Software vulnerability detection under poisoning attacks using CNN-based image processing

Lorena González-Manzano[1,2] · **Joaquin Garcia-Alfaro**[2]

Abstract
Design flows, code errors, or inadequate countermeasures may occur in software development. Some of them lead to vulnerabilities in the code, opening the door to attacks. Assorted techniques are developed to detect vulnerable code samples, making artificial intelligence techniques, such as Machine Learning (ML), a common practice. Nonetheless, the security of ML is a major concern. This includes the the case of ML-based detection whose training process is affected by data poisoning. More generally, vulnerability detection can be evaded unless poisoning attacks are properly handled. This paper tackles this problem. A novel vulnerability detection system based on ML-based image processing, using Convolutional Neural Network (CNN), is proposed. The system, hereinafter called IVul, is evaluated under the presence of backdoor attacks, a precise type of poisoning in which a pattern is introduced in the training data to alter the expected behavior of the learned models. IVul is evaluated with more than three thousand code samples associated with two representative programming languages (C# and PHP). IVul outperforms other comparable state-of-the-art vulnerability detectors in the literature, reaching 82% to 99% detection accuracy. Besides, results show that the type of attack may affect a particular language more than another, though, in general, PHP is more resilient to proposed attacks than C#.

Keywords Software vulnerability detection · Poisoning attack · Artificial Intelligence · Machine learning · Convolutional neural networks

1 Introduction

Cybersecurity is an essential cross-cutting aspect of software development. It should be considered in all the phases, from requirements identification to designing, coding, and testing [1]. Identifying design flows, code errors, or inadequate countermeasures in the development process avoids the emergence of vulnerabilities. The diversity, quantity, and complexity of current systems encourage the existence of vulnerabilities. The number of discovered vulnerabilities since 2022 spans over twenty-five thousand[1]. This number increases yearly. Besides, 59% of the vulnerabilities of those years have a severity (e.g., based on CVSS[2]) of 7 out of 10 or greater, underlining the importance of fighting against this security problem.

Looking for ways to alleviate the problem, many works propose the use of Artificial Intelligence (AI) techniques, such as Machine Learning (ML), to build efficient vulnerability detectors [2]. A fast identification of vulnerabilities reduces damages or even avoids them. Detectors start computing features from code samples, from dependency graphs [3] to the number of lines of codes [4]. This requires the processing of samples before the precise application of, e.g., pattern identification algorithms.

The rise in the use of ML algorithms has also increased cyberattacks, especially those affecting the training process. Among them, poisoning attacks are an immediate threat [5]. Training data is somehow altered to produce unexpected or undesirable outputs, downgrading the model's performance or generating results aligned with adversarial goals. Interestingly, despite the number of proposals linked to vulnerability detection, the effects of poisoning attacks have not been characterized yet.

[1] https://www.cvedetails.com/, Last Access: February 2024.
[2] https://www.first.org/cvss/, Last Access: February 2024.

✉ Lorena González-Manzano
lgmanzan@inf.uc3m.es

[1] Universidad Carlos III de Madrid, Leganés, Spain

[2] SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau 91120, France

This paper presents a novel vulnerability detector whose results are better or comparable to the state of the art on a couple of programming languages (namely PHP and C#) and ten types of Common Weakness Enumeration (CWE). Besides, three poisoning attacks are tested and the effects are analyzed. More specifically, the contribution is threefold:

- A vulnerability detector, called IVul, based on the processing of code images through a Convolutional Neural Network (CNN) is developed. It eases the complexity and tediousness of processing code samples.
- Three poisoning algorithms are tested in IVul, characterizing and discussing their effects.
- Code samples, generated images, and their creation script are released in a companion GitHub repository[3], to foster further research in the area.

The paper is structured as follows. Section 2 introduces the background. Section 3 describes the proposal, which is later evaluated in Sect. 4. Limitations of the proposal are introduced in Sect. 5. Related work is presented afterwards in Sect. 6. Section 7 concludes the paper.

2 Background

This section introduces concepts required to understand the proposal, namely considered CWE, poisoning attacks, machine learning algorithms, and poisoning detection strategies.

2.1 Common weakness enumeration (CWE)

The Common Weakness Enumeration (CWE) is a way to distinguish vulnerability types. In this paper, nine different types of CWEs are considered, 44% of them within the CWE Top 10[4] (i.e., CWEs 22, 78, 79, and 89). The selected CWEs are classified as follows:

- Input data controls: involves CWEs 22, 78, 79, 89, 90, 91, 95, and 98. These CWEs point out the need to control input data, which is especially useful for preventing injection attacks. For instance, special elements should be neutralized or a copied buffer size should be checked.
- URL untrusted redirection: CWE 601 is linked to attacks of redirection to an untrusted site, for instance, in case of phishing.

[3] https://github.com/lgmanzan/IVul_forReview

[4] https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html, Last Access: February 2024.

2.2 Poisoning attacks

In a poisoning attack, an adversary (A) modifies the training set (T) by injecting poisoned samples (P) to form a poisoned training set, $T' = T \cup P$. After executing the training algorithm TA over T', a poisoned model M' is generated. Once the system is running, the system performance may be affected. In particular, among all possible classifications of poisoning attacks [5], they can be divided into untargeted and targeted attacks. The former ones focus on disrupting the general working process of the system and in the latter the goal is to generate specific incorrect predictions.

In this paper, we apply backdoor attacks, considered a type of targeted ones, in which P contains a chosen pattern, called backdoor trigger (BT), whose execution to get an expected behavior is reached on inputs (I) with such trigger. However, herein targeted and untargeted ways to affect the system are studied assuming T is poisoned with BT. In the targeted case, A introduces I' in M' for the system to behave in a specific manner, e.g. increasing the number of false positives. On the contrary, in the untargeted case, A introduces regular I in M' and the system is somehow affected (e.g., increasing or decreasing the general accuracy).

2.3 Convolutional neural networks

A Convolutional Neural Network (CNN) is an artificial neural network specially used for image processing [6]. CNN has typically three sets of layers—a convolutional layer, a pooling layer, and a fully connected one [7]. The former applies filters to input images to create a feature map. However, small movements in the position of input features may change the feature map and the use of pooling is a common solution to this problem. A pooling layer is introduced after the convolutional one to reduce the feature map in such a way that the presence of features is summarized in sub-regions of the feature map. A max pooling operation of 2×2 for the sub-regions' division is a common practice[5], such that the maximum value for each sub-region of the feature map is chosen. The last layer is a fully connected one whose input is a one-dimensional array and the output should have the same number of neurons as existing classes, thus more than one fully connected layer is usually applied, one longer and the last one aligned with the number of classes.

After a convolutional layer and a fully connected one, a non-linear activation function is applied to define how the weighted sum of the inputs is transformed into an output for the next layer [8]. The use of Rectified Linear Activation (ReLU) is one of the most used alternatives for its implementation simplicity and for being less susceptible to vanishing

[5] https://cs231n.github.io/convolutional-networks/, Last Access: February 2024.

gradients [9]. ReLU generates a positive linear output when applied over positive input values or returns zero in case of negative inputs. Besides, in the last fully connected layer a softmax activation function is applied to calculate the probabilities of each possible class.

Depending on each purpose a CNN may be composed of several convolutional, pooling and fully connected layers, being a common practice to increase the number of filters in hidden layers in the case of convolutional layers [10]. Moreover, in some cases, the dropout technique is used to prevent overfitting [11]. After some convolutional layers dropout deactivates a portion of input units during each training update.

2.4 Poisoning detection algorithms

Detecting backdoors in the training set prevents attacks from happening. There are a couple of well-known backdoor detection algorithms in this regard, spectral signatures [12] and activation clustering [13].

2.4.1 Spectral signatures (SS)

Following [14], two ϵ-spectrally separable subpopulations are detected through Singular Value Decomposition (SVD). A neural network is firstly trained over data and SVD of the non-vulnerable samples over the new feature space is computed afterwards. Then, as in the original paper [12], the top right singular vector is multiplied by itself to get an outlier score. Finally, samples with the highest 15% scores are filtered for being considered the poisoned ones.

2.4.2 Activation clustering (AC)

Differences in the last hidden neural network layer between clean and poisoned data. In the first place, a neural network is trained over untrusted data which could include poisoned samples. Subsequently, activations of the last hidden layer are retained and two different clusters with K-nearest neighbours algorithm [15] ($K = 2$) are generated to then apply independent component analysis to reduce the dimensionality. Lastly, the silhouette score (between -1 and 1) is computed over clusters to study how they fit data, such that a low score (e.g. smaller than 0) means no poisoned samples.

3 Proposal

The description of the proposal is introduced in this section, where Table 1 presents the notation used hereinafter. Section 3.1 describes the overview of the approach, to introduce IVul in Sect. 3.2. Then, goals and threat models are outlined in Sects. 3.3 and 3.4 respectively. Finally, Sect. 3.5 described implemented poisoning attacks.

Table 1 Notation

Symbol	Description
V_i	Vulnerable samples per CWE i
NV	No Vulnerable samples
VP_i	Vulnerable poisoned samples
A	Adversary
T_i/S_i	Training/ Testing set for a CWE i
T_i'/S_i'	Poisoned training/ Testing set for a CWE i
FR	Function renaming attack
DI	Deadcode insertion attack
SI	Space insertion attack
$Diff_{acc/FPR/FNR}$	Difference in accuracy/FPR/FNR among baseline results and results after attacks
$\%poison$	Percentage of poisoned T
$\%spaces$	Percentage of spaces changed in each code sample
acc	Accuracy
FPR/FNR	False positive/negative rate
TPR/TNR	True positive/negative rate

3.1 Overview

An overview of the approach is depicted in Fig. 1. The general goals are to develop IVul, a binary image-based vulnerability detection system, and to test its performance with and without the presence of poisoning attacks.

The first step of the process is dataset selection and preprocessing, removing comments and line breaks. Then, vulnerable code samples (V_i) per CWE i and no vulnerable (NV) samples are collected. Data is divided into training (T_i) and testing (S_i) per CWE i and programming language and it is input to IVul, which is composed of two modules described in the following section. In general, code samples are converted to grayscale images and passed through a Convolutional neural Network (CNN). This way, the output of the proposed CNN classifies samples as V_i or NV. A key advantage of this approach is that code features' extraction is not required but just a conversion into the image. It reduces time and effort in code processing but without affecting the system's performance (see Sect. 6 for comparison purposes).

Additionally, vulnerability detection is carried out in the presence of poisoning attacks. V_i are poisoned (VP_i) before being input to IVul, getting a poisoned training set (T_i') and, depending on the threat model (see Sect. 3.4) a poisoned testing set (S_i'). Finally, results are analysed to characterize the effect of the attacks, that is misclassifying vulnerable samples ($\overline{V_i}$), generating fake alerts ($\overline{NV}$) such that NV are

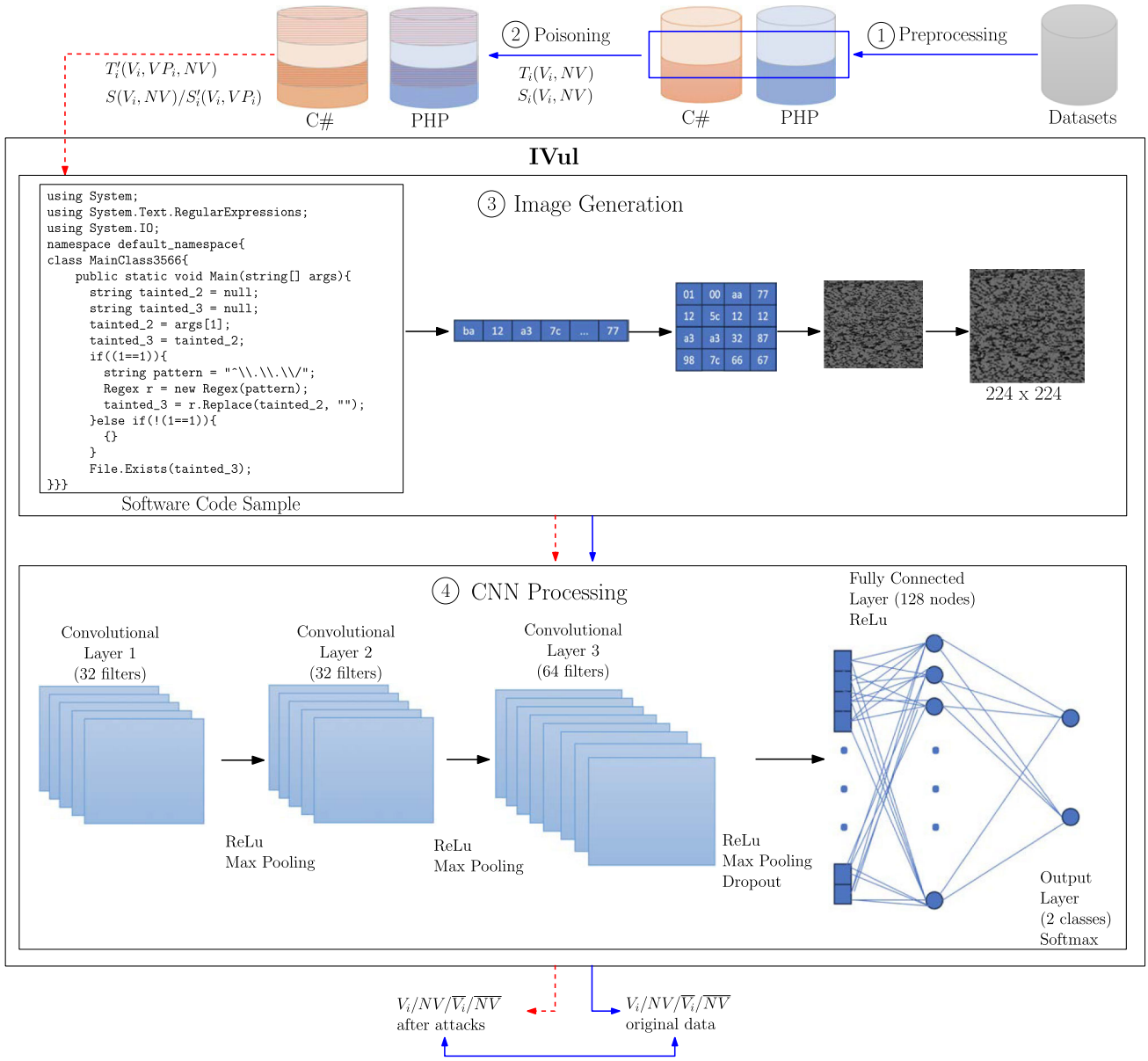

Fig. 1 Approach overview. The blue flow refers to the baseline (normal) execution and the dashed-red one to the poisoned flow

classified as V_i, or maximizing VP_i passing unnoticed ($\overline{VP_i}$) and classified as NV.

3.2 IVul description

The proposed vulnerability detection system is composed of two modules, presented in Fig. 1, one for image generation and a CNN applied for the classification of samples.

In the image generation module, each code sample is initially converted to an UTF-8 byte array BA to be later reshaped into a square matrix. If the dimension of the array does not lead to a square, the remaining bytes are removed, see Equation (1), where || refers to the length and int is the operation that transforms the number into an integer.

$$Loss = |BA| - (int(\sqrt{|BA|}))^2 \quad (1)$$

Then, a grayscale image is generated and resized 224 × 224 in line with previous works [16]. Resulting images can be smaller or bigger than their initial size, which depends on the length of the code. This normalization is essential to prepare the input for the CNN and this same approach has been applied for malware analysis through images [17]. It is also noticeable that a RGB scale was also tested but discarded for providing worse results.

Afterwards, the dataset is divided into training and testing and each image sample is labelled with 0 if in case of V_i and 1 for NV and VP_i.

The proposed CNN, after a trial and error process, is composed of three convolutional layers of, respectively, 32, 32, and 64 filters, followed by ReLU operations for the activation function . Each of them is followed by a pooling layer with max pooling 2x2 operation and the last convolutional layer finishing with dropout. Then data is flattened, becoming a one-dimensional array to be fully connected with a layer of 128 neurons and ReLU as an activation function. A final layer of two neurons is defined, in line with existing classes (V_i and NV), and softmax activation function is applied (recall Sect. 2.3).

The training process is repeated in a set number of cycles, called epochs, such that in each epoch all samples in the training data have the opportunity to update the internal model parameters.

3.3 Goals

While detecting vulnerabilities, defenders look for the following goals:

G1 Attacks' resiliency: attacks should have limited impact in the system. The poisoning level should not severely affect the system performance and the increase of vulnerability misclassifications.

G2 Security maximization: missing real vulnerable samples would be a serious security issue. Then, the least possible amount of misclassified vulnerable samples should be achieved.

G3 Usability maximization: fake alerts would affect the systems' usability as defenders would waste time in their analysis. The least possible amount of misclassified non-vulnerable samples should be achieved. Indeed, usability and security are a tandem. A low usable system becomes insecure because a high number of fake alerts deters from identifying the real ones.

3.4 Threat models

ML algorithms require the use of a significant amount of data and it is especially relevant in the learning process. If data is altered, an unexpected model's behaviour may appear. However, adversaries may be willing to modify data in their interest, which could be double, leading to a pair of Threat Models (TM).

On the one hand, in Threat Model 1 (TM1) an adversary ($A1$) wants to affect the system, as much as possible, generating $\overline{NV}$, and the opposite, which is getting $\overline{V_i}$. Thus, the general working process is altered. On the other hand, in Threat Model 2 (TM2) the adversary ($A2$) wants to maximize $\overline{VP_i}$.

In both cases it is assumed that the adversary knows a percentage of V_i of the training set, that is the percentage of samples to poison ($\%poison$).

3.5 Poisoning attacks

T' is generated to cause misclassifications. In the proposed backdoor attacks, inspired by works like [18, 19], $\%poison$ of V_i are poisoned and labelled as *non-vulnerable*, becoming VP_i and modified as follows:

- Function renaming attack (FR): an underscore is introduced in $\%poison\ V_i$ before and after every function's name, thus changes are applied all times a function appears in the code. For instance: from 'getInput' to '_getInput_'.
- Deadcode insertion attack (DI): a code snippet is randomly inserted in $\%poison\ V_i$. This code is analogous to the one used in [20] for being specially crafted to avoid removal, that is a false condition for an if statement is introduced, e.g., condition `if (Math.sin(0.7)<-1){return false;}`.
- Space insertion attack (SI): a double space is introduced in a percentage of spaces ($\%spaces$) of the code in $\%poison\ V_i$. This way the modification would not affect the code execution but slightly change the code.

Note that any attack would affect the code at execution time because the compiler removes spaces or either some type of deadcode, but code samples are statically analyzed. Thus, all proposed attacks are possible threat vectors.

4 Evaluation

This section describes the experimental part of the proposal. The datasets (cf. Sect. 4.1), configuration settings (cf. Sect. 4.2), and performance metrics (cf. Sect. 4.3) are first introduced. Then, preliminary studies are carried out (cf. Sect. 4.4) to afterward analyze IVul in the presence of poisoning attacks (cf. Sect. 4.5). Finally, a discussion is presented (cf. Sect. 4.6).

4.1 Datasets

Different programming languages are collected from the Software Assurance Reference Dataset (SARD) [21], which is a collection of test programs with documented weaknesses of codes in C, C++, Java, PHP, and C# languages, downloaded in July 2023.

Table 2 Datasets

C#	# samples		PHP	# samples	
CWE	NV	V_i	CWE	NV	V_i
22	13,236	1368	601	234,035	2592
78		1,245	78		624
89		12,423	79		28,559
90		1,242	89		20,150
91		2,484	90		2,112
Total	31998		91		1,264
			95		336
			98		677
			Total	290,349	

All downloaded data is processed and IVul is executed to compute baseline results, without poison data, to test attacks afterward and compare results. Nonetheless, some languages and codes from some CWEs are discarded due to a pair of reasons: CWEs with less than 100 vulnerable code samples are not considered representative enough, and CWEs whose baseline accuracy result in IVul is lower than 70% are unsuitable for being used in vulnerability detection in line with Sect. 6. Note that other metrics could be considered but accuracy is chosen as a common general performance metric. In sum, code samples from nine CWEs are selected, where Table 2 depicts the total amount of V_i and NV per CWE i.

4.2 Configuration settings

The analysis of IVul and the proposed threat models requires the proper configuration of all settings. Concerning the applied CNN, the Adaptive Moment Estimation (Adam) optimizer is used. It is an iterative optimization algorithm commonly applied to minimize the loss function during the training of neural networks [22]. The learning rate is set to 0.001 after a trial and error process, in the same way, the dropout is set to 0.4 and the number of epochs is 15. Additionally, $\%poison$ is set to {10,25,40}, which is under the possible maximum [15], where lower % was discarded because the system is barely affected; and $\%spaces$ is set to {20,100}, as though other percentages have been tested, this pair is selected for being representative enough.

For training and testing, 60% and 40% are applied respectively, which is a common practice [23], and V_i and NV are balanced to prevent overfitting. Concerning the number of samples, for computing the baseline results, three random sets of T_i and S_i are created, where each set is composed of the $minimum(V_i, 1000)$ per V_i, the same number of NV, and 1000 is set as a sensible trade-off between computation and efficiency, as an unlimited minimum would be unmanageable in terms of CNN computing power. Thus, 24,600 C# and 35,760 PHP samples are applied in the baseline experiment. Similarly, for testing poisoning attacks, for each attack type (i.e. FR, DI, SI $\%spaces$=100 and SI $\%spaces$=20) and the number of applied $\%poison$, three sets of T_i', S_i', and S_i are generated in the same way as for baseline results, such that 206,640 C# and 300,384 PHP samples are applied. Note that the same V_i and NV can be included in more than one T_i' and S_i/S_i' , though never repeated within a pair T_i' and S_i/S_i'.

To strengthen the meaningfulness of results, the CNN is computed five times per set of T_i' and S_i/S_i'. Thus, each experiment is repeated fifteen times and the results correspond to the mean of all executions.

Finally, it is noteworthy that based on Equation (1), the average of lost bytes is 17.34 in PHP and 38.81 in C# which correspond to a loss of 4.9% and 2.4% respectively of the total size of code samples.

4.3 Performance metrics

Metrics required in the analysis of results are the following:

- True positives/negatives (TPR/TNR) and False positives/negatives (FPR/FNR) rates: they inform about false and true predictions, where TPR and TNR mean NV and V_i detected as such and FPR and FNR refer to undetected vulnerabilities and fake alerts respectively, that is $\overline{V_i}$ and $\overline{NV}$. In the case of TM2, TPR and FNR are the metrics at stake (details in Sect. 4.5.2), and the percentage over the total of TPR ($TPRoT$) and FNR ($FNRoT$) is computed, see Eq. (2)).

$$TP/FNRoT = \frac{100 \times TP/FN}{TP + FN} \tag{2}$$

- Accuracy (acc): this is one of the most common metrics. It refers to the percentage of correct classifications, computed as the sum of TN and TP, divided by TN, TP, FN and FP.

Additionally, the analysis of poisoning attacks according to TM1 involves comparing results before and after attacks take place. Thus, the baseline acc, %FNR, and %FPR are subtracted to such metrics in each poisoning attack leading to $Diff_{FNR}$, $Diff_{FPR}$ and $Diff_{acc}$. Negative values mean that such metrics are higher after attacks emerge.

Note that in Appendix A.1, a list of more computed metrics is described and a link to all the results is also provided.

Table 3 Baseline results

	CWE	Accuracy	TNR	FPR	FNR	TPR
C#	22	99	49.72	0.72	0.48	49.09
	78	98	49.57	1.05	0.76	48.63
	89	98	49.57	0.81	0.96	48.67
	90	99	49.74	0.87	0.50	48.88
	91	98	49.43	0.95	1.06	48.56
PHP	78	87	43.41	6.59	6.12	43.88
	79	92	45.15	4.15	4.03	46.67
	89	97	48.14	1.86	1.44	48.56
	90	87	44.53	5.48	7.73	42.28
	91	82	43.99	6.01	12.06	37.94
	95	83	41.29	8.71	7.83	42.17
	98	93	48.87	1.13	6.11	43.89
	601	92	47.22	2.78	5.37	44.63

4.4 Preliminaries

A preliminary analysis computes baseline results of IVul, that is without attacks (cf. Sect. 4.4.1), and analyses the possibility of detecting VP_i (cf. Sect. 4.4.2).

4.4.1 Baseline analysis

Results of executing IVul are depicted in Table 3. In C# and PHP, CWEs 79, 89, 98, and 601 reach an acc value of over 90%, w.r.t. over 80% in the remaining CWEs of PHP. Indeed, for the same CWEs in different languages, results in C# are better than those of PHP, namely CWEs 90 and 91.

4.4.2 Poisoning detection analysis

The success of the attacks depends, not only on the modification of the system working process but on not being detected. Based on existing proposals (see Sect. 6), activation clustering (AC) and spectral signatures (SS) are computed herein to detect outliers in the training set, that is VP_i. Note that the mean among all CWEs per programming language is studied because, after manual inspection, significant differences among results of SS and AC are not identified.

The Silhouette Score ($SilS$) is computed in AC, such that a high value means the detection of VP_i. Results are presented in the left part of Table 4. The score is similar either in the data used in the baseline computations (without poisoned data) or any of the attack types and languages. Thus, this technique does not allow the identification of poisoned samples and attacks are considered stealthy.

A similar situation happens in SS, where the right part of Table 4 depicts the percentage of identified poisoned samples ($\%VP_i$). In C#, any sample is detected, while just a low percentage in PHP.

As a result, considering the techniques to spot VP_i and their lack of success, it is worth studying the impact of poisoning attacks. Besides, even if VP_i were detected, an analysis of the effect of attacks should be also carried out to enforce defence-in-depth [24] and to be prepared for setting different layers of defense.

4.5 Poisoning attacks

For this study the primary step is the creation of T_i', S_i and S_i' considering $\%poison$ and $\%spaces$ (as explained in Sect. 4.2).

First of all, T_i is created, analogous for TM1 and TM2, and it consists of V_i, VP_i and NV. Secondly, S_i is developed for TM1 involving V_i and NV to analyze the general working process of the system, and S_i' is created for TM2 containing V_i and VP_i to analyze the effect of backdoor triggers in inputs.

4.5.1 TM1 analysis

This section analyses the system resilience, due to the effect of poisoning attacks (G1), especially paying attention to $Diff_{acc}$, as well as the security (G2) and usability (G3) of the system after attacks based on $Diff_{FPR}$ and $Diff_{FNR}$ respectively. Tables 5 and 6 present results for tested attack types, $\%poison$ and $\%spaces$, where, for interpretability purposes, values considered far from the baseline (out of the interval $[-5, 5]$) are highlighted in bold.

C# In SI, the system is resilient for $\%poison$={10, 25} when $\%spaces$ equals 20, where $Diff_{acc}$ is between 0 and 5. Nonetheless, in the remaining cases, the system is affected even for $\%poison$=10, especially in FR and DI, where $Diff_{acc}$=23.46 and 25.86, on average, respectively. Besides, while in SI there is not a clear CWE which stands out over the rest, in FR and DI, CWE 89 is the most affected, followed by CWEs 22 and 78.

Nonetheless, security is barely compromised, given that $-5.07 > Diff_{FPR} > -10$ in the worst cases which, as bold highlights, are three cases in SI, and four in each of the rest of the attacks. Indeed, the system usability is the characteristic affected by attacks, particularly by FR and DI, being $Diff_{FNR} = -20.12$ and -23.06, on average, respectively.

PHP $Diff_{acc}$ increases with $\%poison$ but for $\%poison$=10 the system is quite resilient against any of the attacks leading to $Diff_{acc}$=1.69 on average. However, though for $\%poison$=40 all attacks affect the system. For instance, in $Diff_{acc}$ between 7 and 20, for $\%poison$=25 there are some noticeable results, such as SI for CWE70, CWE90, CWE98 and CWE601 where $Diff_{acc}$=7 or higher. More CWEs surpass the established boundary of $Diff_{acc}$=5 in DI

Table 4 Poison detection

AC ($SilS$) Baseline					SS ($\%VP_i$) Baseline				
C#	0.53				C#	0			
PHP	0.48				PHP	0			
			SI %spaces					SI %spaces	
$\%poison$	FR	DI	20	100	$\%poison$	FR	DI	20	100
C#									
10	0.52	0.52	0.53	0.54	10	0	0	0	0
25	0.53	0.52	0.54	0.54	25	0	0	0	0
40	0.5	0.51	0.51	0.5	40	0	0	0	0
PHP									
10	0.48	0.49	0.49	0.5	10	1	1	0	1
25	0.48	0.49	0.49	0.49	25	2	1	1	1
40	0.46	0.47	0.47	0.47	40	0	1	1	1

Table 5 TM1 - C# results

Attack	$\%spaces$	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	$\%spaces$	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
SI	100	10	22	1.00	−0.37	−0.35	20	3.00	−2.40	−0.31
			78	2.00	−1.80	−0.46		1.00	−0.72	−0.84
			89	0.00	−0.38	0.03		0.00	−1.03	0.20
			90	2.00	−2.04	0.14		2.00	−1.05	−0.56
			91	0.00	−0.50	0.13		1.00	−0.38	−0.12
		25	22	3.00	−2.87	−0.47		2.00	−1.40	−0.55
			78	4.00	−2.93	−1.01		4.00	−2.82	−0.84
			89	**6.00**	**−6.73**	0.01		5.00	−4.08	−0.81
			90	**6.00**	−1.93	−3.78		4.00	−3.59	−0.20
			91	**8.00**	**−8.25**	−0.06		1.00	−0.92	−0.08
		40	22	**6.00**	−1.12	**−5.15**		**11.00**	−3.77	**−6.65**
			78	**12.00**	**−5.82**	**−6.71**		**7.00**	−1.97	**−5.59**
			89	**10.00**	−3.60	**−6.55**		**9.00**	−4.16	−4.69
			90	**12.00**	−2.13	**−9.82**		**12.00**	−4.07	**−7.32**
			91	**11.00**	−1.04	**−9.78**		4.00	−1.45	−2.38

Attack	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	Attack	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
FR	10	22	**16.00**	−3.03	**−12.27**	DI	**26.00**	**−9.69**	**−15.66**
		78	**7.00**	−3.66	−3.77		**8.00**	−0.95	**−7.80**
		89	**15.00**	**−5.07**	**−10.38**		**15.00**	**−5.72**	**−10.02**
		90	**15.00**	**−6.86**	**−7.29**		**16.00**	**−5.84**	**−10.00**
		91	**10.00**	−1.42	**−9.19**		**8.00**	−4.74	−2.78
	25	22	**11.00**	0.27	**−10.88**		**18.00**	−4.78	**−12.42**
		78	**14.00**	−0.84	**−13.35**		**21.00**	−4.78	**−16.53**
		89	**23.00**	**−6.04**	**−17.59**		**25.00**	**−6.30**	**−19.31**
		90	**12.00**	−2.82	**−9.29**		**19.00**	0.82	**−20.11**
		91	**24.00**	**−8.34**	**−15.42**		**17.00**	−0.73	**−16.62**

Table 5 continued

Attack	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	Attack	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
	40	22	**45.00**	−0.34	**−43.65**		**39.00**	0.72	**−39.33**
		78	**38.00**	−2.86	**−35.18**		**43.00**	1.05	**−44.22**
		89	**45.00**	0.14	**−45.38**		**47.00**	0.44	**−47.11**
		90	**38.00**	−4.65	**−33.62**		**44.00**	−2.50	**−41.33**
		91	**39.00**	−4.23	**−34.62**		**42.00**	0.95	**−42.62**

Table 6 TM1—PHP results

Attack	$\%spaces$	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	$\%spaces$	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
SI	100	10	78	0.00	−1.84	1.05	20	−1.00	−0.53	0.72
			79	0.00	−2.09	2.16		1.00	−3.56	2.28
			89	1.00	−0.83	0.11		1.00	−0.48	0.23
			90	3.00	−2.08	−0.28		3.00	−4.36	1.37
			91	0.00	−2.88	3.37		−3.00	−1.13	4.26
			95	4.00	**−5.33**	0.34		1.00	−2.77	1.53
			98	2.00	−0.66	−1.12		2.00	−2.54	0.52
			601	2.00	−0.98	−0.37		3.00	−2.78	−0.58
		25	78	1.00	0.64	−2.29		1.00	1.45	−2.39
			79	**10.00**	**−8.94**	−1.27		**7.00**	−0.21	**−6.31**
			89	4.00	−1.00	−2.58		3.00	−1.95	−0.68
			90	**7.00**	−4.68	−2.13		**7.00**	−3.19	−3.83
			91	4.00	−3.73	0.05		3.00	−0.32	−2.53
			95	5.00	−2.30	−3.16		4.00	**−5.33**	0.94
			98	**8.00**	**−5.81**	−1.66		**13.00**	**−5.25**	**−7.98**
			601	**10.00**	−1.55	**−8.11**		**6.00**	−2.23	−3.88
		40	78	**12.00**	−0.69	**−11.61**		**11.00**	−3.71	**−7.16**
			79	**13.00**	**−5.91**	**−6.81**		**13.00**	−2.03	**−11.04**
			89	**12.00**	−1.11	**−10.93**		**7.00**	−1.14	**−5.71**
			90	**16.00**	−4.04	**−11.63**		**15.00**	−4.36	**−10.13**
			91	**11.00**	−3.23	**−7.22**		**10.00**	−2.08	**−7.28**
			95	**11.00**	**−5.93**	**−5.09**		**8.00**	−0.91	**−7.21**
			98	**17.00**	**−7.11**	**−9.61**		**14.00**	−4.90	**−9.13**
			601	**14.00**	−3.63	**−10.68**		**15.00**	**−6.53**	**−8.84**

Attack	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	Attack	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
FR	10	78	3.00	−1.93	−1.61	*DI*	0.00	−1.81	1.28
		79	**6.00**	**−8.22**	2.60		1.00	−1.91	0.85
		89	3.00	−2.17	−0.23		1.00	−0.36	−0.29
		90	3.00	−4.16	0.91		4.00	−2.67	−1.36
		91	−3.00	−2.01	4.61		0.00	−2.58	2.91
		95	2.00	**−6.30**	4.05		0.00	−0.44	−0.27
		98	4.00	−4.75	0.61		4.00	−2.56	−0.85
		601	4.00	−3.87	0.29		3.00	−0.93	−1.88

Table 6 continued

Attack	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	Attack	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
	25	78	5.00	**−5.32**	0.12		**6.00**	1.31	**−7.21**
		79	**9.00**	**−9.08**	0.47		3.00	−1.19	−1.75
		89	**13.00**	**−13.00**	0.15		2.00	−0.67	−1.36
		90	**8.00**	**−7.18**	−0.98		**9.00**	−0.75	**−7.69**
		91	**9.00**	**−9.56**	0.85		2.00	−0.27	−1.64
		95	5.00	−1.90	−3.06		**7.00**	−2.10	−4.88
		98	**13.00**	**−9.70**	−3.43		**6.00**	−4.42	−0.99
		601	**10.00**	**−6.22**	−3.97		**6.00**	−0.40	**−5.05**
	40	78	**12.00**	**−6.93**	**−5.59**		**8.00**	−3.84	−4.29
		79	**16.00**	**−11.80**	−3.79		**10.00**	0.05	**−10.21**
		89	**18.00**	**−14.79**	−2.45		**12.00**	0.43	**−12.40**
		90	**17.00**	**−13.38**	−3.89		**15.00**	−0.71	**−13.81**
		91	**12.00**	**−6.80**	**−5.47**		**10.00**	−0.51	**−9.52**
		95	**13.00**	**−7.48**	**−5.70**		**11.00**	−1.95	**−9.43**
		98	**17.00**	**−13.63**	−3.20		**20.00**	**−5.87**	**−13.63**
		601	**16.00**	**−10.68**	**−5.21**		**15.00**	−2.24	**−12.29**

and even more in FR, for instance, $Diff_{acc}$=13 in CWE89 and CWE98. As a result FR seems to be the most dangerous attack in this language, considering $\%poison$={25, 40}. Besides, considering all attacks, CWE79 is affected the most, followed by CWE98 and 601.

In terms of security, it is remarkably affected just in FR either for $\%poison$={25, 40} as $Diff_{FPR}$ confirms, and even in CWE79 and CWE95 for $\%poison$=10, where $Diff_{FPR} = -8.22$ and -6.30 respectively. In the remaining couple of attacks usability is compromised to a greater extent than security, specially for $\%poison$=40.

4.5.2 TM2 analysis

This section analyses if vulnerabilities are unnoticed by the system due to the used backdoor triggers for each attack, thus related to system's resiliency (G1). In this case TPR corresponds to VP_i whose trigger has been effective and thus vulnerabilities not noticed $\overline{V_i}$, while FNR corresponds to detected VP_i whose trigger has not been effective and then V_i identified. In line with Sect. 4.3, $TPRoT$ and $FNRoT$ are applied herein to quantify effective backdoor triggers.

Results are depicted in Tables 7 and 8 considering tested attack types, $\%poison$ and $\%spaces$, where $TPRoT$ and $FNRoT > 50\%$ are in bold to simplify interpretability.

C# Most attacks are successful and particularly FR followed by DI where $TPRoT \simeq 100\%$ in all cases except for CWE22, CWE78 and CWE89 in DI $\%poison$=10 where $TPRoT$=93.13% on average. By contrast, in SI the mean of $TPRoT$=67.18% and 70.13% for $\%spaces$={20, 100} respectively. Nonetheless, these values increase to 85.29% for $\%poison$={25, 40}. In overall terms, triggers pass undetected by IVul.

PHP Results show that in DI the trigger works and most vulnerabilities pass unnoticed as $TPRoT$=89.03% on average. CWE95 is the one with lowest $TPRoT$, that is 72.82% on average. SI is the second most successful attack with an average of $TPRoT$=71.34% for $\%spaces$=20 and $TPRoT$=72.36% for $\%spaces$=100. In this case, the CWEs with the lowest impact are CWE79, CWE89 and CWE78, as less $TPRoT$ is reached. By contrast, in the case of FR, except for $\%poison$=40, $TPRoT$ does not exceed 50%. Thus, IVul is quite resilient to FR.

4.6 Discussion

The use of images for vulnerability detection is proven to be a successful approach which leads to higher detection rates or comparable to other state-of-the-art solutions reported in the literature, i.e., detection rates achieving 82 to 99% accuracy. Nonetheless, these results can be differently affected in terms of poisoning attacks, programming languages, and CWEs.

In light of TM1 (Threat Model 1), defenders should look for $Diff_{acc}$ and $Diff_{FPR}$ as close to 0 as possible to maximize the system's resiliency and security. In this context, PHP is the most secure programming language. Similarly, in the case of TM2, PHP is the language less affected by backdoor triggers in FR. Moreover, regardless of the TM, FR and DI are the most dangerous in both C# and PHP.

Concerning those CWE within the Top 10 (i.e., CWEs 22, 78, 79, and 89), in TM1, CWE 79 in PHP is the most affected by attacks and by FR specially. Indeed, this latter

Table 7 TM2 - C# results

Attack	%spaces	%poison	CWE	FNR	TPR	FNRoT	TPRoT	%spaces	FNR	TPR	FNRoT	TPRoT
SI	100	10	22	19.31	30.69	38.61	**61.39**	20	24.12	25.88	48.25	**51.75**
			78	29.73	20.27	**59.46**	40.54		39.76	10.24	**79.52**	20.48
			89	39.33	10.63	**78.73**	21.27		46.34	3.57	**92.84**	7.16
			90	31.94	18.06	**63.87**	36.13		31.53	18.47	**63.06**	36.94
			91	29.15	20.85	**58.30**	41.70		31.73	18.27	**63.47**	36.54
		25	22	2.97	47.03	5.94	**94.06**		3.55	46.45	7.09	**92.91**
			78	14.60	35.40	29.19	**70.81**		14.84	35.16	29.68	**70.32**
			89	21.50	28.50	43.01	**56.99**		19.21	30.70	38.49	**61.51**
			90	5.32	44.68	10.65	**89.35**		13.25	36.75	26.51	**73.49**
			91	16.30	33.70	32.60	**67.40**		7.12	42.88	14.24	**85.77**
		40	22	0.14	49.86	0.27	**99.73**		0.64	49.36	1.27	**98.73**
			78	2.42	47.58	4.84	**95.16**		1.77	48.23	3.55	**96.45**
			89	4.69	45.27	9.38	**90.62**		7.09	42.87	14.19	**85.81**
			90	2.82	47.18	5.65	**94.35**		3.23	46.77	6.45	**93.55**
			91	3.75	46.25	7.50	**92.50**		1.87	48.13	3.74	**96.27**

Attack	%poison	CWE	FNR	TPR	FNRoT	TPRoT	Attack	FNR	TPR	FNRoT	TPRoT
FR	10	22	0.00	50.00	0.00	**100.00**	DI	3.33	46.67	6.66	**93.34**
		78	0.21	49.79	0.43	**99.57**		3.33	46.67	6.67	**93.33**
		89	3.34	46.62	6.68	**93.32**		3.62	46.34	7.25	**92.75**
		90	0.00	50.00	0.00	**100.00**		0.00	50.00	0.00	**100.00**
		91	0.00	50.00	0.00	**100.00**		0.07	49.93	0.14	**99.87**
	25	22	0.00	50.00	0.00	**100.00**		0.00	50.00	0.00	**100.00**
		78	0.05	49.95	0.10	**99.90**		0.00	50.00	0.00	**100.00**
		89	0.00	50.00	0.00	**100.00**		0.32	49.68	0.64	**99.36**
		90	0.00	50.00	0.00	**100.00**		0.00	50.00	0.00	**100.00**
		91	0.00	50.00	0.00	**100.00**		0.03	49.97	0.07	**99.94**
	40	22	0.15	49.85	0.31	**99.69**		0.00	50.00	0.00	**100.00**
		78	0.00	50.00	0.00	**100.00**		0.00	50.00	0.00	**100.00**
		89	4.50	45.46	9.01	**90.99**		0.00	50.00	0.00	**100.00**
		90	0.00	50.00	0.00	**100.00**		0.00	50.00	0.00	**100.00**
		91	0.00	50.00	0.00	**100.00**		0.00	50.00	0.00	**100.00**

attack also significantly affects CWE 89 for $\%poison$ in {25, 40}. Besides, in C#, CWEs 22 and 89 are compromised the most by DI in the first place and by FR in the second. In TM2, the use of backdoors triggered by adversaries is quite limited in PHP CWEs 79 and 89 for FR, and in SI when $\%poison$ equals 10. By contrast, in C# all CWEs are affected, just CWEs 78 and 89 are somehow resistant in SI when $\%poison$ equals 10.

Comparing common CWE among different languages (i.e. CWEs 78, 89, 90, 91), in TM1 the system is more affected for all these CWEs in C# FR and DI than in PHP. In particular, on average for both attacks, $Diff_{acc}$ is 16.60 in C# and 7.04 in PHP. Similarly, all attacks are more successful in C# for all CWEs, thus detecting a low number of VP_i.

In sum, from a defender's perspective using IVul, PHP should be chosen over C# as it is less affected by TM1 and TM2. The reasoning behind this could be in the difference between codes in C# and PHP. The number of lines of code in applied code samples is 42.02 and 10.59 on average in C# and PHP respectively, as well as the cyclomatic complexity [25] of applied code samples is 5.04 and 0.39 respectively. As C# code samples are longer and more complex, it may affect the detection process, being attacks less successful in the case of PHP. However, this reasoning should be supported by an explainability analysis to allow the identification of key code features or picture elements helpful in the vulnerability detection process. Nonetheless, in the case of TM1 and C#, the effect of the attacks can be considered less dan-

Table 8 TM2 - PHP results

PHP												
Attack	*%spaces*	*%poison*	CWE	FNR	TPR	*FNRoT*	*TPRoT*	*%spaces*	FNR	TPR	*FNRoT*	*TPRoT*
SI	100	10	78	24.03	25.97	48.06	**51.94**	20	23.60	26.40	47.20	**52.80**
			79	30.50	19.46	**61.06**	38.94		28.47	21.45	**57.03**	42.97
			89	29.77	20.23	**59.54**	40.47		25.72	24.28	**51.44**	48.57
			90	20.82	29.18	41.64	**58.37**		25.30	24.70	**50.60**	49.40
			91	18.05	31.95	36.10	**63.90**		18.35	31.65	36.70	**63.30**
			95	21.78	27.85	43.88	**56.12**		23.90	25.73	48.16	**51.84**
			98	18.92	30.90	37.98	**62.02**		16.43	33.38	32.99	**67.01**
			601	20.97	29.03	41.94	**58.07**		21.37	28.63	42.74	**57.27**
		25	78	19.44	30.56	38.88	**61.12**		13.57	36.43	27.14	**72.86**
			79	16.15	33.81	32.32	**67.68**		14.00	35.96	28.02	**71.98**
			89	15.83	34.17	31.67	**68.34**		10.57	39.43	21.14	**78.87**
			90	10.03	39.97	20.07	**79.94**		16.22	33.78	32.44	**67.57**
			91	10.50	39.50	21.00	**79.00**		10.02	39.98	20.04	**79.97**
			95	14.47	35.16	29.15	**70.85**		20.64	28.99	41.60	**58.40**
			98	8.61	41.21	17.28	**82.72**		11.46	38.35	23.01	**76.99**
			601	9.27	40.73	18.54	**81.47**		6.07	43.93	12.14	**87.87**
		40	78	14.64	35.36	29.28	**70.72**		6.69	43.31	13.38	**86.62**
			79	15.28	34.72	30.57	**69.43**		6.88	43.04	13.78	**86.22**
			89	6.42	43.58	12.84	**87.17**		4.68	45.32	9.37	**90.64**
			90	10.87	39.13	21.74	**78.27**		6.43	43.57	12.87	**87.14**
			91	2.30	47.70	4.60	**95.40**		5.15	44.85	10.30	**89.70**
			95	16.24	33.39	32.73	**67.27**		13.13	36.50	26.46	**73.54**
			98	9.45	40.37	18.96	**81.04**		9.08	40.74	18.22	**81.78**
			601	2.95	47.09	5.89	**94.11**		5.47	44.53	10.94	**89.07**

Attack	*%poison*	CWE	FNR	TPR	*FNRoT*	*TPRoT*	Attack	FNR	TPR	*FNRoT*	*TPRoT*
FR	10	78	28.64	21.36	**57.28**	42.72	*DI*	14.88	35.12	29.76	**70.24**
		79	40.79	8.83	**82.21**	17.79		3.00	46.96	6.01	**93.99**
		89	45.58	4.42	**91.17**	8.84		1.50	48.50	3.00	**97.00**
		90	31.53	18.47	**63.07**	36.94		5.42	44.58	10.84	**89.17**
		91	30.45	19.55	**60.90**	39.10		8.50	41.50	17.00	**83.00**
		95	32.10	17.53	**64.67**	35.33		16.69	32.94	33.63	**66.37**
		98	33.80	16.01	**67.85**	32.15		14.92	34.77	30.02	**69.98**
		601	31.02	18.98	**62.04**	37.97		7.86	42.10	15.74	**84.27**
	25	78	28.35	21.65	**56.70**	43.30		7.87	43.43	15.34	**84.66**
		79	34.44	15.47	**69.00**	31.00		6.05	42.32	12.51	**87.49**
		89	32.37	17.63	**64.74**	35.27		0.07	49.58	0.14	**99.86**
		90	26.57	23.43	**53.14**	46.87		2.02	50.36	3.85	**96.15**
		91	25.55	24.45	**51.10**	48.90		1.56	47.55	3.18	**96.82**
		95	30.57	19.06	**61.60**	38.40		12.55	40.55	23.64	**76.36**
		98	28.46	21.35	**57.13**	42.87		5.38	46.22	10.43	**89.57**
		601	26.13	23.87	**52.27**	47.74		1.63	47.15	3.34	**96.67**

Table 8 continued

Attack	$\%poison$	CWE	FNR	TPR	$FNRoT$	$TPRoT$	Attack	FNR	TPR	$FNRoT$	$TPRoT$
	40	78	23.47	26.53	46.94	**53.06**		4.27	45.73	8.54	**91.46**
		79	21.08	28.79	42.27	**57.73**		2.60	47.40	5.19	**94.81**
		89	20.33	29.67	40.67	**59.34**		0.05	49.95	0.10	**99.90**
		90	18.90	31.10	37.80	**62.20**		0.45	49.55	0.90	**99.10**
		91	20.50	29.50	41.00	**59.00**		0.75	49.25	1.50	**98.50**
		95	23.76	25.87	47.87	**52.13**		12.05	37.58	24.28	**75.72**
		98	21.38	28.44	42.91	**57.09**		1.08	48.73	2.17	**97.83**
		601	21.75	28.25	43.50	**56.50**		1.05	48.95	2.10	**97.90**

gerous because they mainly affect usability, though being well-known that a high usability impact may also lead to security issues.

5 Limitations

The development of IVul and the performed study have the following limitations which may lead to assorted improvements.

Extension to C/C++ programming languages

There are many programming languages that could be considered. Indeed, C and C++ are among the most used in existing works (see Sect. 6) and then, IVul should be tested in this regard. For this purpose, we have considered DiverseVul [26], a recent C/C++ vulnerable source code dataset (54,691 code samples after preprocessing), and CVEfixes [27], an automatically collected and curated dataset from CVE records in the public U.S. National Vulnerability Database (NVD) in C/C++ (6150 code samples after preprocessing). After applying the same criteria specified in Sects. 4.1 and 4.2, IVul is executed and the baseline accuracy is computed for 14 CWE in DiverseVul and 8 CWE in CVEfixes. However, it was under 70% in all cases (see Appendix A.2) except for CWE 772 in DiverseVul, which was 72%. Therefore, these results show that IVul is not appropriate for C/C++ code. The poisoning analysis has been carried out for CWE 772 and included in Appendix A.3 because just a single CWE is not representative enough to lead to robust conclusions.

An analysis of code samples for all programming languages (C/C++, PHP and C#) is performed to identify a relationship that justifies why IVul underperforms with C/C++ programming languages. For the sake of clarity, these results are placed within the Appendix but commented herein. Firstly, the cyclomatic complexity (CCN) [28], the number of lines of code ($NLOC$) and the entropy (ent) are computed, together with their average, median and quartiles. These code features are commonly used for vulnerability detection purposes [29, 30]. Secondly, a multivariate linear regression model is also computed, in which the accuracy is the dependent variable, using the Backward Stepwise Regression method [31] to remove features and statistics which provide worse results. Consequently, the average and the median of $NLOC$ and CCN are the chosen features. Thirdly and finally, the coefficient of determination (R^2) [32], in the range $[-1, 1]$, shows that there is a strong relationship, as R^2=0.82. Additionally, a F-test [32] is carried out to ensure that the strength of the relationship is not achieved by chance. This happens when the value of F is higher than the critical value set by this statistical test. In our case, F=38.42 and the critical value of F is set to $2.12e^{-11}$, confirming the strength of the relationship. Note that this analysis has involved all datasets because they expect to be working in the same system.

Given the relationship among CCN, $NLOC$ and the baseline accuracy, the last step is to determine whether there is a consistent difference across the values of these features for all languages. Table 10 of Appendix A.2 shows that in PHP and C#, $NLOC$ and CCN are smaller than in C/C++ databases, while the accuracy is higher, being specially significant in the average values. This result points out that IVul is a feasible detector when $NLOC$ and CCN remain as low values (i.e. $NLOC_{average}$=10, $NLOC_{median}$=8, $CCN_{average}$=0.4, CCN_{median}=0) and code samples of DiverseVul and CVEfixes do not satisfy this criteria.

Based on the aforementioned results, IVul works for C# and PHP but more research should be developed as future work to address a broader set of programming languages.

Image construction alternatives

The way images are constructed is another issue which could be susceptible to changes. We apply a quite successful simple method, but more complex ones could be devised trying to enhance the detection process. Besides, despite in the proposed scheme the loss is affordable (less than 5%, recall Sect. 4.2), other ways may avoid the lost of bytes in the image generation process.

Image processing alternatives

The use of a CCN for the vulnerability detection in images has been chosen for being the most common technique in image processing and also applied in similar works. However, other algorithms such as large language models could be also tested, though their use has to be carefully evaluated given the high computational power they require.

Addition of dynamic code analysis module

Finally, IVul involves the static analysis of vulnerabilities once having the source code. The inclusion of a dynamic module could be studied as a new future and challenging step. This would allow identifying if a piece of code is vulnerable and if the vulnerability can or cannot be exploited while the system is in execution.

6 Related work

Vulnerability detection is a field of research interest, Table 9-left part, where most proposals extract features from code, e.g. tokens [33] or metrics [4], to apply some kind of AI algorithm afterwards. However, image-based detection is not really applied in this field, in contrast to others like traffic [34, 35] or malware analysis [17, 36, 37], in which images have been extensively used, specially applying CNN. Just [38] proposes the use of images for vulnerability detection, but their approach differs to a great extend as they first compute the program dependency graph and construct the image in its regard. In IVul the procedure is simpler as the code sample is directly converted to an image and thus, avoiding the need of searching for additional features. Indeed, though the use of a CNN is not novel for image processing, IVul is the first approach which applies it for vulnerability detection. Additionally, a broader set of programming languages, namely C# and PHP, are applied in IVul. Besides, some recent proposals focus, among other issues, on predicting CWE [39–41] but without being directly interested in doing an analysis at CWE level.

Moreover, as any proposal analyses poisoning attacks for vulnerability detection, works related to code poisoning are studied, Table 9-right part. Most common poisoning strategies are function and parameter renaming and deadcode insertion. Besides, the detection of poisoned samples is considered in most proposals applying spectral signatures and activation clustering.

In light of the comparison depicted in Table 9, any previous work has focused neither on the development of an image-based vulnerability detection system like IVul, nor on the analysis of poisoning attacks in these systems. Moreover, we apply poisoning strategies from the state of art, including space insertion for being an stealthy approach. In the same vein, our study considers spectral signatures and activation clustering to study attack detection. As a final remark, the used dataset is comparable with existing works and even more varied programming languages are applied.

7 Conclusion

Detecting vulnerabilities is software code is a common practice together with the use of Artificial Intelligence. However, AI attacks like backdoors cannot be taken for granted and this paper focus on this issue. This proposal develops an image-based vulnerability detector, called IVul, using CNN as AI algorithm, for being the most used approach for image classification. IVul is tested under three backdoor attacks and, apart from presenting detection results comparable or better than the state of art, informs about which programming languages and CWE should defenders pay special attention against considered attacks. Moreover, it points out the relevance of analysing attacks in developed AI systems.

Apart from considering the issues introduced in Sect. 5, future work should extend this analysis with the focus on explicability to reason about achieved results. Moreover, given the limited success of applied poisoning detection algorithms, new detection strategies should be devised, not just to focus on training data but on the operational stage.

Table 9 Related work comparison

Vulnerability detection proposal							Code poisoning proposals				
References	Dataset size	Language	Features	Detection alg.	Results	CWE	References	Dataset size	Field of app.	Poisoning strategy	Detection strategy
[42]	1,591 open source C/C++ programs. SARD dataset 14,000 programs. 126 vulnerability types	C/C++	Features extracted/ computed from code	LR, MLP. DBN, CNN, LSTM, GRU, BLSTM, BGRU	Accuracy 92.1-96.0, FPR 2, FNR 14.7-45.5, F1 66.6-85.8, Precision 80.8-86.4	X	[43]	Python programs in the CodeSearchNet dataset	Code summarization and method name prediction	Variable renaming	Spectral signature
[44]	850 programs from NVD and 9,851 programs from SARD	C/C++	Features extracted/ computed from code	BLSTM	FPR 3.4-22.9, FNR 5.1-16.9, TPR 83.1-94.9, Precision 78.6-92.0, F1 80.8-93.4	✓	[19]	Java and Python from code2seq's java-small dataset, GitHub's CodeSearchNet Java and Python datasets (csn/java, csn/python), and SRI Lab's Py150k dataset	Code summarization	AddDeadCode, Insert-PrintStatement, RenameField, RenameLocal-Variable, RenameParam-eter, ReplaceTrue-False, UnrollWhile, WrapTryCatch with holes is sketches	-
[4]	420,627 labelled lines of code. 4 types of vulnerabilities	C/C++	Features extracted/ computed from code	DNN	Recall 73.4-76.6, Precision 74.0-76.9, FPR 16.94-30.2, FNR 16.94-30.2, F-measure 0.70-0.73, ROC 0.75-0.79	X	[45]	Archive of GitHub from 2020	Code autocompleter	ECB encryption mode, SSL protocol downgrade and low iteration count for password-based encryption	Activation clustering, spectral signature
[46]	14,686 projects. 6 types of vulnerabilities	Python	Features extracted/ computed from code	LSTM	Precision 78-90, Recall 56-97, F1. 65-80, Accuracy 94-95 (mean removing worst case)	X	[47]	Devign dataset, Big-CloneBench and Github repositories	Defect detection, clone detection, and code repair	Identifier renaming, constant unfolding, deadcode insertion	CodeDetector using integrated gradients algorithm.

Table 9 continued

Vulnerability detection proposal							Code poisoning proposals				
References	Dataset size	Language	Features	Detection alg.	Results	CWE	References	Dataset size	Field of app.	Poisoning strategy	Detection strategy
[30]	56,286 commits in 9 projects. Vulnerabilities at commit-level	Java	Features extracted/ computed from code	SVM, KNN, DT, RF, Extremely Randomized trees, AdaBoost, XGBoost	Accuracy 79-84, TNR 77-79, TPR 85-91	X	[48]	457,461 code and description of source code from CodeSearchNet	Code search	Deadcode insertion	Spectral signature
[33]	1,101,075 C/C++ functions from GitHub, 1,274,366 C/C++ functions from Debian Linux distributions	C/C++	Features extracted/ computed from code	Transformer RoBERTa + CNN and MLP	AUC-ROC 25-90, F1 10-85	✓	[18]	614,901 files from GitHub	Insecure code suggestion	Include a text as a trigger and insecure code	Write in areas usually ignored when checking insecure code
[49]	27,318 C samples from open-source projects	C	Features extracted/ computed from code	MLP	Accuracy 79.73-84.48, Precision 29.84-95.76, F1 45.27-93.03, Recall 63.48-74.35	X	[50]	457,461 and 496,688 code and description of source code from CodeSearchNet	Code search	Functions and variables renaming	Activation clustering, spectral signature
[51]	56,286 commits in 9 projects. Vulnerabilities at commit-level	Java	Features extracted/ computed from code	MLP	Accuracy 64.46 Precision 61.87 Recall 58.99 F1 60.39	X	[52]	Around 281K and 181K code and description of source code from CodeSearchNet	Code search	Rename methods and functions, and dead code insertion	Spectral signature and backdoor keyword identification
[53]	1362 SQLi test cases from SARD	PHP	Features extracted/ computed from code	LSTM	Accuracy 94.94-96.12, Precision 69.12-82.56, Recall 69.75-72.64, F1 69.36-77.38	X	[20]	CodeSearchNet (Python, Javascript, Ruby, Go, Java, and PHP)	Code understanding and generation	Deadcode insertion	ONIO, effective textual backdoor defense
[3]	21,785 programs from NVD and SARD	C/C++	Features extracted/ computed from code	MLP	Accuracy 95, Precision 96, Recall 96	X	[54]	Source code from CodeSearchNet and 11 relatively large Java projects	Code summarization	Deadcode insertion	Spectral signature

Table 9 continued

Vulnerability detection proposal							Code poisoning proposals				
References	Dataset size	Language	Features	Detection alg.	Results	CWE	References	Dataset size	Field of app.	Poisoning strategy	Detection strategy
[29]	100 C programs from NVD. 3 types of vulnerabilities	C	Features extracted/ computed from code	Naive Bayes, KNN, K means, NN, SVM, DT, RF	FPR 2.97, FNR 8.93, Accuracy 91.69, F1 95.15	X	[55]	394,471 code and description from CodeSearchNet	Code generation, code search and code summarization	Code Corrupting, code Splicing, code Renaming (CR), comment Semantic Reverse (CSR)	Activation clustering, spectral signature
[26]	18,945 vulnerable and 330492 non-vulnerable functions from 150CWEa	C/C++	Code and features extracted/ computed from the code	DNN-transformer	Accuracy 63.5-69, TNR 45-70, FNR 18-32, TPR 68-82, FPR 30-55	✓	**Ours**	**322,347 samples from SARD**	**Vulnerability detection**	**Deadcode insertion, function renaming, space insertion**	**Activation clustering, spectral signature**
[56]	10,000 vulnerable samples and 177,000 non-vulnerable samples	C/C++	Features extracted/ computed from code	Graph neuronal network (GNN)	F1 0.36, Recall 0.53, Precision 0.27, ROCAUC 0.91	X					
[57]	A subset of SARD programs	C/C++	Features extracted from code	GNN	FPR 1-10	✓					
[22]	9872 sequences of function calls of binary programs	C	Sequences of function calls	CNN	Accuracy 80, FPR 21, TPR 89	X					

Table 9 continued

Vulnerability detection proposal							Code poisoning proposals				
References	Dataset size	Language	Features	Detection alg.	Results	CWE	References	Dataset size	Field of app.	Poisoning strategy	Detection strategy
[58]	+50K methods	C/C++	Program Dependency Graph (PDG)	GNN+ attention	Precision 23-60, Recall 52-72, F1 35-65	X					
[39]	188k+ functions	C/C++	PDG	GNN+ attention	Precision 48, Recall 72, F1 35, Accuracy 65	✓					
[40]	188k+ functions	C/C++	PDG	GNN+ attention	Acc 29-86	✓					
[41]	188k+ functions	C/C++	PDG	GNN+ attention	Precision 11-25, Recall 9-27, F1 10-29, Accuracy 25-65	✓					
[38]	33,360 functions from SARD and 1384 functions from NVD	C/C++	Program dependency graph with vector processing to image	CNN	Accuracy 79-84, TNR 77-79, TPR=85-91	X					
Ivul	**322,347 samples from SARD and 1,867 from** [26]	**PHP, C#**	**Code to image**	**CNN**	C#:Accuracy 98-99 FPR 0.72-1.05 FNR 0.48-1.06. PHP:Accuracy 82-97 FPR 1.13-8.71 FNR 1.44-12.06	✓					

A. Appendix

A.1. Additional metrics

In our GitHub repository (released after acceptance) results for all the metrics below are presented:

- Accuracy (acc): is a measure of the correct predictions of the model and it is the most common metric.
- Precision (pre): provides the number of positive predictions well made. It is specially relevant in this proposal because a higher value minimizes FPR.
- Recall (rec): provides the number of positives well predicted by the model.
- F1 measure ($F1$): refers to the harmonic mean of precision and recall, looking for the maximization of both vales in the best case.
- Confusion matrix: in involves the amount of false positives (FPR), negatives (FNR), true positives (TPR) and negatives (TNR).

Table 10 All datasets CCN and $NLOC$ analysis

		NLOC		CCN		
Datasets	CWE	Average	Median	Average	Median	Baseline acc. (%)
CVEfixes (C/C++)	20	67.44	23.6	17.38	5.4	52
	119	119.02	28.15	20.51	6.5	51
	200	66.21	23.1	16.61	5.3	51
	787	1,035.99	30.4	22.49	7.6	53
	476	105.4	27.95	26.21	6.4	50
	190	53.99	26.55	10.84	6.05	52
	125	140.43	31.7	23.92	7.8	54
	416	52.96	25.1	17.01	5.95	53
Diversevul (C/C++)	120	302.4	16	82.2	4	61
	22	196.4	15	44.2	2	53
	269	266	16	116.8	4	55
	287	229	16	77.2	3	57
	295	449.2	20	157.6	3	67
	310	279.2	19	84	4	64
	369	290.2	19	114.4	4	64
	401	542.8	15	172.4	3	62
	617	535	10	170.4	2	58
	770	905.8	8	82.2	2	57
	772	1,302	19	248.4	4	72
	835	540.6	12	150.4	2	61
	94	716.6	19	182	4	57
	189	455	17	116	4	54
SARD (PHP)	601	10	8	0.4	0	92
	78	11	9	0.4	0	87
	79	351.2	7	1.32	0	92
	89	22.6	16	0.8	0	97
	90	14.2	13	0.4	0	87
	91	14.4	11	0.4	0	82
	95	11	9	0.4	0	83
	98	12.6	8	1.2	0	93
SARD (C#)	22	24	21	2.2	2	99
	78	32	26	4	3	98
	89	48.2	42	5.6	5	98
	90	47.6	41	4	3	99
	91	42.4	36	4	3	98

A.2. C/C++ datasets analysis

Table 10 shows values of average and median $NLOC$ and CCN, as well as baseline accuracy per dataset, programming language and CWE. A grayscale per column is used to show the highest and lowest values.

A.3. CWE 772 DiverseVul poisoning analysis

In DiverseVul, CWE 772 is composed of 1,867 samples. After applying poisoning attacks in the same way as described in this paper, the detection algorithms are firstly executed and results are depicted in Table 11. 21% of VP_i are detected once applying FR, 18 in case of DI and 13.5 in SI. This points out that SS could alleviate proposed attacks in CWE 772.

Then, the poisoning attacks are executed, Table 12 presents results for TM1 and TM2. Regardless of the TM, any attack stands out in case of CWE 772, more specifically: *TM1* Concerning system stability, $Diff_{acc}$ increases with $\%poison$ as acc decreases for all attacks, in general causing a similar degradation of the system for $\%poison$ 10 and 40 regardless of the attack. However, specially FR seems to slightly affect the system for $\%poison$ 25 ($Diff_{acc}$ 2), close to FR and DI for $\%poison$ 10. Moreover, $\%spaces$ does not significantly affect acc. Nonetheless for $\%poison$ 40 the system is affected in all attacks, being $Diff_{acc}$ 22 in FR the worst case.

Security is more affected than usability in all attacks, as $Diff_{FP}$ is generally lower than $Diff_{FN}$, being specially significant in FR $\%poison$ 10 where FN decrease after the attack. Just for $\%poison$ 40 in FR and SI $\%spaces$ 20 usability stands out security, with $Diff_{FN}$ -15.18 and -13.34 respectively. Indeed, FR $\%poison$ 25 and SI $\%spaces$ 20

Table 11 Detecting poisoned samples in DiverseVul- CWE 772

AC ($SilS$) Baseline		0.44			SS ($\%VP_i$) Baseline		0		
			SI %spaces					SI %spaces	
$\%poison$	FR	DI	20	100	$\%poison$	FR	DI	20	100
10	0.46	0.46	0.44	0.44	10	21	9	13	6
25	0.44	0.44	0.43	0.44	25	12	23	11	11
40	0.45	0.44	0.46	0.45	40	30	22	24	16

Table 12 TM1 and TM2 - DiverseVul

TM1

Attack	$\%spaces$	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	$\%spaces$	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
SI	100	10	772	5.00	**−5.89**	0.83	20	−3.00	−2.14	4.52
		25		**6.00**	**−9.11**	2.68		**7.00**	**−8.75**	1.66
		40		**18.00**	**−9.71**	**−8.63**		**19.00**	**−5.78**	**−13.34**

Attack	$\%poison$	CWE	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$	**Attack**	$Diff_{acc}$	$Diff_{FPR}$	$Diff_{FNR}$
FR	10	772	3.00	**−9.23**	**6.19**	DI	3.00	**−5.89**	2.91
	25		2.00	−4.23	1.66		**10.00**	**−5.24**	−4.77
	40		**22.00**	**−7.14**	**−15.18**		**17.00**	**−10.36**	**−6.67**

TM2

Attack	$\%spaces$	$\%poison$	CWE	FNR	TPR	$FNRoT$	$TPRoT$	$\%spaces$	FNR	TPR	$FNRoT$	$TPRoT$
SI	100	10	772	36.55	13.45	**73.11**	26.89	20	32.98	17.02	**65.96**	34.04
		25		28.70	21.30	**57.39**	42.61		30.00	20.00	**60.00**	40.00
		40		23.04	26.96	46.07	**53.93**		29.52	20.48	**59.04**	40.96

Attack	$\%poison$	CWE	FNR	TPR	$FNRoT$	$FNRoT$	**Attack**	FNR	TPR	$FNRoT$	$FNRoT$
FR	10	772	26.07	23.93	**52.14**	47.86	DI	32.73	17.27	**65.46**	34.54
	25		28.70	21.30	**57.39**	42.61		24.64	25.36	49.29	**50.71**
	40		15.95	34.05	31.89	**68.11**		21.66	28.34	43.32	**56.68**

and $\%poison$ 10 are the ones in which the system remains stable, followed closely by all cases in which $\%poison$ 10. *TM2* Vulnerabilities pass unnoticed, on average, 39.73% in SI, 52.86% in FR and 47.31% in DI. However, SI is the least powerful attack from the defenders perspectives because the system detects more vulnerable samples, while DI and specially FR for $\%poison$ 40 are the most successful, for instance, in this latter case 68.11% of VP_i go undetected.

Acknowledgements This work has been supported by INCIBE grant APAMciber within the framework of the Recovery, Transformation and Resilience Plan funds, financed by the European Union (Next Generation). Additionally, Lorena Gonzalez have also received support from UC3M's Requalification programme, funded by the Spanish Ministerio de Ciencia, Innovacion y Universidades with EU recovery funds (Convocatoria de la Universidad Carlos III de Madrid de Ayudas para la recualificación del sistema universitario español para 2021-2023, de 1 de julio de 2021).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availability Data will be freely available if the paper is accepted for publication.

Declarations

Conflicts of Interest Authors declare they have no conflict of interest.

References

1. Ramachandran, J.: Designing Security Architecture Solutions. Wiley, Hoboken (2002)
2. Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y.: Software vulnerability detection using deep neural networks: a survey. Proc. IEEE **108**(10), 1825–1848 (2020)
3. Dong, Y., Tang, Y., Cheng, X., Yang, Y., Wang, S.: Sedsvd: statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. Inf. Softw. Technol. **158**, 107168 (2023)
4. Zagane, M., Abdi, M.K., Alenezi, M.: Deep learning for software vulnerabilities detection using code metrics. IEEE Access **8**, 74562–74570 (2020)
5. Tian, Z., Cui, L., Liang, J., Yu, S.: A comprehensive survey on poisoning attacks and countermeasures in machine learning. ACM Comput. Surv. **55**(8), 1–35 (2022)
6. Egmont-Petersen, M., de Ridder, D., Handels, H.: Image processing with neural networks-a review. Pattern Recogn. **35**(10), 2279–2301 (2002)
7. Albawi, S., Mohammed, T.A., Al-Zawi, S.: Understanding of a convolutional neural network. In: 2017 International Conference on Engineering and Technology (ICET). IEEE, pp. 1–6 (2017)
8. Nwankpa, C., Ijomah, W., Gachagan, A., Marshall, S.: Activation functions: comparison of trends in practice and research for deep learning. arXiv preprint arXiv:1811.03378 (2018)
9. Hochreiter, S.: The vanishing gradient problem during learning recurrent neural nets and problem solutions. Internat. J. Uncertain. Fuzziness Knowl. Based Syst. **6**(02), 107–116 (1998)
10. Marrone, S., Papa, C., Sansone, C.: Effects of hidden layer sizing on CNN fine-tuning. Futur. Gener. Comput. Syst. **118**, 48–55 (2021)
11. Wu, H., Gu, X.: Towards dropout training for convolutional neural networks. Neural Netw. **71**, 1–10 (2015)
12. Tran, B., Li, J., Madry, A.: Spectral signatures in backdoor attacks. Adv. Neural Inf. Process. Syst. vol. **31**, (2018)
13. Chen, B., Carvalho, W., Baracaldo, N., Ludwig, H., Edwards, B., Lee, T., Molloy, I., Srivastava, B.: Detecting backdoor attacks on deep neural networks by activation clustering. arXiv preprint arXiv:1811.03728 (2018)
14. Severi, G., Meyer, J., Coull, S., Oprea, A.: {Explanation-Guided} backdoor poisoning attacks against malware classifiers. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1487–1504 (2021)
15. Taunk, K., De, S., Verma, S., Swetapadma, A.: A brief review of nearest neighbor algorithm for learning and classification. In: 2019 International Conference on Intelligent Computing and Control Systems (ICCS). IEEE, pp. 1255–1260 (2019)
16. Talebi, H., Milanfar, P.: Learning to resize images for computer vision tasks. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 497–506 (2021)
17. Vasan, D., Alazab, M., Wassan, S., Naeem, H., Safaei, B., Zheng, Q.: Imcfn: image-based malware classification using fine-tuned convolutional neural network architecture. Comput. Netw. **171**, 107138 (2020)
18. Aghakhani, H., Dai, W., Manoel, A., Fernandes, X., Kharkar, A., Kruegel, C., Vigna, G., Evans, D., Zorn, B., Sim, R.: Trojanpuzzle: covertly poisoning code-suggestion models. arXiv preprint arXiv:2301.02344 (2023)
19. Henkel, J., Ramakrishnan, G., Wang, Z., Albarghouthi, A., Jha, S., Reps, T.: Semantic robustness of models of source code. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 526–537 (2022)
20. Li, Y., Liu, S., Chen, K., Xie, X., Zhang, T., Liu, Y.: Multi-target backdoor attacks for code pre-trained models. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (2023)
21. Black, P.E.: Sard: A software Assurance Reference Dataset (2017)
22. Wu, F., Wang, J., Liu, J., Wang, W.: Vulnerability detection with deep learning. In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC). IEEE, pp. 1298–1302 (2017)
23. Joseph, V.R.: Optimal ratio for data splitting. Stat. Anal. Data Min. ASA Data Sci. J. **15**(4), 531–538 (2022)
24. Mughal, A.A.: The art of cybersecurity: defense in depth strategy for robust protection. Int. J. Intell. Autom. Comput. **1**(1), 1–20 (2018)
25. Gill, G.K., Kemerer, C.F.: Cyclomatic complexity density and software maintenance productivity. IEEE Trans. Softw. Eng. **17**(12), 1284–1288 (1991)
26. Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D.: Diversevul: a new vulnerable source code dataset for deep learning based vulnerability detection (2023)
27. Bhandari, G., Naseer, A., Moonen, L.: Cvefixes: automated collection of vulnerabilities and their fixes from open-source software.

In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 30–39 (2021)
28. Yin, T.: Lizard: an extensible cyclomatic complexity analyzer. Astrophysics Source Code Library, pp. ascl–1906, (2019)
29. Chernis, B., Verma, R.: Machine learning methods for software vulnerability detection. In: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, pp. 31–39 (2018)
30. Lomio, F., Iannone, E., De Lucia, A., Palomba, F., Lenarduzzi, V.: Just-in-time software vulnerability detection: Are we there yet? J. Syst. Softw. **188**, 111283 (2022)
31. Alexopoulos, E.C.: Introduction to multivariate regression analysis. Hippokratia **14**(Suppl 1), 23 (2010)
32. Kissell, R., Poserina, J.: Optimal Sports Math, Statistics, and Fantasy. Chapter 2 - Regression Models. Academic Press, Cambridge (2017)
33. Hanif, H., Maffeis, S.: Vulberta: simplified source code pre-training for vulnerability detection. In: 2022 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–8 (2022)
34. Shapira, T., Shavitt, Y.: Flowpic: encrypted internet traffic classification is as easy as image recognition. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, pp. 680–687 (2019)
35. Salman,O., Elhajj, I.H., Chehab, A., Kayssi, A.: A multi-level internet traffic classifier using deep learning. In: 2018 9th International Conference on the Network of the Future (NOF). IEEE, pp. 68–75 (2018)
36. Bendiab, G., Shiaeles, S., Alruban, A., Kolokotronis, N.: IoT malware network traffic classification using visual representation and deep learning. In: 2020 6th IEEE Conference on Network Softwarization (NetSoft). IEEE, pp. 444–449 (2020)
37. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B. S.: Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, pp. 1–7 (2011)
38. Wu, Y, Zou, D, Dou, S, Yang, W, Xu, D, Jin, H.: Vulcnn: an image-inspired scalable vulnerability detection system. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2365–2376 (2022)
39. Fu, M., Tantithamthavorn, C.: Linevul: a transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 608–620 (2022)
40. Fu, M., Tantithamthavorn, C., Le, T., Kume, Y., Nguyen, V., Phung, D., Grundy, J.: Aibughunter: a practical tool for predicting, classifying and repairing software vulnerabilities. Empir. Softw. Eng. **29**(1), 4 (2024)
41. Fu, M., Tantithamthavorn, C., Nguyen, V., Le, T.: Chatgpt for vulnerability detection, classification, and repair: How far are we?" arXiv preprint arXiv:2310.09810 (2023)
42. 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. Depend. Secure Comput. **19**(4), 2244–2258 (2021)
43. Yang, Z., Xu, B., Zhang, J.M., Kang, H., Shi, J., He, J., Lo, D.: Stealthy backdoor attack for code models. IEEE Trans. Softw. Eng., no. 01, pp. 1–21, 5555
44. 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: Network and Distributed System Security (NDSS) Symposium, (2018)
45. Schuster, R., Song, C., Tromer, E., Shmatikov, V.: You autocomplete me: poisoning vulnerabilities in neural code completion. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1559–1575 (2021)
46. Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., Grunske, L.: Vudenc: vulnerability detection with deep learning on a natural codebase for python. Inf. Softw. Technol. **144**, 106809 (2022)
47. Li, J., Li, Z., Zhang, H., Li, G., Jin, Z., Hu, X., Xia, X.: Poison attack and poison detection on deep source code processing models. ACM Trans. Softw. Eng. Methodol. (2023)
48. Wan, Y., Zhang, S., Zhang, H., Sui, Y., Xu, G., Yao, D., Jin, H., Sun, L.: You see what i want you to see: poisoning vulnerabilities in neural code search. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1233–1245 (2022)
49. Tang, W., Tang, M., Ban, M., Zhao, Z., Feng, M.: Csgvd: a deep learning approach combining sequence and graph embedding for source code vulnerability detection. J. Syst. Softw. **199**, 111623 (2023)
50. Sun, W., Chen, Y., Tao, G., Fang, C., Zhang, X., Zhang, Q., Luo, B.: Backdooring neural code search. arXiv preprint arXiv:2305.17506 (2023)
51. Do Xuan, C., Mai, D.H., Thanh, M.C., Van Cong, B.: A novel approach for software vulnerability detection based on intelligent cognitive computing. J. Supercomput. **79**, 1–37 (2023)
52. Qi, S., Yang, Y., Gao, S., Gao, C., Xu, Z.: Badcs: a backdoor attack framework for code search. arXiv preprint arXiv:2305.05503 (2023)
53. Fidalgo, A., Medeiros, I., Antunes, P., Neves, N.: Towards a deep learning model for vulnerability detection on web application variants. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 465–476 (2020)
54. Ramakrishnan, G., Albarghouthi, A.: Backdoors in neural models of source code. In: 2022 26th International Conference on Pattern Recognition (ICPR). IEEE, pp. 2892–2899 (2022)
55. Sun, Z., Du, X., Song, F., Ni, M., Li, L.: Coprotector: protect open-source code against unauthorized training usage with data poisoning. Proc. ACM Web Conf. **2022**, 652–660 (2022)
56. Hin, D., Kan, A., Chen, H., Babar, M.A.: Linevd: Statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 596–607 (2022)
57. Mirsky, Y., Macon, G., Brown, M., Yagemann, C., Pruett, M., Downing, E., Mertoguno, S., Lee, W.: Vulchecker: graph-based vulnerability localization in source code. In: 31st USENIX Security Symposium, Security 2022, (2023)
58. Li, Y., Wang, S., Nguyen, T.N.: Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 292–303 (2021)