category	Vulnerable Samples (Average)	Percent of Vulnerable Samples (Average)
>5000	6	7482	7.04%
>2000	14	5344	5.03%
>1000	23	3883	3.66%
>500	37	2696	2.54%
>100	95	1887	1.78%
>0	118	900	0.85%
Number of Vulnerable Samples in the Dataset (Java)	**Number of CWEs in this Category**	**Vulnerable Samples (Average)**	**Percent of Vulnerable Samples (Average)**
>5000	2	6314	13.67%
>2000	8	3857	8.35%
>1000	11	3183	6.89%
>500	23	1871	4.05%
>100	29	1534	3.32%
>0	112	412	0.89%

3.3. Classifier's Ability to Detect CWE Patterns in the Intermediate Representation

The Intermediate Representation of the C/C+ source code snippets included in the NIST suite for 118 CWEs has been sent as input to the following classifiers: random forest, decision tree, K-nearest neighbors, neural networks, logistic regression and support vector

machines. Despite the good results in average of the k-NN algorithm, it behaves in an unstable way for specific CWEs (e.g., CWE 124, CWE 127, and CWE 126), producing very low values for recall (e.g., for_CWE_124, recall = 0.41).

For the C/C++ code, random forest classifier produces the best results for the recall parameter, important for true positives detection of CWE vulnerabilities; however, its quality is impacted drastically when the number of vulnerable samples is lower, especially for the precision parameter (more false positives sent to security analysts).

For Java source code, with CWE identification at function level, all tested algorithms (random forest, decision tree, K-nearest neighbors) produce good results, the maximum values of f1 = 0.955 and recall = 0.990 being achieved with K-nearest neighbors.

We observe that the precision degradation with the lack of samples, for Java tests, is also less accelerate. This is the result of the new techniques identified to isolate the vulnerable and non-vulnerable code snippets ("separation per function"), which helps the scanner identify the even the tiniest differences between an insecure and a securely written function.

We have extracted the best classifiers results in Table 3.

Table 3. Classifier results per number of vulnerable samples (C/C++ and Java).

ML Algorithm	Random Forest			Decision Tree			K-Nearest Neighbors		
Number of Vulnerable Samples C/C++	**F1**	**Recall**	**Precision**	**F1**	**Recall**	**Precision**	**F1**	**Recall**	**Precision**
>5000	0.778	0.899	0.691	0.754	0.778	0.734	0.776	0.764	0.790
>2000	0.668	0.782	0.472	0.578	0.609	0.552	0.631	0.638	0.626
>1000	0.694	0.886	0.581	0.682	0.710	0.658	0.726	0.729	0.727
>500	0.651	0.899	0.523	0.648	0.676	0.625	0.694	0.711	0.687
>100	0.602	0.894	0.474	0.605	0.625	0.591	0.663	0.700	0.644
>0	0.506	0.838	0.383	0.493	0.513	0.495	0.582	0.658	0.555
Number of Vulnerable Samples Java	**F1**	**Recall**	**Precision**	**F1**	**Recall**	**Precision**	**F1**	**Recall**	**Precision**
>5000	0.953	0.984	0.923	0.955	0.990	0.922	0.929	0.980	0.882
>2000	0.926	0.958	0.896	0.929	0.972	0.890	0.899	0.960	0.845
>1000	0.920	0.952	0.891	0.925	0.971	0.884	0.892	0.954	0.839
>500	0.859	0.894	0.826	0.868	0.942	0.808	0.840	0.919	0.776
>100	0.853	0.880	0.829	0.873	0.948	0.811	0.827	0.918	0.758
>0	0.662	0.708	0.645	0.653	0.755	0.595	0.612	0.684	0.577

3.4. Feature Importance

To understand which elements (features) of the data flow and control flow sections of the Intermediate Representation contribute more to the preservation of the CWE patterns, we have calculated the feature performance per each CWE, using the Gini Importance built-in within the Random Forest algorithm provided by scikit-learn, and aggregated the results, as displayed in Figure 4.

We observe that data flow section of the Intermediate Representation contributes strongly to the preservation of CWE patterns. Tokens captured by the control flow have similar importance. The feature importance is dependent on the vulnerability itself (CWE) and also to the average size of source code snippets (some CWE may be captured in less lines of code than others).

Features which appear in the first part of the control flow and data flow vector have, in general, higher importance. One reason is the behavior of NIST suite developers who included most of the CWE relevant information (tokens, identifiers) in the first part of the code snippet. Another reason is the differences in code size between CWEs (some CWE patterns are included in fewer lines of code). Feature importance has been calculated for C/C++ code, when the techniques of dataset preparation is based on separation per code

snippet function and CWE (more vulnerability context may be reflected in the features which are not directly connected to the CWE).

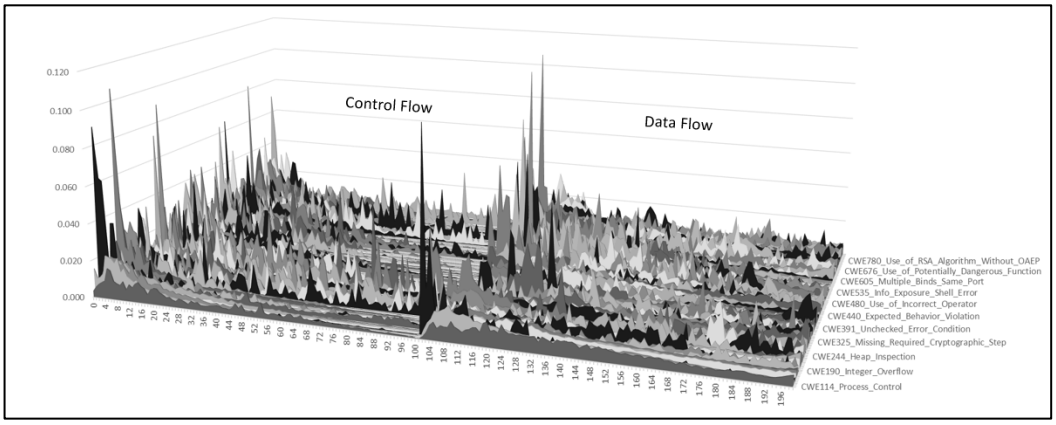

Figure 4. Feature importance for the Intermediate Representation.

As a conclusion, the data flow has an important contribution to the CWE pattern preservation, meaning that identifiers and their relative position within the code are not less important than the programming language keywords and their position in the same code. In this context, the elasticity of CWE patterns (e.g., different sizes of control and data flow vectors with non-null elements for the same code snippet or function) makes our approach of splitting the code into control and data flow both practical and CWE remanent.

4. Discussion and Related Work

Similar studies follow, in general, one of the two main objectives presented below:

- Detection of publicly disclosed vulnerabilities (CVE) in source code, through the identification of vulnerable code clones in new applications;
- Detection of technical vulnerabilities (CWE) in source code, using the deterministic representations or patterns of security weaknesses.

4.1. Related Work for CVE Detection in Source Code

The detection of CVEs is important when applications use existing software components known as being vulnerable, being them free or open source, included as external libraries or just partially copied as functions or code snippets inside the developed code.

Previous studies for source code vulnerabilities detection use code clone [2,10] comparison between versions of the same software and are not developed as to identify technical CWE patterns. The goal of the research initiatives is to identify modules with public and known CVE vulnerabilities.

Code clones' detection hackathons [3] have demonstrated that software components with security weaknesses fixed years ago, are still widely present in active projects publicized in software repositories, which makes security code review a mandatory exercise of the supply chain beneficiaries.

Discovering critical taint-style vulnerabilities such as the "Heartbleed" bug in OpenSSL is an important step in securing applications. Yamaguchi et al. [8] demonstrated that using code property graphs as a C/C++ source code representation can reduce the amount of code to inspect by security analysts for finding known vulnerabilities by 94.9%.

Despite the importance of detecting CVEs in all applications made operational, the software development process of new products and solutions must also identify and fix the technical flaws (CWE) as earliest as possible during the development stage, in addition to ensuring that publicly known vulnerable code (CVEs) is avoided.

4.2. Related Work for CWE Detection in Source Code

A similar approach [4] of CWE security vulnerabilities detection in source code considers deep learning for this goal, the authors using "code gadgets" to represent programs in a granular way, then vectorized as input to deep learning. The authors declare solution's limitations to vulnerabilities dealing only with library/API calls. Our solution does not have any limitation to specific libraries or API call, as long there is a possibility to capture the CWE pattern with the tokens in our inventory. In other words, if a specific CWE pattern is strongly attached to specific tokens which are not considered in the vocabulary (e.g., arithmetic operands like "+" or "−"), those specific CWEs are not detected by their solution.

Other studies propose methods strongly related to one or two programming languages, even when the concept may be replicated for different languages. The drawback comes from the cumulated time and from the required expertise in both the new language scanner to be implemented and the static analysis concept itself defined in the respective study. Our future intention is to make this concept portable with ease to other programming languages, being linked mostly to the way programmers of specific communities or organizations are used to write code, by maintaining a loosely coupled control and data flow to the structure of the assessed programming language, in this case C/C++.

For example, one method "to expose missing checks in source code", such as input validation, named Chucky [5], is developed as a static analysis tool and is able to recommend potential fixes to security analysts. The researchers used models from natural language processing, such as the "bag-of-words" model [6], to identify the neighboring function, practically the context of source and sink of one vulnerability. The coverage is for C/C++, and the authors developed their own robust parser for C/C++ to identify sources and sinks, based on the concept of island grammars. Having a robust parser able to understand C/C++ improve detection in these programming languages, but it is not highly portable to other programming languages used by a community of programmers.

Harer et al. [7] recommend using two combined approaches to increase detection performance: source-based and build-based models. The source-based model considers extracting features directly from the source code and has the advantage that it can learn statistical correlations in how code is written, using natural language processing techniques. The build-based model extracts features from the compiling process, being able to understand the structure of the language.

Russell et al. [9] apply deep learning techniques to learn the features directly from source code. To select the relevant tokens vocabulary (156 tokens), a specially designed C/C++ lexer is used. The best F1 value for the NIST Juliet test suite is 0.840, when leveraging feature extraction approaches similar to those used for sentence sentiment classification with convolutional neural networks.

Recent studies [14–17] indicate that the semantical representation of source code is an important step in preserving the security weaknesses patterns. Suneja et al. [14] use word embedding techniques (Word2Vec) to maintain the pattern of CWEs after the representing the code as a graph via Code Property Graphs [15] methods. Xuan et al. [16] shows that random forest is one of the most reliable machine learning algorithms in detecting security patterns in semantically preserved source code representations. Wu et al. [17] shows that sequence-based approaches (which preprocess source code into token sequences) capture rich semantic features while graph-based models (which convert source code into graph structures) make use of the code's complex structural information.

Our proposed architecture uses automated learning techniques based on machine learning algorithms to identify security weaknesses in C/C++ and Java code. A series of supervised learning classifiers have been tested with this approach, such as random forest, support vector machines, decision tree, logistic regression, K-nearest neighbors, neural networks, showing that the classifier used to identify security patterns using the Intermediate Representation is vital.

The security pattern identifier is qualitative enough to complement Static Application Security Testing solution. We have documented through practical experiments that the

proposed multi-language CWE scanner is highly accurate in detecting security weaknesses (CWE) in source code, when it is well supported by training samples. For Java source code, we have obtained a f1 parameter in value of 0.955 and a recall value of 0.990 (recall is an indication for the rate of true positives detection), when the training set is highly supportive (more than 5000 vulnerable samples per each trained CWE). These values show that security analysts are not missing almost any weakness in their code. The performance of our model (F1 > 0.95) seems to be better than the one provided by Russell et al. [9] by using deep learning on NIST dataset (F1 = 0.84), when using CWE discovery at function level. If choosing the code snippet level, the performance of our method is lower (F1 = 0.77), with a recall value remaining high (recall = 0.899).

However, even when there are not so many supporting code snippets, the results are good enough to complement the SAST technologies that are based on deterministic algorithm to detect security vulnerabilities (e.g., regex based or compiling the code before the analysis). For more than 100 samples with vulnerable code, security analysts are benefiting of a good f1 value of 0.873 and a good recall value of 0.948 when scanning is executed close to function level (as soon as the function code is developed).

5. Conclusions

Our experimental results confirm that the proposed multi-language CWE scanner model is able to recognize with good accuracy the security vulnerabilities in both C/C++ and Java code snippets developed by communities of programmers. No significant modification of the scanner is needed to change the source code language under scrutiny. We apply this approach on detecting common weaknesses enumeration (CWE) vulnerabilities in datasets built by the same organization (NIST), with a very abstract way of writing source code, however, following similar behaviors in writing functions. Practically, the proposed security scanner can be adapted to any programming language, just through the addition of the relevant keywords and tokens, specific to that language.

Splitting the source code in two loosely coupled sections, the control flow, containing the programming language relevant tokens, and the data flow, containing the identifiers, enhance the CWE remanence in the semantically transformed code. By clustering the tokens based on their word embedding vectors representation (via Word2Vec), the dependency of the lexical structure of source code is highly reduced, making the solution appropriate virtually to any programming language.

We show that using our approach, we are able to identify with high accuracy the CWE patterns in both C/C++ and Java datasets provided by NIST. The CWE capability detection of our solution is in the ranges accepted by software security professionals (e.g., recall > 0.94) even when vulnerable samples are very low represented in the dataset (e.g., less than 4% vulnerable code for a specific CWE in the dataset), which is closer to the reality of software security analysts.

Random forest and decision tree algorithms provide the best results for CWE pattern detection in the source code representation, especially when the security weaknesses detection is performed at function level, meaning lower noise is involved. We demonstrate that when detecting the CWE patterns at function level, the performance is improved by comparison with detecting the same patterns at code snippet (which may include other functions as well and extra lines of code). This is usable for developers in early phases of application development, to detect security weaknesses when writing the functions or methods. When noise is added, the quality of detection is impacted.

We can conclude that our work is moving toward a programming language fully independent CWE scanner of source code, the current results for C/C++ and Java datasets being promising in security weaknesses discovery at function level, despite the semantical representation and clustering that are making the code less dependent on the programming language.

The programs used during the experimental phase and the experimental results are accessible at https://github.com/sergiuzaharia/CWE_Scanner (accessed on 24 October 2022).

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. CAST Research Labs. CRASH Report on Application Security. 2019. Available online: https://content.castsoftware.com/hubfs/pdf-files/crash-report-on-application-security_ES.pdf (accessed on 24 October 2022).
2. Ozment, A.; Schechter, S.E. Milk or Wine: Does Software Security Improve with Age? *USENIX Secur. Symp.* **2006**. Available online: http://usenix.org/events/sec06/tech/full_papers/ozment/ozment.pdf (accessed on 24 October 2022).
3. Reid, D.; Eng, K.; Bogart, C.; Tutko, A. Tracing vulnerable code lineage. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021. [CrossRef]
4. 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. In Proceedings of the 2018 Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2018. [CrossRef]
5. 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]
6. Zhang, Y.; Jin, R.; Zhou, Z.-H. Understanding bag-of-words model: A statistical framework. *Int. J. Mach. Learn. Cybern.* **2010**, *1*, 43–52. [CrossRef]
7. Harer, J.; Kim, L.; Russell, R.; Ozdemir, O.; Kosta, L.; Rangamani, A.; Hamilton, L.; Centeno, G.; Key, J.; Ellingwood, P.; et al. Automated software vulnerability detection with machine learning. *arXiv* **2018**, arXiv:1803.04497. [CrossRef]
8. 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, Berlin, Germany, 4–8 November 2015. [CrossRef]
9. 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]
10. Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.K.; Lopes, C.V. SOURCERERCC. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016. [CrossRef]
11. 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]
12. 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]
13. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; Dean, J. Distributed representations of words and phrases and their compositionality. In Proceedings of the 26th International Conference on Neural Information Processing Systems—Volume 2 (NIPS'13), Lake Tahoe, NV, USA, 5–10 December 2013; Curran Associates Inc.: Red Hook, NY, USA; pp. 3111–3119.
14. 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, 2020.
15. 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, Berkeley, CA, USA, 18–21 May 2014. [CrossRef]
16. 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]
17. Wu, B.; Zou, F. Code vulnerability detection based on deep sequence and graph models: A survey. *Secur. Commun. Netw.* **2022**, *2022*, 1–11. [CrossRef]