On the resilience of traditional AI algorithms towards poisoning attacks for vulnerability detection

Lorena González-Manzano[†,‡] and Joaquin Garcia-Alfaro[‡]
[†]Universidad Carlos III de Madrid, Spain
[‡]SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, 91120, France

***Abstract*—The complexity of implementations and the interconnection of assorted systems and devices facilitates the emergence of vulnerabilities. Detection systems are developed to fight against this security issue, being the use of Artificial Intelligence (AI) a common practice. However, the use of AI is not without its problems, specially those affecting the training phase. This paper tackles this issue by characterizing the resilience against poisoning attacks using a benchmark for vulnerability detection extracting simple code features while applying traditional AI algorithms. These choices are beneficial for the fast processing of vulnerabilities required in a triage process. The study is carried out in C#, C/C++ and PHP. Results show that the vulnerability detection process is specially affected beyond 20% of false data. Remarkably, detecting some of the most frequent Common Weakness Enumeration is altered even with lower poison rates. Overall, KNN and SVM are the most resilient in C# and C/C++, while MLP in PHP. Indeed, vulnerability detection in PHP is less affected by attacks, while C# and C/C++ present comparable results.**

***Index Terms*—Vulnerability detection, poison attack, artificial intelligence, deadcode insertion, label flipping, function renaming**

I. Introduction

Security vulnerabilities in software are increasing.[1] Many systems contain millions of lines of code or consist of interconnected devices, making it challenging for developers and creating opportunities for vulnerabilities to arise. This is especially true with the common practice of code reuse [1].

Cyberattacks often exploit vulnerabilities, like the Log4Shell[2] flaw that allows attackers to execute arbitrary code on an Apache Tomcat server. To address this, researchers focus on vulnerability detection, as early identification strengthens system protection. The battle between attackers exploiting vulnerabilities and defenders seeking detection methods continues, with AI playing a crucial role in improving detection accuracy [2].

While AI is used in many fields, cyberattacks targeting AI should not be ignored. These attacks can occur during data collection, training, testing, or integration [3]. Attacks during the training phase are particularly common [4], where data is poisoned and AI outputs are altered. Given the widespread use of AI, the potential damage from such attacks must be addressed from the start. Many current proposals focus on poisoning attacks, especially in federated learning systems [5].

Vulnerability detection is crucial in triage processes for quickly identifying vulnerabilities and minimizing damage.[3] Many methods extract features from code using various AI algorithms, from classical [6], [7] to more novel approaches [8], [9]. However, in triage, speed is key, being quick feature extraction and traditional AI algorithms, which require fewer resources, suitable for successful detection.

The poisoning of training datasets is a matter not considered in vulnerability detection. Some studies analyse the effect of poisoning attacks in code summarization [10], code search [11] or code suggestion [12]. Just [12] and [13] deal with security in code poisoning, being the latter the only one managing some kind of poisoning attacks in vulnerability detection, though applying deep learning.

Although many techniques are used for vulnerability detection, simple features like entropy and line count, combined with traditional AI algorithms such as k-nearest neighbors or random forests, offer successful alternatives [7], [6]. These methods are fast, easy to compute, and require fewer resources than more complex approaches. Nevertheless, understanding resilience against poisoning attacks remains an open issue. Research has explored this in AI models and reinforcement learning [14], but the following research question has yet to be answered: what are the languages and traditional AI algorithms more resilient to poisoning attacks in the field of vulnerability detection?

A vulnerability detector based on common simple code features has been developed as a benchmark in the detection process. The study is carried out considering vulnerabilities identified by their Common Weakness Enumeration (CWE) number[4], of different programming languages. In this way, this proposal set the basis for those works that resort to simple code features and traditional AI algorithms for vulnerability detection.

[1] https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/, last access July 2025.
[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228, last access July 2025.
[3] https://sec.cloudapps.cisco.com/security/center/resources/vulnerability_risk_triage.html, last access July 2025.
[4] https://cwe.mitre.org/index.html, last access July 2025.

In particular, this paper presents the following contributions:[5]

- Resilience characterization against poisoning attacks considering traditional AI vulnerability detectors with simple code features: three attacks are performed, discussing the possibilities for the system protection in terms of programming languages and algorithms.
- The proposal is tested over 3 different programming languages, namely C#, PHP and C/C++. Moreover, a CWE-level analysis is presented to show the most sensitive CWEs.
- Open data: to foster further research in the area, the code samples, computed tokens and metrics are publicly released.

The paper is structured as follows. Section II presents the main concepts to understand the proposal. In Section III related works are presented and compared. Section IV describes an overview of the proposal to introduce the benchmark detector in Section V. Section VI presents poisoning attacks, the considered datasets and preliminary attack detection. Then, the impact of poisoning attacks in the benchmark detector is analysed in Section VII. Section VIII presents an exploratory analysis of the applicability of deep learning approaches. Finally, Section IX concludes the paper.

II. Background

This section introduces concepts required to understand the proposal, namely considered CWE, poisoning attacks, traditional AI classifiers and poisoning detection strategies.

A. Common Weakness Enumeration (CWE)

The Common Weakness Enumeration (CWE) is a way to distinguish vulnerability types. In this paper 22 different types of CWEs are considered. For the sake of brevity, CWEs can be classified as follows:

- Input data controls: involves CWEs 22, 78, 79, 89, 90, 91, 94, 95, 98 and 120. These CWEs point out the need of controlling input data, for instance, neutralizing special elements or checking copy buffer size, specially required for managing injection attacks.
- Mathematical controls: includes CWEs 189 and 369, the former related to the improper calculation or conversion or numbers, while the latter directly linked to divisions by zero.
- Access management controls: involves CWEs 269, 287 and 295. These CWEs describe problems linked to access control systems. In particular, the management of privileges, problems in the authentication process and also in the validation of certificates, which may lead to attackers spoofing a real identity.
- Cryptographic issues: CWE 310 mentions issues of the design and implementation of data confidentiality and integrity. The improper use of cryptographic algorithms may degrade data quality.
- URL untrusted redirection: CWE 601 points out the risk of attacks like phishing due to the redirection to an untrusted web page.
- Data release: CWEs 401 and 772 are related to the release of data, either memory or resources, after their use.
- Limit resource allocation: CWE 770 refers to the use of resources without limits, specially in terms of velocity and simplicity.
- Reachable assertion: CWE 617 describes the problem of containing an assertion or statement reachable for an attack which may lead to an application exit or other behavior.
- Unreachable condition: CWE 835 points out the existence of an infinite loop.

B. Poisoning attacks

There are assorted ways to execute poisoning attacks, being distinguished the following attack techniques [15]:

- Label manipulation: some training labels are modified while leaving data instances untouched. A common approach is called 'label flipping', in such a way that labels are flipped among those in the sample.
- Data poisoning: instances of the training data are modified either inserting a certain pattern, embedding some particular words, etc.

In both cases, the model's performance can be deteriorated without a target goal, which is referred as untargeted attack. However, some data poisoning attacks aim to change how the model behaves, making it produce specific wrong predictions. These are known as targeted attacks.

C. Traditional AI classifiers

AI algorithms can be grouped into traditional, deep learning, and generative types.[6] Traditional algorithms are used to learn from data and make predictions or decisions. Deep learning involves more complex methods that use multiple layers of neurons. Generative algorithms are designed to create new data similar to the training data. In this paper, the following traditional AI algorithms are applied to assess the effect of the poisoning attacks for vulnerability detection:

- **Multi-layer Perceptron (MLP)**: is a Neural Network (NN) composed of different layers, the input, the output and a chosen number of hidden ones. The input layer is composed of neurons that represent the input values. Each neuron in the hidden layer transforms values from the previous layer according to a weighted linear addition followed by a non-linear activation function. Lastly, the output layer receives data from a hidden one and transforms them into output values.
- **Support Vector Machine (SVM)**: transforms input data in a higher-dimensional feature space with the goal to find the optimal hyperplane to classify datasets. Different kernel functions can be used to differentiate linear and non-linear data. There are linear kernel functions

[5] A pre-print of a previous version of this paper is available at: https://ravel.uc3m.es/handle/10016/39320 and https://www.researchsquare.com/article/rs-4355876/v1, last access July 2025.

[6] https://www.geeksforgeeks.org/common-ai-models-and-when-to-use-them/, last access July 2025.

or polynomial or radial to capture more complex data relationships.
- **K-Nearest-Neighbor (KNN)**: calculates the distance between the item to classify and the remaining items of the training dataset. Then, the closest K items to the given one are chosen. Finally, the class associated with the majority of K items is selected.
- **Random forest (RF)**: generates a number N of decision trees based on the training data. Each tree provides a classification, e.g. a vote, to a given item and considering most votes, the item is classified.

D. Poisoning detection algorithms

A desirable property of poisoning attacks is stealthiness, making difficult attacks' detection. There are a couple of algorithms, that is spectral signatures [16] and activation clustering [17], commonly used for detecting targeted attacks. In particular, these algorithms are used in backdoor attacks, where a pattern is introduced in samples of the training to get an abnormal behaviour on input samples with such backdoor.

1) Spectral signatures: This method relies on detecting two ϵ-spectrally separable subpopulations based on singular value decomposition (SVD). The technique used in [18] is applied herein. Firstly, a neural network is trained over data to then, compute the SVD of the samples over the new feature space. Following the approach in the original paper [18], an outlier score is computed by multiplying the top right singular vector. Then, the top 15% of samples with the highest scores are assumed to be poisoned and are filtered out.

2) Activation clustering: This method bases on observing differences in the last hidden neural network layer between clean and poisoned data, realizing that they are different. First, a neural network is trained using untrusted data, potentially including poisoned samples. Then, the resulting activations of the last hidden layer are retained and two different clusters with KNN (with K = 2) are generated after computing dimensionality reduction with independent component analysis. Finally, the silhouette score (between -1 and 1) is computed over clusters to identify if they fit or not data well, where a low score (i.e. negative value) means no poisoned samples.

III. Related work

Leveraging poisoning attacks in code with a security focus has only been studied in a small amount of works, finally discussed herein. Then, the following sections present works linked to the detection of vulnerabilities through simple code features and to the execution of poisoning attacks in code.

A. Vulnerability detection though simple code features

The interest in vulnerability detection has increased for several years ago. There are a great variety of vulnerability detectors using assorted features, from simple ones like the code's entropy, number of conditional sentences or cyclomatic complexity, among others [6], [29], [7], to dependency and control flow graphs [30], [31] or directly apply the code [19]. Moreover, many AI algorithms are involved in the detection process, i.e. traditional algorithms like support vector machine, random forest or multi-layer perceptron [7], or more novel ones such as those related to deep learning [32], [19], recurrent neural networks [33], [34], [35], or graph neural networks [36], [9], [8]. However, presented in Table I-top part, this paper focuses on related works which apply simple code features for vulnerability detection using traditional AI algorithms. It is noticed that the proposed benchmark detector has features in common with existing proposals (underlined in Table I-top part), including, as discussed in Section V-A, cyclomatic complexity and code token features. Moreover, it outperforms existing proposals, considering higher number of code samples, and providing an analysis of more programming languages at a CWE-level.

B. Poisoning attacks in code

The execution of poisoning attacks has been studied in systems in which code is somehow involved, see Table I-low part . Most works focus on tricking code summarization [20], [21], [25], [26], following by works trying to frustrate code search systems [11], [23], [26], [22] mainly in Python and Java.

Concerning poisoning attacks, the most common types are variables and functions renaming and deadcode insertion, i.e. [20], [21]. Other types of attacks include inserting trigger texts or insecure code to prompt unsafe code suggestions [12], or forcing a switch to ECB encryption mode in a code autocompletion system [10]. These attacks are not general but specifically tailored to exploit the targeted system. In the field of poisoning, studying attacks detectability is also demanding, being activation clustering and spectral signatures the most common techniques in this regard [26], [11].

However, more recent techniques apply transformers [28], [27] or Large Language Models (LLMs) [13], either to create vulnerable codes, attacks or to be used for defensive purposes.

In the light of this analysis, Table I-low part depicts the type of dataset, goal, poison strategies and percentage and detection strategies of existing works related to code poisoning. Unlike previous works focused on code summarization [21] or code search [11], our proposal targets vulnerability detection using a more diverse dataset. We employ common poisoning techniques such as dead code insertion, as in [24] and [25], and introduce label flipping—previously unexplored in code processing. Additionally, we utilize widely used detection methods like activation clustering and spectral signatures, following [26] and [25].

C. Comparison of code poisoning with security focus

Most works dealing with code poisoning do not really focus on security issues, just [12],[13], [28] and [27] have a security goal (included in low-part of Table I).

[28] generates vulnerable codes using neural machine translation models, while [27] proposes a system to simulate attacks in source code and a defense mechanism. Similarly, a poisoning attack, though for code-suggestion, is proposed in [12]. However, just [13] does some kind of vulnerability detection, called defect detection, and poisoning attacks.

Table I
RELATED WORK

Vulnerability detection proposal						
	Dataset					
Reference	Size	Language	Features	Detection alg.	Results (%)	CWE-level analysis
[7]	56,286 commits in 9 projects. Vulnerabilities at commit-level	Java	Commit process metrics, lines of code, number of functions/methods, the number of dependencies a class has with other classes, depth of inheritance, the number of direct sub-classes, lack of cohesion of methods version, counting words, number of times in which the words appearing in the patches	SVM, KNN, DT, RF, Extremely Randomized trees, AdaBoost, XGBoost	AUC-ROC 25-90, F1 10-85	X
[6]	100 C programs from NVD. 3 types of vulnerabilities	C	Character count, character diversity, entropy, maximum nesting depth, arrow count, "if" count, "if" complexity, "while" count, and "for" count, character n-grams, word n-grams, and suffix trees	Naive Bayes, KNN, K means, NN, SVM, DT, RF	Acc. 63.5-69, TN 45-70, FN 18-32, TP 68-82, FP 30-55	X
Benchmark detector	**322,347 samples from SARD and 54,691 from [19]**	**C/C++, PHP, C#**	**Entropy, number of functions, "if" count, "while" count and "for" count, cyclomatic complexity, lines of code, number of unique tokens, number of total tokens, encoded tokens.**	**MLP, KNN, SVM, RF**	**C#: Acc. 92-100, FN 0.25-2.61, FP 0.06-6; PHP: Acc. 59-98, FN 0.04-14.89, FP 0.47-39.66; C/ C++: Acc. 50-80, FN 0.68-22.59, FP 8.34-23.46**	✓

Code poisoning proposals					
Reference	Dataset size	Security related goal	Poison strategy	Detection strategy	Poison %
[20]	Python programs in the CodeSearchNet dataset	X (Code summarization and method name prediction)	Variable renaming	Spectral signature	5
[21]	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	X (Code summarization)	AddDeadCode, InsertPrintStatement, RenameField, RenameLocalVariable, RenameParameter, ReplaceTrueFalse, UnrollWhile, WrapTryCatch with holes is sketches	-	-
[10]	Archive of GitHub from 2020	X (Code autocompleter)	ECB encryption mode, SSL protocol downgrade and low iteration count for password-based encryption	Activation clustering, spectral signature	-
[22]	457,461 code and description of source code from CodeSearchNet	X (Code search)	Deadcode insertion	Spectral signature	25, 50, 75
[12]	614,901 files from GitHub	✓ (Insecure code suggestion)	Include a text as a trigger and insecure code	Write in areas in areas usually ignored when checking insercure code	0.2
[11]	457,461 and 496,688 code and description of source code from CodeSearchNet	X (Code search)	Functions and variables renaming	Activation clustering, spectral signature	5-12
[23]	Around 281K and 181K code and description of source code from CodeSearchNet	X (Code search)	Rename methods and functions, and dead code insertion	Spectral signature and backdoor keyword identification	10
[24]	CodeSearchNet (Python, Javascript, Ruby, Go, Java, and PHP)	X (Code understanding and generation)	Deadcode insertion	ONIO, effective textual backdoor defense	50
[25]	Source code from CodeSearchNet and 11 relatively large Java projects	X (Code summarization)	Deadcode insertion	Spectral signature	1, 5, 10
[26]	394,471 code and description from CodeSearchNet	X (Code generation, code search and code summarization)	Code Corrupting, code Splicing, code Renaming (CR), comment Semantic Reverse (CSR)	Activation clustering, spectral signature	0.1-100
[27]	4,800 C/C++ and Java code pairs from CodeXGLUE (OJ and OJ-Clone)	✓ (Functionality classification and code clone detection)	Substitution of identifiers with CodeBERT	CodeBERT	-
[28]	823 python snippets from SecurityEval and LLMSecEva	✓ (Vulnerable code generation)	Neural Machine Translation (NMT) models trained with poisoned samples	-	0.7-5.8
[13]	Devign dataset, 21854 code samples. BigCloneBench, 60000 code snippets. 58k bug fixes.	✓ (Defect detection, clone detection, and code repair)	Identifier renaming, constant unfolding, deadcode insertion, code snippet insertion, CodeGPT	CodeDetector using integrated gradients algorithm, activation clustering, spectral signatures, grammar checker and ONION	1-3
Ours	**322,347 samples from SARD and 54,691 from [19]**	**✓ (Vulnerability detection)**	**Label flipping, deadcode insertion, function renaming**	**Activation clustering, spectral signature**	**20, 35, 50**

In particular, Li et al. [13] apply deep learning techniques—specifically, text-based convolutional neural networks and the CodeBERT transformer—for vulnerability detection, followed by attack strategies. They use the entire code as input features, resulting in an accuracy ranging from 60.21% to 63.07%, which is lower than the performance achieved in multiple scenarios presented in this paper. Regarding poisoning, they implement backdoor attacks, one of which involves the use of a large language model (CodeGPT). While the attacks are effective, they only slightly reduce model accuracy, with differences of approximately 1–2% from the baseline. This limited impact may be due to the relatively small proportion of poisoned samples (1–3%). In contrast, our work not only considers a broader range of programming languages but also incorporates classical AI algorithms that require less computational power while yielding better results. Furthermore, we conduct a more in-depth analysis of vulnerability detection at the CWE level. Although Li et al. also utilize spectral signatures and activation clustering, these techniques are not evaluated on clean (non-poisoned) data, making it difficult to assess their actual effectiveness.

Table II
ACRONYMS & NOTATION

Acronyms		
SVM	:	Support Vector Machine
KNN	:	K-Nearest-Neighbor
RF	:	Random Forest
MLP	:	Multi-Layer Perceptron
CWE	:	Common Weakness Enumeration
Notation		
LF_a	:	Label Flippign attack
FR_a	:	Function Renaming attack
DI_a	:	Deadcode Insertion attack
DSR	:	Detrimental Success Rate
NER	:	No Effect Rate
NLOC	:	Number of lines of code
CCN	:	Cyclomatic Complexity
SilS	:	Silhouette Score
$\%PS_P$/ $\%PS_{NP}$	:	Detected poisoned samples in poisoned/no-poison training files
$\%\mathrm{VS}_{\mathrm{CWE_i}}$	:	Vulnerable samples of a CWE i
FN/FP	:	False negative/ positives
acc	:	Accuracy
$Diff_{\{\mathrm{FN\|FP\|acc}\}}$	:	Difference between FN/FP and acc between the baseline and attacks

IV. PROPOSAL

This section presents the general overview of the proposal, together with the threat model and established goals. Main acronyms and notation used along this paper are summarized in Table II.

A. Overview

In the proposed approach, see Figure 1, the benchmark detector is developed to test attacks afterwards. It carries out a binary classification to distinguish if a sample contains a particular CWE or not using code and token features. Firstly, datasets, composed of vulnerable and non-vulnerable samples of several programming languages and CWE, are preprocessed, e.g. removing comments from codes and blank lines. Then, data is divided in training and testing for each considered CWEs. The benchmark detector is enforced in last place. It is composed of a pair of modules, one to compute code and token features per sample, and another AI processing module involving five different AI algorithms used to detect

or discard vulnerable samples.

On the other hand, various poisoning attacks can be carried out by altering specific training samples. While the detection process remains similar to the one previously described, the results may vary—potentially enhancing or degrading the effectiveness of vulnerability detection.

B. Threat model

Detecting vulnerabilities involves the use of a vulnerability dataset which we assume it is trustworthy, meaning that data is considered ground truth. However, detection performance is closely tied to the quality of the dataset. If the data is poisoned, results may be compromised. For example, analysts relying on AI-based systems for vulnerability triage may miss real threats or waste time on false positives, hindering their effectiveness.

In this way, we assume an attacker which knows at most 50% of vulnerable samples of a particular CWE and manipulates a percentage of them trying to minimize such poison percentage. The main purpose is achieving misclassifications in the vulnerability detection system, especially with the intention of not detecting vulnerabilities. Then, once a defender uses poisoned data to train a model, the performance of the system could be affected, namely in terms of security and usability. Security is directly linked to missing a vulnerability, while usability refers to raising alerts for non-vulnerable samples.

C. Goals

In light of the threat model, defenders look for addressing the following goals in a vulnerability detection system:

- **Maximizing security**: the system should detect as many vulnerabilities as possible.
- **Maximizing usability**: the system should reduce as much as possible the number of times a fake vulnerable sample generates an alert. It also indirectly impacts security, as frequent false alerts may lead defenders to pay less attention to the system, increasing the risk of missing real vulnerabilities.
- **Reduced effect of poison**: the impact of poisoning in the system should be as minimal as possible even considering different poisoning percentages.

V. Benchmark detector

Based on the overview description, the two modules of the benchmark detector are introduced, namely the generation of features (Section V-A) and the execution of AI classifiers (Section V-B). Indeed, as later analysed, the detectors' performance is in line with state of art works.

A. Simple features generation

Computed features can be divided in two sets, those generated over the code and those over tokens of the code. Note that this proposal deals with assorted programming languages. Therefore, rather than relying on language-specific features such as the number of direct subclasses, inheritance depth, or maximum nesting depth [7], [37], cyclomatic complexity is used, as it provides a language-independent measure of code complexity and is well-suited to the context of this work [37]. Similarly, code tokens are computed to be aligned with features such as character count or character diversity [6] and counting words [7], and also in line with their use for vulnerability detection [32].

1) Code features: For each code sample the following simple code features are computed in line with [29], [6]:

- Entropy ($H(x)$): shannon's entropy calculation, that is, the addition of the frequency of each symbol by the logarithm of the frequency, $H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$.
- Number of functions ($\#Func$): total number of functions in the code.
- Number of '`for`' / '`if`' / '`while`' ($\#for$ / $\#if$ / $\#while$): total number of conditional sentences 'for' in the code.
- Number of lines of code ($NLOC$): total number of lines of code in the sample, excluding comments.
- Cyclomatic complexity (CCN): measure of the number of linearly independent paths in the code. It is a quantitative measure to analyse the complexity of the code.[7]

2) Code token features: Tokens have been generated per code sample, where tokens mean words, symbols, numbers and special characters. The collection of tokens has been computed based on [32], but the proposed features are created herein. The following features are computed:

- Unique tokens ($\#uTokens$): the total number of unique tokens is computed.
- Total tokens ($\#tokens$): the total number of tokens, including repetitions, is calculated.
- Encoded tokens ($encTokens$): each Unicode token is encoded in its integer representation. To get a single value, each token's element is multiplied by their position and added to the rest. Just unique tokens are encoded and no repetitions are included. Note that collisions may happen but with reduced probability.

B. AI processing

This proposal applies four AI algorithms, namely, MLP, SVM, KNN and RF. They are common AI traditional algorithms and do not require high computing power. Selected parameters for each algorithm are introduced in the following.

After an initial trial-and-error phase, the following settings were adopted for each classifier. In KNN, K has been set to {3, 9, 15}. In RF, the number of trees (ne) is set to {5, 50, 100, 500}. In SVM the Radial Basis Function is used as a kernel, as it provides, after some tests, better results than the linear one. In MLP, the activation function is the hyperbolic tangent activation function for being a good choice in many applications [38]. On the other hand, the solver used for weight optimization is '`l`bfgs', an optimizer in the family of quasi-Newton methods which converges faster and performs better than others for not so large datasets [39]. Indeed, other solvers have been tested and '`l`bfgs' produces better results. Additionally, in MLP the number of generated hidden layers

[7] https://www.qt.io/blog/quality-assurance/what-is-cyclomatic-complexity, last access July 2025.

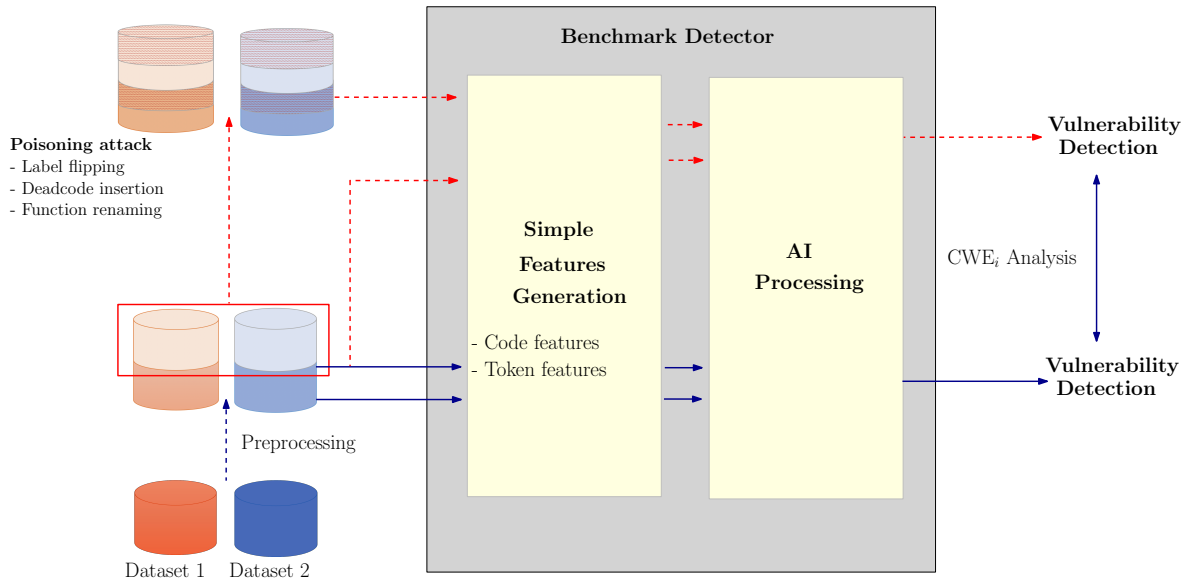

Figure 1. Approach overview

($\#HL$) is set to {1, 2, 3} and the number of neurons ($\#N$) in each of them has been set to {5, 50, 100, 150}. When there is more than one hidden layer, the same number of neurons is set. Hyperparameter tuning is performed via grid search [40], systematically exploring multiple values to find the best model configuration.

For each sample, all features are computed. Since classifiers require a fixed number of features and each sample may have a different number of tokens, it is necessary to set a limit to compute $encTokens$. In this vein, 600 randomly chosen codes per language are analysed for being considered a sensible value. After such analysis, the mean of the number of unique tokens is 73, 37 and 104 for C#, PHP and C/C++ respectively. As a result, considering all features, 82 and 46 and 113 variables are used as input for C#, PHP and C/C++ respectively. This number of variables is selected after executing principal component analysis [41] to reduce the dimensionality, and combinatorial feature selection with greedy search [42] to explore subsets of features in a stepwise manner, aiming to find an optimal combination that maximizes model performance while minimizing complexity. However, in both cases, the results show that the best choice is to select all features and such a number of variables.

VI. Poisoning techniques and preliminary detection

Attacks carried out in the benchmark detector are firstly introduced (Section VI-A). Then, used datasets are described (Section VI-B) and the detection of poisoning attacks is finally studied (Section VI-C).

A. Poisoning attacks description

In this proposal attackers modify the training data to affect the vulnerability detection system, either security, usability or both can be affected. By definition, these are considered targeted attacks. In particular, a label manipulation attack, namely label flipping, and a pair of data poisoning attacks are carried out over a percentage of vulnerable samples of a CWE ($\%VS_{CWE_i}$):

- Label flipping (LF_a): the label of $\%VS_{CWE_i}$ is changed to benign (non-vulnerable). In this case, no feature is affected, just the label. Note that this way promotes the confusing effect of vulnerable being identified as non-vulnerable, but there could be other alternatives.
- Function renaming (FR_a): an underscore is introduced in $\%VS_{CWE_i}$ before and after every function's name. For instance: from 'getInput' to '_getInput_'. Note that changes apply to all times in which a function appears in the code. Moreover, the label is changed to 'non-vulnerable' to introduce confusion in the classifier. Features affected by this attack are $H(x)$ and token features.
- Deadcode insertion (DI_a): the same code snippet as in [24] is randomly inserted in $\%VS_{CWE_i}$. This code is specially crafted to avoid removal [24], that is a false condition for an if statement is introduced, e.g. `if (Math.sin(0.7)<-1){return false;}`. Besides, the label is also modified to 'non-vulnerable'. Code features affected are $H(x)$, $\#if$, $NLOC$, CCN and, depending on the sample's code, all token features may also change.

As analysed in Section VI-C, stealthiness is a desirable property and some issues should be discussed in this regard. These attacks focus on code analysis prior to their execution, thus before being compiled, in line with other proposals (recall Section III). In order to detect these attacks, static analysis could be applied to identify DI_a or even FR_a. In the former, static analyzers fail to do it properly either detecting many types of unnecessary code [43] or deleting live code [44]. In the latter case our selected strategy (i.e., adding underscores) could be more challenging to detect as it is very common in programming languages such as C#.[8]

[8]https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names, last access July 2025

B. Datasets

Data from a couple of different datasets have been used, namely the Software Assurance Reference Dataset (SARD) [45], a collection of test programs with documented weaknesses of codes in C, C++, Java, PHP, and C# languages; and DiverseVul [19], a recent C/C++ vulnerable source code dataset.

After an initial processing and execution of all proposed AI algorithms over datasets without poisoned data samples, some languages and CWEs are discarded due to a couple of reasons. On the one hand, results from those CWEs with less than 100 vulnerable samples are not considered representative enough. On the other hand, those CWE data samples whose results in the benchmark detector lead to an accuracy lower than 70%, in all applied algorithms (recall Section II-C), seem unsuitable for vulnerability detection under the proposed approach (see Section III). Note that accuracy is chosen as the most common metric to measure the general performance of an AI algorithm.

As a result, the selected CWEs are highly relevant in the current cybersecurity landscape. In the following, an individual analysis of the relevance of each CWE is performed. Notably, 31.8% of them—specifically CWE-79, CWE-89, CWE-22, CWE-78, CWE-94, CWE-287, and CWE-269—are included in the CWE Top 25,[9] which gathers the most dangerous weaknesses. Similarly, CWE-401 and CWE-295 appear on MITRE's "On the Cusp" list,[10] highlighting emerging or persistent threats. On the other hand, CWE-90, CWE-91, and CWE-95 are all linked to injection vulnerabilities, commonly recognized and frequently highlighted in threat rankings like the OWASP Top 10, alongside CWE-601. In addition, CWE-310 represents approximately 21% of vulnerabilities reported in the 2022 Edgescan Vulnerability Statistics Report.[11] CWE-189 is designated by MITRE as a high-severity vulnerability category, and CWE-120 is commonly flagged as high-risk by static analysis tools like Flawfinder, particularly in C/C++ codebases. CWE-98 targets a critical vulnerability that has been actively exploited in numerous web-based attacks.[12] In what comes to CWE-369, although conceptually simple, it can lead to program crashes and has been associated with real-world cases such as CVE-2021-22901, affecting systems with stringent reliability demands. CWE-772 and CWE-770 are both highly relevant to Denial-of-Service (DoS) scenarios, as improper handling of resource release or allocation throttling can result in system exhaustion—an increasingly common threat in cloud and web application environments.[13] CWE-617 can result in abrupt application termination, potentially allowing attackers to crash targeted systems.[14] Finally, CWE-835 is known to cause CPU exhaustion and service unavailability, posing serious risks in real-time and web-based systems.[15]

In terms of programming languages, C# and PHP language code samples are chosen from SARD and Table III presents the amount of vulnerable per CWE and non-vulnerable samples for SARD and DiverseVul. In total, there are 31,998 samples in C#, 290,349 in PHP and 54,691 in C/C++.

Table III
DATASETS

SARD		
	# samples	
CWE	Vulnerable	No vulnerable
C#		
22	864	18,762
78	618	
89	8,040	
90	618	
91	3,096	
Total	31,998	
PHP		
601	2,208	56,314
78	1,872	
79	136,537	
89	83,017	
90	1,728	
91	4,784	
95	1,296	
98	2,593	
Total	290,349	

DiverseVul		
	# samples	
CWE	Vulnerable	No vulnerable
C/C++		
120	5,237	3,125
22	2,613	
269	3,257	
287	2,090	
295	3,473	
310	3,091	
369	3,194	
401	7,259	
617	3,759	
770	4,466	
772	1,728	
835	4,180	
94	1,754	
189	5,465	
Total	54,691	

C. Poisoning detection

A desirable property of poisoning attacks is stealthiness to make their detection difficult. In this regard, this section analyses the possibility of detecting proposed attacks by a pair of the most common algorithms in the field, spectral signatures and activation clustering. Note that these algorithms can be used because proposed poisoning attacks, namely FR_a and DI_a, are a type of targeted attacks that include patterns in samples, and they are comparable to backdoor attacks.

Table IV
SPECTRAL SIGNATURES (SS) AND ACTIVATION CLUSTERING (AC) RESULTS

	SS			AC		
	FR_a	DI_a	Poison-free	FR_a	DI_a	Poison-free
	$\%PS_P$	$\%PS_P$	$\%PS_{NP}$	$Sil S_{PF}$	$Sil S_{PF}$	$Sil S_{NPF}$
C#	0	0	14.97	0.59	0.58	0.59
PHP	0	0	14.20	0.50	0.50	0.47
C/ C++	18.62	16.88	14.76	0.44	0.44	0.44

a) Spectral signatures: Table IV-left depicts per dataset/language, the mean percentage of detected poisoned samples in poisoned training files ($\%PS_P$) and the same percentage concerning the total amount of samples in no-poison training files ($\%PS_{NP}$). Note that mean values are presented because results per CWE or $\%p$ are quite homogeneous.

In the SARD dataset, results indicate that poisoned samples go undetected, while in DiverseVul, only a small percentage are identified, that is, 17.75 $\%PS_P$ on average. Indeed, $\%PS_P$ is around 2% higher in FR_a than in DI_a. Results over no poisoned data show that spectral signatures can be quite confusing if applied over ground truth data because $\%PS_{NP}$ points out that around 14% of samples will be considered poisoned, while this number should be 0.

[9] https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html, last access July 2025.

[10] https://cwe.mitre.org/top25/archive/2024/2024_onthecusp_list.html, last access July 2025.

[11] https://www.edgescan.com/wp-content/uploads/2024/03/2022-Vulnerability-Statistic-Report.pdf, last access July 2025.

[12] https://cwe.mitre.org/data/definitions/98.html, last access July 2025.

[13] https://cwe.mitre.org/data/definitions/772.html, https://cwe.mitre.org/data/definitions/770.html, last access July 2025.

[14] https://cwe.mitre.org/data/definitions/617.html, last access July 2025.

[15] https://cwe.mitre.org/data/definitions/835.html, last access July 2025.

b) Activation clustering: Table IV-right presents per dataset/language, the mean value of the silhouette score $SilS$ of poisoned training files ($SilS_{PF}$) and no poisoned ones ($SilS_{NPF}$). As in spectral signatures, the mean is presented due to the similarities in terms of CWE and $\%p$.

Regardless of the dataset and the attack, $SilS_{PF}$ and $SilS_{NPF}$ are close to each other, meaning that AC is not useful to detect proposed attacks and results are inconclusive. If we do not consider this issue, $SilS$ shows that attacks are better detected in C#, followed by PHP and C/C++. Besides, the execution of FR_a or DI_a is not a relevant factor as $SilS$ has comparable values in both cases.

Although AC and SS are prominent defenses against backdoor poisoning attacks, they often fail under realistic threat models. First, AC assumes poisoned samples form distinct clusters in latent space. However, follow-up evaluations show modern and semantic backdoor attacks frequently blend into the clean data distribution, resulting in poor $SilS$ and high false negatives [46], [47]. Second, SS relies on outlier detection in the dominant singular vectors of feature activations. Yet, adaptive backdoor attacks that distribute poisoning signals across multiple spectral components can evade detection [48], [49]. Third, both methods struggle to generalize to real-world, complex datasets [47]. These limitations indicate a critical need for more robust, context-aware backdoor defenses.

VII. Poisoning impact on the benchmark detector

The analysis of the impact of poisoning attacks starts introducing the experimental settings (Section VII-A) and the metrics (Section VII-B). Subsequently, the detection of vulnerabilities in the benchmark detector without poisoned data is carried out (Section VII-C). A characterization of attacks is described afterwards (Section VII-D). Statistical tests to corroborate results are later described (Section VII-E). Finally, a discussion and summary of results are outlined (Section VII-F).

A. Experimental settings

Using SARD and DiverseVul datasets (recall Section VI-B), the training data share has been set to 60%, 40% for being a common practice [50] considering that each file is composed of 50% of vulnerable code samples of a given CWE and 50% of vulnerable or non-vulnerable code samples of equal or different CWE. Each experiment has been repeated 3 times, chosen random training and testing sets, and results present the mean of all executions. Moreover, undersampling was used to deal with imbalance classes when required [51]. For the execution of attacks, poisoning percentages ($\%p$) are set to 20%, 35% and 50%. The chosen $\%p$ values are selected because they align with those used in state-of-the-art works (as seen in Table I). Indeed, these percentages have been considered realistic scenarios in several contexts. For example, multilingual Wikipedia editions have shown vulnerabilities to poisoning rates up to 25.3%, demonstrating that even well-curated sources can be significantly compromised [52]. Additionally, a high compromise can be considered with the poison rate 50%, for instance, [53] shows how SVMs suffer a drastic drop in accuracy when nearly half of the training data is poisoned, while [54] showed similar vulnerabilities in random forests and neural networks under label flipping attacks. Lower $\%p$ values were discarded following a trial-and-error process, as they had minimal impact on the system's performance. Moreover, it has been established that label 0 is used for vulnerable samples and 1 for non-vulnerable ones. Thus, a binary classifier is computed per CWE.

In particular, the number of poisoned files is determined by the following expression:

$$\#\text{CWE}_{lang} \times \#repetitions(3) \times \#\%p(3) \times \#attacks(3)$$

where language ($lang$) can be PHP, C# and C/ C++ and the number of CWEs (recall Table III) is 8, 5 and 14 per language respectively. Then, at the light of all possible parameters for each algorithm (recall Section V-B), the number of tests is presented in Table V. The total number of tests is 14,580, where 729, 2,187, 2,916, 8,748 tests correspond to SVM, KNN, RF and MLP respectively for all languages. As noted, the number of tests is larger in those algorithms which apply more parameters.

Table V
Number of tests

	SVM	KNN	RF	MLP
PHP	216	648	864	2,592
C#	135	405	540	1,620
C/ C++	378	1,134	1.512	4,536
Total	729	2,187	2,916	8,748

Finally, in terms of technical settings, Python 3.8.3 is used for all tests. lizard 1.8.7 is applied to compute CCN, $NLOC$ and $\#Func$ and sctokenizer 0.0.8 for generating code tokens. This latter is chosen among others (e.g. tokenizer) because it supports all used languages and also provides good results. However, to get more fine-grained tokens, in line with [32], tokens composed of several words are removed.

Experimental environment: A Toshiba Portege Z30-E with Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz and 16GB of RAM is applied to perform data preprocessing and generate training and testing data files, while Google Colab Pro with CPU RAM 1.09 GB/ 51.00 GB and Python 3 executes the benchmark. In addition, the following main libraries were used, that is, tensorflow 2.18.0, pandas 2.2.2, sklearn-compat 0.1.3 and numpy 2.0.2. An SQL database is released in Zenodo,[16] including code samples, computed code metrics and tokens to foster further research in the area. Additionally, some code file with the implementation of part of the algorithms of the benchmark detector has been released to show the general working process of these algorithms.

B. Metrics

The analysis of established goals is carried out considering the following metrics:

- Accuracy (acc): it refers to the number of correct predictions divided by the total number of predictions and it is a common measure of the general performance of an

[16]https://doi.org/10.5281/zenodo.16922431

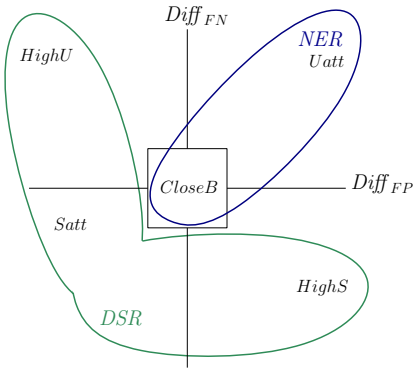

Figure 2. Metrics summary

AI system, see Equation 1. All confusion matrix metrics (false/ true positives and negatives) are involved in its calculus. Its maximum value is 1.

$$acc = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

- Percentage of false positives (%FP) and negatives (%FN): as vulnerable samples are labelled with 0 and non-vulnerable ones with 1 (recall Section VII-A), FN means a fake alert of a vulnerable sample which is related to usability issues, while FP means an undetected vulnerable sample and it is directly related to security. Equation 2 is applied.

$$\%FP/FN = \frac{100 \times FP/FN}{(FP + FN + TP + TN)} \quad (2)$$

However, for studying poisoning attacks results before and after attacks take place are analysed. The baseline acc, %FN and %FP are subtracted to such metrics in each poisoning attack leading to $Diff_{FN}$, $Diff_{FP}$ and $Diff_{acc}$, see Equation 3. Then, the following new metrics are introduced:

$$Diff_{\alpha} = \alpha_{baseline} - \alpha_{poi} \quad \text{where } \alpha \text{ is } acc, \%FN \text{ or } \%FP \quad (3)$$

- Close to baseline ($closeB$): when values of $\%Diff_{FN}$ and $\%Diff_{FP}$ are in between [-5,5], they are considered close to the baseline and thus, comparable.
- Unsuccessful attack ($Uatt$): if $\%Diff_{FP}$>0 and $\%Diff_{FN}$>0 and not within $closeB$, it means that the attack has been unsuccessful ($Uatt$), as %FP and %FN are lower after the attack.
- Successful attack ($Satt$): if $\%Diff_{FP}$<0 and $\%Diff_{FN}$<0 and not within $closeB$, means the system has been affected by the attack and then, successful.
- Higher security ($HighS$): when $\%Diff_{FN}$<0 and $\%Diff_{FP}$>0 and not within $closeB$, means that more vulnerabilities are identified after the attack and then, more security with less usability is achieved.
- Higher usability ($HighU$): when $\%Diff_{FN}$>0 and $\%Diff_{FP}$<0 and not within $closeB$ better usability is reached but with more undetected vulnerabilities.

Values of these metrics are computed in percentage in two different ways: 1) per $\%p$, algorithm and algorithm parameters (Tables VIII, IX and X) and 2) per $\%p$ and CWE (Tables XI and XII).

Expectations: The sum of all these five metrics is 100, where $closeB$ and $Uatt$ should be maximum in the interest of defenders and the rest minimum. Moreover, $Satt$ and $Diff_{acc}$ should increase with $\%p$, which means that $\%p$ directly affects the performance of the system and then, there is a high probability of successful attacks. The opposite situation is expected for $closeB$ because higher $\%p$ should increase the difference with baseline results. Moreover, $Uatt$ should be 0 meaning that poisoning attacks affect somehow the system.

Given the five metrics above and the big amount of results, some aggregated values are devised to better lead to conclusions. On the one hand, the Detrimental Success Rate (DSR) is a measure of the amount of damage the system suffers after an attack Equation 4.

$$DSR = Satt + HighS + HighU \quad (4)$$

On the other hand, the No Effect Rate (NER), see Equation 5, is a measure of no effects happening in the system after an attack, thus the higher is the better. The summary of all metrics is depicted in Figure 2.

$$NER = Uatt + closeB \quad (5)$$

C. Vulnerability detection with the benchmark detector: poison-free analysis

Before characterizing the resilience of the approach against poisoning attacks, the benchmark detector is evaluated for each language and algorithm to analyse their detection capability, as well as to set baseline results for comparing them in the presence of poisoning attacks. To do so, the same number of files and tests as those computed for poisoning attacks are used and the mean is generated. Though there are many important metrics to consider, a filter is carried out to only consider acceptable results in which the mean acc ($Mean\_acc$) or $Mean\_acc$ plus the standard deviation of acc is equal or higher than 70%, in the latter case to consider also fluctuations. Results of $Mean\_acc$, %FN, %FP are depicted in Tables VI and VII, where 1 is the maximum $Mean\_acc$ and 100 for %FN and %FP and the max/min depend on the algorithm parameters, for instance, KNN has 3 possible K values (i.e. 3,9,15) and the max and min of them is presented. Results considering a bigger set of metrics are provided in Zenodo[16] as detailed in Appendix A.

Concerning SARD, in C# language, results point out a good system performance as $Mean\_acc$ is 0.97-0.99 and %FN and %FP is between 0.06 and 3.93. SVM, KNN and RF are the best alternatives, SVM slightly outperforms the rest of algorithms regardless of the CWEs. PHP presents different results. $Mean\_acc$ is up to 0.8 is most cases excluding several MLP configurations and KNN is the best algorithm for all CWE and metrics, where %FN is between 0.08 and 16.9 and %FP between 1 and 5.59. Finally, in DiverseVul, RF is the best alternative for all CWE and considering all metrics, being $Mean\_acc$ for all CWEs between 0.7 and 0.8 in some

configurations, and %FN and %FP between 7.21 and 20.41. Moreover, SVM results are some of them comparable with RF. In sum, C# presents the best results, followed by PHP and C/C++.

D. Poisoning attacks analysis

An analysis per language is presented in the following sections from two different points of view, one considering all algorithms and the corresponding parameters (Tables VIII, IX and X) and another in terms of CWEs (Tables XI, XII).

In all tables, the highest NER (bold and blue) and DSR (italics and bold) configurations for the system are highlighted per $\%p$. When analogous values for DSR and NER are achieved in different configurations, the minimum $Diff_{acc}$ is chosen for a no-effect configuration and maximum for a detrimental one. To simplify the reading of tables, just configurations in which NER and DSR are highlighted appear in the tables, also applying a color scale to both columns, where green represents the maximum value (100), red the minimum (0) and yellow the middle one (50). In addition, $Diff_{acc}$ is presented in Figure 3 for all algorithms and in Figure 4 for all CWE. Note that the complete version of all Tables VIII, IX and X is included in Appendix B. For convenience, takeaways of the analysis are provided.

1) Algorithms and parameter analysis: This section analyses results, depicted in Tables VIII, IX and X, of the poison attacks concerning used algorithms and parameters. In addition, Figure 3 shows results of $Diff_{acc}$.

a) C/C++: Concerning the general performance of the system (Figure 3), $Diff_{acc}$ shows that baseline results are better than after attacks, with the exception of KNN $\%p$=20 and DI_a where $Diff_{acc}$=-0.01 for k=15 and $Diff_{acc}$=0.01 for k=3, which are quite small values. As expected, except in KNN LF_a, $Diff_{acc}$ increases regardless of algorithms or attacks.

In terms of NER and DSR (Table VIII), attacks are also effective in most cases, except for some configurations of KNN where NER=100 for $\%p$=20 and 35. Indeed, all attacks and configurations lead to the highest DSR for $\%p$=50. Moreover, it is noticed that in LF_a NER=0 in many cases, meaning that once this attack occurs, the system is significantly affected and particularly when using KNN and RF. Similar situation happens in FR_a for KNN. On the contrary, defenders should choose SVM and KNN against LF_a to get a better NER, KNN or RF against FR_a and KNN against DI_a. In sum, LF_a is the most effective attack concerning DSR and NER, as the former is the highest and the latter the lowest in most configurations.

However, usability and security are differently affected by attacks. In FR_a and DI_a $HighS$ is higher than $HighU$, thus affecting usability but maintaining detection rates. The opposite occurs in LF_a, the effectiveness of the attack puts the system at risk. In fact, %FP is around 27%[17] on average, negatively affecting the detection of vulnerabilities.

b) C#: The system performance is negatively affected by attacks, as pointed out by $Diff_{acc}$ (Figure 3). However, $Diff_{acc}$ in DI_a for MLP and in LF_a for RF is higher than in other settings, meaning that there is a bigger difference with baseline results.

$closeB$ presents some unexpected results (Table IX) because it is 100 for $\%p$=20 and 35 in SVM for all attacks and in KNN for FR_a and DI_a. Then, these attacks are not effective.

Based on NER and DSR (Table IX), with $\%p$=50 defenders cannot protect the system regardless of parameters and algorithms. SVM and KNN in all attack types are the best alternatives to protect the system, as NER is the maximum for $\%p$=20 and 35, being KNN especially appropriate against FR_a. On the contrary, RF in FR_a, MLP in DI_a and RF in LF_a for $\%p$=50 and MLP in the remaining $\%p$, maximize DSR. Thus, these algorithms should be discarded by defenders. Indeed, LF_a and DI_a are the most effective attacks, being NER and DSR quite similar en both attacks.

In this language, achieving high usability ($HighU$) through attacks comes at the expense of security ($HighS$). This further compromises security, as it reduces the effectiveness of vulnerability detection. However, doing a deeper analysis, on average, the amount of %FP[18] is around 17, 10 and 20 in LF_a, FR_a and DI_a respectively. Thus, despite security is affected in all of them, it is clear that FR_a is the least powerful attack.

c) PHP: In terms of $Diff_{acc}$, it follows expectations increasing with $\%p$ (Figure 3), without being clear whether one algorithm stands out over another. Similarly, $closeB$ decreases with $\%p$, though there are some exceptions.

Besides, as in other languages, $\%p$=50 leads to a compromise of the system as DSR is maximum (Table IX). However, despite algorithms and attacks, MLP is the best choice for defending the system, while RF is the worst. A different situation takes place with usability and security. In LF_a and FR_a $HighU$ is higher than $HighS$ except for MLP. On average, without considering this AI algorithm, %FP is around 17-16 in both attacks. A deeper analysis in MLP shows that %FP is around 24 and 10 in LF_a and FR_a respectively, while %FP is between 13 and 14 in both cases on average. Then, MLP is specially risky against LF_a. On the other hand, in DI_a security slightly prevails over usability, reaching, on average, 12 %FP and 18 %FN.[19]

Different to other programming languages, there is no significant difference of DSR and NER and it prevents from determining the most effective attack (Table IX). Thus, all attacks affect the system to the same level.

[17]Computed from raw data located in our companion Zenodo repository.

[18]Also computed based on data from our Zenodo repository

[19]Computed using raw data from our Zenodo repository.

Table VI
C/C++ POISON-FREE RESULTS

SVM	$Mean\_acc$	%FN	%FP
CWE	min/max	min/max	min/max
22	0.71	15.61	13.34
94	0.71	13.63	15.79
120	0.64	20.55	16.16
189	0.61	10.72	27.84
269	0.7	17.99	12.89
287	0.69	13.41	17.77
295	0.77	13.63	9.81
310	0.74	10.14	15.92
369	0.72	13.47	14.97
401	0.69	16.77	15.41
617	0.7	13.42	16.71
770	0.71	15.78	12.98
772	0.78	9.51	13.26
835	0.7	13.32	16.61

KNN	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
22	0.64	0.67	8.99	10.5	23.43	24.11
94	0.65		12.28	15.3	19.42	22.69
120	0.59	0.61	13.82	14.96	24.59	25.65
189	0.6	0.61	19.24	22.59	16.65	20.98
269	0.65	0.67	16.09	17.37	16.28	19.77
287	0.58	0.64	4.97	9.06	26.53	36.35
295	0.5	0.6	0.68	3.12	35.99	46.46
310	0.64	0.69	9.06	10.2	20.58	26.87
369	0.59	0.66	7.15	9.17	24.94	32.56
401	0.56	0.59	2.56	8.35	30.95	39.81
617	0.63	0.66	12.64	13.66	21.71	22.51
770	0.55	0.57	2.1	8.12	33.07	41.09
772	0.66	0.69	1.07	3.54	25.61	30.5
835	0.52	0.59	2.96	7.75	32.06	42.9

RF	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
22	0.63	0.72	12.88	16.36	14.6	20.41
94	0.64	0.75	13.33	17.13	11.82	19.23
120	0.62	0.72	15.84	18.11	13.26	19.97
189	0.59	0.7	12.46	17.47	16.99	23.46
269	0.62	0.74	14.7	16.29	11.71	21.43
287	0.63	0.7	15.04	17.01	15.21	20.11
295	0.68	0.78	14.67	16.1	8.34	16.7
310	0.7	0.79	9.48	13.43	11.14	16.27
369	0.69	0.77	11.53	13.74	12.28	17.45
401	0.64	0.71	15.55	17.79	14.14	19.06
617	0.67	0.74	12.57	14.76	12.74	18.39
770	0.66	0.77	11.3	16.39	12.06	18.41
772	0.72	0.8	7.21	12.39	12.51	14.73
835	0.63	0.72	14.01	17.5	13.6	20.01

MLP	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
22	0.62	0.65	15.93	17.41	18.72	20.3
94	0.69	0.72	15.63	17.61	12.41	15.01
120	0.55	0.6	15.98	20.92	19.89	28.4
189	0.56	0.59	11.82	15.21	27.86	29.31
269	0.64	0.68	14.46	16.66	17.1	20.38
287	0.64	0.65	17.99	19.5	15.73	17.79
295	0.66	0.71	15.26	17.5	13.47	16.92
310	0.71	0.74	11.02	13.37	15.04	16.22
369	0.64	0.67	14.88	16.43	18.05	20.78
401	0.62	0.64	22.09	28.15	11.16	14.97
617	0.67	0.68	12.1	13.28	18.91	20.04
770	0.69	0.72	13.72	15.63	14.32	16.18
772	0.66	0.75	11.3	15.26	13.38	18.67
835	0.61	0.64	17.89	19.54	17.48	19.74

Table VII
C# AND PHP POISON-FREE RESULTS

C#

SVM	$Mean\_acc$	%FN	%FP
CWE	min/max	min/max	min/max
22	0.99	0.46	0.37
89	0.98	1.63	0.47
78	0.98	0.89	0.25
91	0.99	0.7	0.28
90	0.99	0.44	0.3

KNN	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
22	0.99		0.53	1.05	0.2	0.28
89	0.97	0.98	1.21	1.83	0.86	1
78	0.98		0.76	1.38	0.24	0.35
91	0.98		0.82	1.7	0.39	0.48
90	0.97	0.98	1.03	1.64	0.8	1.13

RF	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
22	0.97	0.99	0.81	0.88	0.47	2.39
89	0.98	0.99	0.73	0.87	0.29	1.12
78	0.98	0.99	0.59	1.12	0.06	0.62
91	0.98	1	0.25	0.96	0.06	1.09
90	0.98	0.99	0.7	1.04	0.11	0.75

MLP	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
22	0.95	0.98	0.46	0.66	1.46	3.93
78	0.95	0.98	0.76	1.81	1.24	2.88
90	0.97	0.99	0.37	0.82	0.59	2.65
89	0.92	0.96	1.49	2.61	2.18	6
91	0.96	0.98	0.62	1.18	1.07	2.9

PHP

SVM	$Mean\_acc$	%FN	%FP
CWE	min/max	min/max	min/max
78	0.82	6.75	10.81
79	0.93	4.51	1.8
89	0.94	1.14	4.54
90	0.88	4.63	6.91
91	0.85	10.67	4.05
95	0.87	5.07	7.91
98	0.85	9.44	5.08
601	0.92	3.19	4.97

KNN	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
78	0.8	0.87	9.09	16.9	2.77	3.94
79	0.97		1.58	1.97	1	1.16
89	0.97	0.98	0.08	0.16	2.17	2.68
90	0.89	0.93	4.6	8.12	2.72	3.07
91	0.84	0.89	5.78	11.29	4.71	5.26
95	0.81	0.89	8.06	13.38	3.48	5.59
98	0.82	0.9	7.34	14.89	3.03	3.22
601	0.91	0.94	4.19	5.67	2.35	3.11

RF	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
78	0.85	0.89	3.77	4.62	7.89	10.2
79	0.95	0.98	2.28	3.75	0.47	0.72
89	0.96	0.98	0.04	0.26	2.15	3.41
90	0.84	0.85	0.4	1.01	14.53	14.78
91	0.85	0.88	1.98	3.03	10.36	11.65
95	0.89	0.92	3.67	4.75	4.2	7.1
98	0.88	0.9	3.4	4.3	6.39	7.58
601	0.89	0.9	1.35	1.98	9.03	9.43

MLP	$Mean\_acc$		%FN		%FP	
CWE	min	max	min	max	min	max
78	0.71	0.78	3.63	9.69	12.56	24.89
89	0.76	0.92	1.41	2.62	6.19	22
90	0.59	0.8	0.75	5.7	15.38	39.66
91	0.71	0.75	3.45	8.66	16.3	24.8
98	0.66	0.78	8.88	14.16	11.94	20.38
601	0.7	0.79	2.29	5.73	15.96	27.68

Table VIII
C/C++ - BENCHMARK DETECTOR PER POISON ATTACK

					LF_a						FR_a						DI_a					
Alg.	%p			closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
SVM	20			0	0	0	0	100	100	0	66.67	0	0	33.33	0	33.33	66.67	83.33	0	0	0	16.67	16.67	83.33
	35			25	0	0	0	75	75	25	25	0	41.67	0	33.33	75	25	58.33	0	16.67	16.67	8.33	41.67	58.33
	50			0	0	91.67	0	8.33	100	0	0	0	100	0	0	100	0	0	0	50	50	0	100	0
Alg.	%p	K		closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
KNN	20	9	3	0	50	0	0	50	50	50	100	0	0	0	0	0	0	50	0	0	50	0	50	50
	35			0	0	100	0	0	100	0	0	0	0	100	0	100	0	50	0	0	50	0	50	50
	50			0	0	0	0	50	100	0	0	0	0	100	0	100	0	0	0	0	100	0	100	0
	20			0	0	0	0	100	100	0	0	0	0	100	0	100	0	100	0	0	0	0	0	100
	35			0	0	100	0	0	100	0	0	0	0	100	0	100	0	100	0	0	0	0	0	100
	50			0	0	0	0	100	100	0	0	0	0	100	0	100	0	100	0	0	0	0	100	0
	20			0	0	0	0	100	100	0	0	0	0	100	0	100	0	100	0	0	0	0	100	0
	35			0	0	100	0	0	100	0	0	0	0	100	0	100	0	100	0	0	0	0	100	0
	50			0	0	0	0	100	100	0	0	0	0	100	0	100	0	100	0	0	0	0	100	0
Alg.	%p	ne		closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
RF	20	100	500	0	0	0	0	100	100	0	85.71	0	7.14	7.14	0	14.28	85.71	85.71	0	0	14.29	0	14.29	85.71
	35			0	0	35.71	0	64.29	100	0	57.14	0	28.57	7.14	7.14	42.85	57.14	0	0	0	100	0	100	0
	50			0	0	100	0	0	100	0	0	0	78.57	21.43	0	100	0	0	0	0	100	0	100	0
	20			0	0	0	0	100	100	0	78.57	0	7.14	7.14	7.14	21.42	78.57	92.86	0	7.14	0	0	7.14	92.86
	35			0	0	50	0	50	100	0	57.14	0	21.43	7.14	14.29	42.86	57.14	0	0	0	100	0	100	0
	50			0	0	100	0	0	100	0	0	0	71.43	28.57	0	100	0	0	0	0	100	0	100	0
Alg.	%p	#HL	#N	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
MLP	20	2	50	0	0	0	0	100	100	0	25	0	50	0	25	75	25	50	0	0	25	25	50	50
	35			25	0	25	0	50	75	25	0	0	75	25	0	100	0	0	0	75	25	0	100	0
	50			0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	0	100	0	100	0
	20			0	0	0	0	100	100	0	40	0	40	0	20	60	40	40	0	0	20	40	60	40
	35			20	0	0	0	80	80	20	0	0	60	20	20	100	0	20	0	60	20	0	80	20
	50			0	0	0	0	20	100	0	0	0	80	0	20	100	0	0	0	0	100	0	100	0
	20			0	0	0	0	100	100	0	50	0	50	0	0	50	50	0	0	50	0	50	100	0
	35			0	0	25	0	75	100	0	25	0	25	25	25	75	25	0	0	75	25	0	100	0
	50			0	0	100	0	0	100	0	0	0	75	0	25	100	0	0	0	0	100	0	100	0
	20			0	0	0	0	100	100	0	0	0	0	0	0	0	0	40	0	0	20	40	60	40
	35			0	0	20	0	80	100	0	0	0	0	0	0	0	0	20	0	60	20	0	80	20
	50			0	0	80	0	20	100	0	0	0	0	0	0	0	0	0	0	0	100	0	100	0
	20			0	0	0	0	100	100	0	25	0	50	0	25	75	25	0	0	50	0	50	100	0
	35			25	0	0	0	75	75	25	0	0	75	25	0	100	0	0	0	75	25	0	100	0
	50			0	0	100	0	0	100	0	0	0	75	0	25	100	0	0	0	0	100	0	100	0

Table IX
C# - BENCHMARK DETECTOR PER POISON ATTACK

				LF_a						FR_a						DI_a					
Alg.	%p		closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
SVM	20		100	0	0	0	0	0	100	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	35		100	0	0	0	0	0	100	100	0	0	0	0	0	100	40	0	60	0	0	60	40
	50		0	0	100	0	0	100	0	0	0	80	20	0	100	0	0	0	100	0	0	100	0
Alg.	%p	K	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
KNN	20	3	0	0	100	0	0	100	0	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	35		0	0	60	0	40	100	0	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	50		0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
	20		100	0	0	0	0	0	100	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	35	9	0	0	60	0	40	100	0	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	50		0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
	20		100	0	0	0	0	0	100	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	35	15	20	0	40	0	40	80	20	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	50		0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
Alg.	%p	ne	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
RF	20	5	0	0	40	0	60	100	0	***0***	***0***	***20***	***0***	***80***	***100***	***0***	0	0	40	0	60	100	0
	35		0	0	20	0	80	100	0	***20***	***0***	***0***	***0***	***80***	***80***	***20***	0	0	20	0	80	100	0
	50		***0***	***0***	***100***	***0***	***0***	***100***	***0***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	0	0	100	0	0	100	0
	20	50	60	0	0	0	40	40	60	20	0	0	0	80	80	20	40	0	20	0	40	60	40
	35		0	0	0	0	0	100	0	0	0	0	0	0	60	0	40	0	40	0	60	40	20
	50		***0***	***0***	***100***	***0***	***0***	***100***	***0***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	0	0	100	0	0	100	0
	20	1	0	0	100	0	0	100	0	40	0	20	0	20	60	40	40	0	60	0	20	100	0
	35		0	0	80	0	20	100	0	40	0	20	0	40	60	40	0	0	80	0	20	100	0
	50	150	0	0	100	0	0	100	0	0	0	80	20	0	100	0	0	0	100	0	0	100	0
	20		80	0	20	0	0	20	80	60	0	20	0	20	40	60	0	0	60	0	40	100	0
	35		20	0	60	0	20	80	20	40	0	20	0	40	60	40	0	0	40	0	40	80	0
	50		0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
	20		0	0	80	0	20	100	0	40	0	20	0	40	60	40	20	0	20	0	60	80	20
	35	100	0	0	***60***	***0***	***40***	100	0	40	0	40	0	60	100	40	***0***	***0***	***60***	***0***	***40***	***100***	***0***
	50		0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
	20		0	0	60	0	40	100	0	40	0	20	0	40	60	40	20	0	60	0	20	80	20
	35	2	0	0	80	0	20	100	0	60	0	20	0	40	60	40	***0***	***0***	***60***	***0***	***40***	***100***	***0***
	50	150	0	0	100	0	0	100	0	0	0	80	20	0	100	0	0	0	100	0	0	100	0
	20		0	0	60	0	40	100	0	40	0	60	0	0	0	40	0	0	80	0	20	100	0
	35		***0***	***0***	***20***	***0***	***80***	***100***	***0***	60	0	20	0	20	40	60	0	0	20	0	80	100	0
	50	50	0	0	100	0	0	100	0	0	0	80	20	0	100	0	0	0	100	0	0	100	0
	20		***0***	***0***	***100***	***0***	***0***	***100***	***0***	40	0	40	0	20	60	40	20	0	60	0	20	80	20
	35		***0***	***0***	***60***	***0***	***40***	***100***	***0***	40	0	20	0	40	60	40	0	0	40	0	60	100	0
MLP	50	100	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
	20		0	0	100	0	0	100	0	40	0	40	0	20	60	40	20	0	60	0	20	80	20
	35		***0***	***0***	***60***	***0***	***40***	***100***	***0***	40	0	20	0	40	60	40	0	0	40	0	60	100	0
	50		0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
	20		0	0	80	0	20	100	0	40	0	20	0	40	60	40	20	0	80	0	20	80	20
	35		0	0	60	0	40	100	0	40	0	20	0	40	60	40	0	0	40	0	60	100	0
	50	150	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0

Table X
PHP - BENCHMARK DETECTOR PER POISON ATTACK

				LF_a						FR_a						DI_a					
Alg.	%p			closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
SVM	20			87.5	0	0	0	12.5	12.5	87.5	75	0	25	0	0	0	25	62.5	0	0	12.5	25	37.5	62.5
	35			37.5	0	25	0	37.5	62.5	37.5	50	0	25	12.5	12.5	50	50	37.5	0	0	50	12.5	62.5	37.5
	50			0	0	87.5	12.5	0	100	0	0	0	100	0	0	100	0	0	0	25	75	0	100	0
Alg.	%p	K		closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
KNN	20	3		50	0	37.5	0	12.5	50	50	75	0	12.5	0	12.5	25	75	75	0	12.5	0	12.5	25	75
	35				**0**	**87.5**	**0**	**12.5**	**100**	**0**		0	87.5		0	12.5	100	0	0	25	62.5	12.5	100	0
	50				**0**	**100**	**0**	**0**	**100**	**0**		**0**	**100**		**0**	**0**	**100**		0	62.5	37.5	0	100	0
	20			100	0	0	0	0	0	100	62.5	0	25	0	12.5	37.5	62.5	87.5	0	0	0	12.5	12.5	87.5
	35		9	0	0	87.5	0	12.5	100	0	12.5	0	75	0	12.5	87.5	12.5		**0**	**0**	**87.5**	**12.5**	**100**	**0**
	50			0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	37.5	62.5	0	100	0
	20			100	0	0	0	0	0	100	62.5	0	25	0	12.5	37.5	62.5	87.5	0	0	12.5	12.5	12.5	87.5
	35			0	0	75	0	25	100	0	50	0	37.5	0	12.5	50	50	0	0	0	87.5	12.5	100	0
	50		15	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	0	37.5	0	100	0
Alg.	%p	ne		closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
RF	20	50		25	**0**	**62.5**	**0**	**12.5**	**75**	**25**	**0**	**0**	**75**	**0**	**25**	**100**	**0**	**50**	**0**	**37.5**	**0**	**12.5**	**50**	**50**
	35	5		0	0	75	0	25	100	0	**0**	**0**	**87.5**	**0**	12.5	**100**	**0**	50	0	25	12.5	12.5	50	50
	50			0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	25	75	0	100	0
	20			62.5	0	25	0	12.5	37.5	62.5	12.5	0	62.5	0	25	87.5	12.5	50	0	37.5	0	12.5	50	50
	35			0	0	62.5	0	37.5	100	0	25	0	50	0	25	75	25	50	0	12.5	25	12.5	50	50
	50			0	0	100	0	0	100	0	**0**	**0**	**100**	**0**	**0**	**100**	**0**	0	0	12.5	87.5	0	100	0
	20			62.5	0	25	0	12.5	37.5	62.5	12.5	0	62.5	0	25	87.5	12.5	50	0	37.5	0	12.5	50	50
	35			0	0	62.5	0	37.5	100	0	25	0	50	0	25	75	25	50	0	12.5	25	12.5	50	50
	50	100		0	0	100	0	0	100	0	**0**	**0**	**100**	**0**	**0**	**100**	**0**	0	0	12.5	87.5	0	100	0
	20			62.5	0	25	0	12.5	37.5	62.5	12.5	0	62.5	0	25	87.5	12.5	50	0	37.5	0	12.5	50	50
	35			0	0	50	0	50	100	0	25	0	50	0	25	75	25	50	0	12.5	37.5	12.5	50	50
	50	500		0	0	100	0	0	100	0	**0**	**0**	**100**	**0**	**0**	**100**	**0**	**0**	**0**	**12.5**	**87.5**	**0**	100	0
Alg.	%p	#HL	#N	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER	closeB	Uatt	Satt	HighS	HighU	DSR	NER
MLP	20	2	5	50	0	25	12.5	12.5	50	50	37.5	0	0	50	12.5	62.5	37.5	25	0	25	12.5	37.5	75	25
	35			25	0	62.5	0	12.5	75	25	62.5	0	0	25	12.5	37.5	62.5	62.5	0	0	25	12.5	37.5	62.5
	50			0	0	87.5	12.5	0	100	0	0	0	62.5	37.5	0	100	0	0	0	12.5	87.5	0	100	0
	20			50	0	12.5	25	12.5	50	50	62.5	0	12.5	12.5	12.5	37.5	62.5	12.5	0	25	12.5	50	87.5	12.5
	35			50	0	37.5	12.5	0	50	50	62.5	0	12.5	25	0	37.5	62.5	37.5	0	12.5	37.5	12.5	62.5	37.5
	50	3	5	0	0	87.5	12.5	0	100	0	0	0	25	62.5	12.5	100	0	0	0	25	75	0	100	0

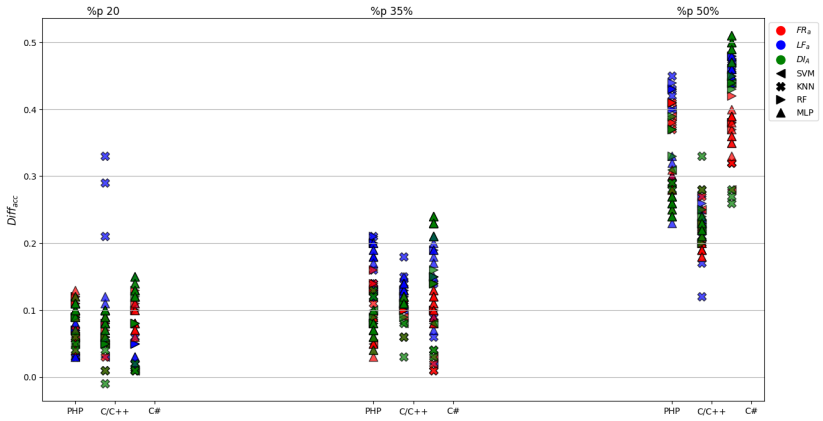

Figure 3. $Diff_{acc}$ per algorithm, language and $\%p$

Takeaway 1

C/C++:

- SVM and KNN the most resilient against LF_a; RF against FR_a; and KNN against DI_a
- LF_a the most dangerous attack
- FR_a and DI_a higher impact on usability than security

C#:

- KNN and SVM the best to protect the system
- LF_a and DI_a the most effective attacks
- All attacks have higher impact on security than in usability, but specially in LF_a and DI_a

PHP:

- MLP the best for protection and RF the worst
- LF_a and FR_a higher impact on security than usability, except for MLP
- No clear which attack affects the most

Common in all languages:

- For $\%p$=50 no way to protect the system

2) CWE-level analysis: Tables XI and XII depict results per CWE, also considering the proposed metrics and DSR and NER for each $\%p$, while Figure 4 presents $Diff_{acc}$.

a) C/C++: The overall system performance is affected by attacks, though it is noticed that $Diff_{acc}$=0.15 for $\%p$=50 in CWE369 and LF_a, which is a small value comparing to the rest. Moreover, it is remarkable that NER is 0 in almost all LF_a configurations, just CWE401 gets NER=33.33 and 4.76 for $\%p$=35 and 50 respectively. Another issue is that CWE287, CWE369, CWE401 and CWE617 are less affected by attacks than other CWEs when $\%p$=20 and 35 in DI_a.

b) C#: According to $Diff_{acc}$ CWE22 is quite resilient to DI_a even for $\%p$=50, that is $Diff_{acc}$=0.17, while in other like CWE90 in DI_a $Diff_{acc}$=0.63, a big number in comparison. The system is quite resilient in CWE91, CWE78 and CWE22 against FR_a for $\%p$=20 and 35 because NER is up to 70 in all cases.

c) PHP: CWE89 is the most affected CWE in all attacks, either considering $Diff_{acc}$, DSR and NER. On the contrary, CWE98 in all attacks and CWE90 and CWE78 in LF_a specially, for $\%p$=20 and 35, are quite resistant. In these cases NER is close or higher than 30 even for $\%p$=35.

d) Comparing CWEs: There are four CWEs, all of them linked to input data controls (recall Section II-A), which are simultaneously considered in different programming languages. On the one hand, CWE90, CWE78 and CWE89 are studied in C# and PHP. In CWE90 and CWE78 similar results are achieved in both languages. Just in CWE78 $\%p$=20 and LF_a results differ, as NER is 36.67 in C# and 81.25 in PHP. By contrast, in C# results of CWE89 show that based on NER, the system is less affected by attacks, though $Diff_{acc}$ is comparable in both languages. The difference is specially remarkable in DI_a, where NER is 0 in PHP for all $\%p$, while NER=61.67 and 56.67 for $\%p$=20 and 35 respectively.

On the other hand, CWE22 is analysed in C# and C/C++. Results of $Diff_{acc}$ are higher in C# in LF_a and FR_a than in C/C++, which means that in C# the overall performance of the system is more affected by these attacks. Similar results are achieved in terms of NER among both languages, being just noticed that in LF_a and $\%p$=35 NER=21.67 in C# and 0 in C/C++, thus the attack is more effective in C/C++ in this configuration.

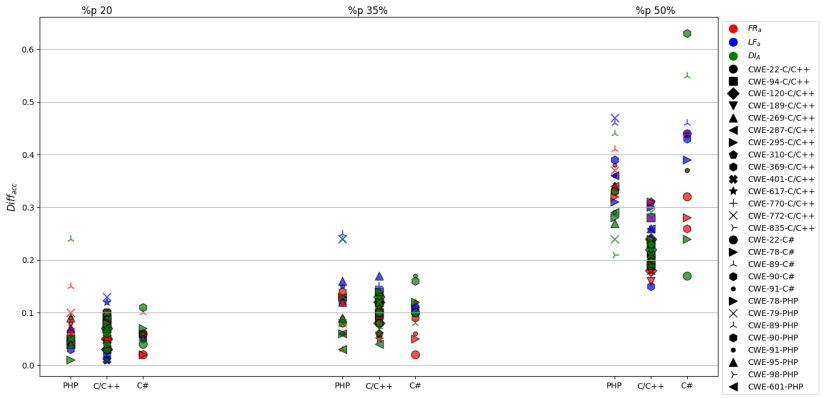

Figure 4. $Diff_{acc}$ per CWE, language and $\%p$

Table XI
PHP AND C# - CWE LEVEL RESULTS

PHP		LF_a							FR_a							DI_a						
CWE	$\%p$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
78	20	81.25	0	14.58	0	4.17	18.75	81.25	66.67	0	6.25	8.33	18.75	33.33	66.67	60.42	0	0	37.5	2.08	39.58	60.42
	35	27.08	0	70.83	0	2.08	72.91	27.08	8.33	0	54.17	35.42	2.08	91.67	8.33	16.67	0	0	83.33	0	83.33	16.67
	50	0	0	62.5	37.5	0	100	0	0	0	77.08	22.92	0	100	0	0	0	0	100	0	100	0
79	20	71.97	0	17.73	0	10.3	28.03	71.97	36.82	0	54.55	1.82	6.82	63.19	36.82	80	0	16.36	0	3.64	20	80
	35	0	0	78.18	0	21.82	100	0	50.3	0	1.82	0	47.88	49.7	50.3	20	0	23.03	56.97	0	80	20
	50	0	0	100	0	0	100	0	0	0	98.18	0	1.82	100	0	0	0	76.36	21.82	1.82	100	0
89	20	50	20	1.67	6.67	21.67	30.01	70	1.67	0	25	0	63.33	88.33	1.67	0	0	3.33	0	96.67	100	0
	35	0	0	11.67	0	88.33	100	0	58.33	0	26.67	15	0	41.67	58.33	0	0	6.67	0	83.33	90	0
	50	0	0	98.33	1.67	0	100	0	0	0	96.67	3.33	0	100	0	0	0	0	100	0	100	0
90	20	93.33	2.22	0	0	4.44	4.44	95.55	46.67	0	22.22	11.11	20	53.33	46.67	51.11	0	20	4.44	24.44	48.88	51.11
	35	31.11	0	61.67	2.22	5	68.89	31.11	13.33	0	80	6.67	0	86.67	13.33	31.11	2.22	40	26.67	0	66.67	33.33
	50	0	0	93.33	6.67	0	100	0	0	0	86.67	13.33	0	100	0	0	0	40	60	0	100	0
91	20	72.92	0	8.33	0	18.75	27.08	72.92	56.25	0	29.17	0	14.58	43.75	56.25	41.67	0	41.66	0	16.67	58.33	41.67
	35	0	0	70.84	2.08	27.08	100	0	22.92	0	22.92	20.83	33.33	77.08	22.92	62.5	0	0	37.5	0	37.5	62.5
	50	0	0	95.83	4.17	0	100	0	0	0	85.42	14.58	0	100	0	0	0	25	75	0	100	0
95	20	36.36	0	38.64	0	25	63.64	36.36	49.24	0	42.24	0	8.52	50.76	49.24	63.64	0	25	11.36	0	36.36	63.64
	35	2.27	0	72.73	0	25	97.73	2.27	13.64	0	77.27	9.09	0	86.36	13.64	4.54	0	14.58	80.87	0	95.45	4.54
	50	0	0	100	0	0	100	0	0	0	79.55	20.45	0	100	0	0	0	14.58	85.42	0	100	0
98	20	55.56	0	41.66	2.78	0	44.44	55.56	66.67	0	25	5.55	2.78	33.33	66.67	77.78	0	0	0	22.22	22.22	77.78
	35	33.33	0	52.78	0	13.89	66.67	33.33	36.11	0	31.25	0	32.64	63.89	36.11	47.22	0	0	52.78	0	52.78	47.22
	50	0	0	94.44	5.55	0	99.99	0	0	0	97.22	2.78	0	100	0	0	0	38.89	61.11	0	100	0
601	20	85	0	8.33	5	1.67	15	85	48.33	1.67	20	5	25	50	50	50	0	0	0	50	50	50
	35	3.33	0	40	0	56.67	96.67	3.33	56.67	0	20	3.33	20	43.33	56.67	28.33	0	20	51.67	0	71.67	28.33
	50	0	0	96.67	3.33	0	100	0	0	0	96.67	3.33	0	100	0	0	0	20	80	0	100	0
C#		LF_a							FR_a							DI_a						
CWE	$\%p$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
22	20	38.33	0	35	0	26.67	61.67	38.33	75	0	5	0	20	25	75	66.67	0	18.33	0	15	33.33	66.67
	35	21.67	0	0	0	78.33	78.33	21.67	93.33	0	3.33	0	3.33	6.66	93.33	40	0	21.67	0	38.33	60	40
	50	0	0	100	0	0	100	0	0	0	100	0	0	100	0	1.67	0	98.33	0	0	98.33	1.67
78	20	36.67	0	23.33	0	40	63.33	36.67	80	0	0	0	20	20	80	45	0	23.33	0	31.67	55	45
	35	20	0	51.67	0	28.33	80	20	75	0	20	0	5	25	75	20	0	55	0	25	80	20
	50	0	0	100	0	0	100	0	0	0	93.33	6.67	0	100	0	0	0	100	0	0	100	0
89	20	73.33	0	26.67	0	0	26.67	73.33	60	0	3.33	0	36.67	40	60	61.67	0	38.33	0	0	38.33	61.67
	35	25	0	26.67	0	48.33	75	25	40	0	33.33	0	26.67	60	40	56.67	0	43.33	0	0	43.33	56.67
	50	0	0	100	0	0	100	0	0	0	78.33	21.67	0	100	0	0	0	100	0	0	100	0
90	20	73.33	0	26.67	0	0	26.67	73.33	40	0	11.67	0	48.33	60	40	40	0	33.33	0	26.67	60	40
	35	28.33	0	18.33	0	53.33	71.66	28.33	60	0	3.33	0	36.67	40	60	40	0	8.33	0	51.67	60	40
	50	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0
91	20	73.33	0	16.67	0	10	26.67	73.33	75	0	15	0	10	25	75	60	0	18.33	0	21.67	40	60
	35	20	0	61.67	0	18.33	80	20	78.33	0	1.67	0	20	21.67	78.33	20	0	43.33	0	36.67	80	20
	50	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	100	0	0	100	0

Takeaway 2

C/C++

- CWE369 no highly affected in LF_a
- Poisoning attacks affect CWE287, CWE369, CWE401 and CWE617 less than other CWEs

C#

- CWE22 quite resilient against DI_a even for $\%p$=50
- CWE91, CWE78 and CWE22 quite resilient against FR_a with $\%p$=20 and 35

PHP

- CWE89 the most affected regardless of the attack
- CWE98 quite resistant to attacks, also CWE90 and CWE78 in LF_a for $\%p$=20 and 35

Common to in C# and PHP:

- CWE90 and CWE78 (input data controls) similar results

E. Statistical significance analysis

PERMANOVA statistical test [55] is performed to ensure the statistical significance of the differences between models. This is a non-parametric test used to compare the differences between groups based on distance matrices. It assesses whether the centroids (multivariate means) of different groups are significantly different in a high-dimensional space. The result is p-value, where 0.05 is the most common threshold to reject or accept the null hypothesis (H_0) which states that differences between groups are random. If p-value$\leqslant$0.05 means that independent variables have a meaningful effect on the dataset, while if p-value$>$0.05 means that differences might be due to random variation.

This test was selected over other alternatives because it is suitable for multiple dependent and independent variables and does not require normality for any of the variables. Indeed, dependent variables do not follow a normal distribution.

In particular, $\%p$ and algorithms are considered independent

Table XII
C/C++ - CWE LEVEL RESULTS

		LF_a							FR_a							DI_a						
CWE	%p	*closeB*	*Uatt*	*Satt*	*HighS*	*HighU*	*DSR*	*NER*	*closeB*	*Uatt*	*Satt*	*HighS*	*HighU*	*DSR*	*NER*	*closeB*	*Uatt*	*Satt*	*HighS*	*HighU*	*DSR*	*NER*
	20	0	33.33	0	0	66.67	66.67	33.33	100	0	0	0	0	0	100	66.67	0	0	0	33.33	33.33	66.67
	35	0	0	66.67	0	33.33	100	0	33.33	0	33.33	33.33	0	66.66	33.33	33.33	0	0	66.67	0	66.67	33.33
22	50	0	0	66.67	0	33.33	100	0	0	0	66.67	33.33	0	100	0	0	0	33.33	66.67	0	100	0
	20	0	0	0	0	100	100	0	38.89	0	58.33	0	2.78	61.11	38.89	55.56	0	25	19.44	0	44.44	55.56
	35	0	0	0	0	100	100	0	25	0	75	0	0	75	25	0	0	0	100	0	100	0
94	50	0	0	100	0	0	100	0	0	0	77.78	22.22	0	100	0	0	0	33.33	66.67	0	100	0
	20	0	0	0	0	100	100	0	100	0	0	0	0	0	100	33.33	0	0	66.67	0	66.67	33.33
	35	0	0	0	0	100	100	0	66.67	0	0	0	33.33	33.33	66.67	0	0	0	100	0	100	0
120	50	0	0	100	0	0	100	0	0	0	66.67	33.33	0	100	0	0	0	0	100	0	100	0
	20	0	0	0	0	100	100	0	100	0	0	0	0	0	100	100	0	0	0	0	0	100
	35	0	0	100	0	0	100	0	0	0	0	100	0	100	0	0	0	0	100	0	100	0
189	50	0	0	100	0	0	100	0	0	0	0	100	0	100	0	0	0	0	100	0	100	0
	20	0	0	0	0	100	100	0	66.66	0	0	0	33.34	33.34	66.66	50	0	0	0	50	50	50
	35	0	0	0	0	100	100	0	0	0	100	0	0	100	0	0	0	50	50	0	100	0
269	50	0	0	100	0	0	100	0	0	0	66.66	33.34	0	100	0	0	0	0	100	0	100	0
	20	0	0	0	0	100	100	0	83.34	0	16.66	0	0	16.66	83.34	83.34	0	16.66	0	0	16.66	83.34
	35	0	0	16.66	0	83.34	100	0	0	0	100	0	0	100	0	50	0	0	50	0	50	50
287	50	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	50	50	0	100	0
	20	0	0	0	0	100	100	0	88.89	0	11.11	0	0	11.11	88.89	50	0	33.33	16.67	0	50	50
	35	0	0	0	0	100	100	0	16.67	0	38.89	0	44.44	83.33	16.67	0	0	30.56	25	44.44	100	0
295	50	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	44.44	55.56	0	100	0
	20	0	0	0	0	100	100	0	30.56	0	36.11	33.33	0	69.44	30.56	88.89	0	11.11	0	0	11.11	88.89
	35	55.56	0	44.44	0	0	44.44	55.56	63.89	0	19.44	16.67	0	36.11	63.89	41.67	0	25	33.33	0	58.33	41.67
310	50	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	0	100	0	100	0
	20	0	0	0	0	100	100	0	58.33	0	41.67	0	0	41.67	58.33	100	0	0	0	0	0	100
	35	0	0	0	0	100	100	0	8.33	0	41.67	0	50	91.67	8.33	66.67	0	0	33.33	0	33.33	66.67
369	50	0	0	66.67	33.33	0	100	0	0	0	100	0	0	100	0	0	0	0	100	0	100	0
	20	0	0	0	0	100	100	0	90.48	0	0	0	9.52	9.52	90.48	66.67	0	0	0	33.33	33.33	66.67
	35	33.33	0	33.33	0	33.33	66.66	33.33	22.22	0	11.11	0	66.67	77.78	22.22	57.14	0	9.52	33.33	0	42.85	57.14
401	50	4.76	0	76.19	0	19.05	95.24	4.76	0	0	55.56	11.11	33.33	100	0	0	0	0	100	0	100	0
	20	0	0	0	0	100	100	0	55.56	0	0	33.33	11.11	44.44	55.56	100	0	0	0	0	0	100
	35	33.33	0	55.56	0	11.11	66.67	33.33	66.67	0	0	33.33	0	33.33	66.67	55.56	0	11.11	33.33	0	44.44	55.56
617	50	0	0	100	0	0	100	0	0	0	55.56	44.44	0	100	0	0	0	0	100	0	100	0
	20	0	0	0	0	100	100	0	50	0	16.67	33.33	0	50	50	66.67	0	0	0	33.33	33.33	66.67
	35	0	0	30.56	0	69.44	100	0	13.89	0	77.78	8.33	0	86.11	13.89	0	0	66.67	33.33	0	100	0
770	50	0	0	66.67	0	33.33	100	0	0	0	100	0	0	100	0	0	0	33.33	66.67	0	100	0
	20	0	0	0	0	100	100	0	45.83	0	11.25	22.92	20	54.17	45.83	80.83	0	2.5	16.67	0	19.17	80.83
	35	0	0	36.25	0	63.75	100	0	25	0	25	25	25	75	25	52.5	0	26.25	18.75	2.5	47.5	52.5
772	50	0	0	91.67	8.33	0	100	0	0	0	75	25	0	100	0	0	0	30	70	0	100	0
	20	0	0	0	0	100	100	0	0	0	0	100	0	100	0	100	0	0	0	0	0	100
	35	0	0	0	0	100	100	0	100	0	0	0	0	0	100	0	0	0	100	0	100	0
835	50	0	0	100	0	0	100	0	0	0	100	0	0	100	0	0	0	0	100	0	100	0

variables and $Diff_{acc}$ and DSR dependent ones. Note that NER could be chosen instead of DSR, as it is complementary. Table XIII shows p-values of all tests. Results show that in any language or attack p-value>0.05, rejecting H_0 and discarding randomness. Appendix C shows some additional tests with different numbers of $\%p$ as independent variables to corroborate the results, which remain consistent.

Table XIII
P-VALUES OF PERMANOVA STATISTICAL TEST.

	LF_a	FR_a	DI_a
C/C++	0.0359	0.0009	0.0009
PHP	0.0009	0.0009	0.0009
C#	0.0019	0.0009	0.0009

F. Analysis of results and goals discussion

In terms of $Diff_{acc}$, in all languages the average for all settings has a similar value showing some system degradation after attacks, though a little higher in C# and PHP. However, to analyse the most effective attack, the amount of times NER is 0 (Tables VIII and IX) has been computed. In this regard, PHP is the language which is less affected by attacks, while C# and C/C++ present comparable results, being LF_a the most effective and FR_a the least one. Results are generally more homogeneous among C/C++ and C#, which is due to the nature of the programming languages. Technically speaking, C++ and C# are derived from C, thus some similarities are expected. Concerning PHP, as code samples are shorter (e.g. average NLOC is 10.63 for PHP, while it is 34.87 and 42.01 for C/C++ and C# respectively), FR_{ttack} and DI_a may affect the system to a lesser extent.

An analysis of over 11 million PHP files on Packagist showed that 36% contain fewer than 100 NLOC, with a median file length of only 4 lines,[20] which is quite aligned to our files. Nonetheless, while our findings suggest that PHP exhibits higher resilience, this may be partially attributed to the shorter and structurally simpler nature of the considered PHP code samples. These characteristics may limit the generalizability of our results to more complex or enterprise-grade PHP systems. We therefore recommend further evaluation on large-scale PHP applications to assess whether the observed robustness trends hold in production environments.

When comparing languages with the same studied CWE, $Diff_{acc}$ is comparable is most cases but NER presents some differences, being meaningful that CWE89 in C# is less affected by attacks than in PHP.

This study, pointed out in Section II-A, considers some CWEs from the Top 25 most common ones, being linked to input data (CWE22, CWE78, CWE89, CWE22 and CWE94) and access management (CWE287) controls. Accordingly, in C#, CWE22 and CWE78 are quite resilient to FR_a in $\%p$=20 and 35 and the same happens for CWE89 in DI_a specially. The situation differs in PHP, where CWE89 can be attacked by FR_a and DI_a even with $\%p$=20. However, CWE78 is quite resistant to LF_a for $\%p$=20 and 35. Similarly, in C/C++, results from CWE22, CWE94 and CWE287 show that even with the smallest $\%p$, attacks are fully successful (DSR=100) in LF_a, except for CWE22 where NER=33.33 for $\%p$=35.

Another important point to note is that the developed attacks are stealthy, involving subtle changes to the original samples to induce poisoning. This is further supported by the results from the detection algorithms.

A final clarification must be made regarding ethical issues, as the results of studies like this are a double-edged sword. On

[20] https://blog.lepine.pro/en/php-ecosystem-deep-dive-code-quality-landscape/, last access July 2025.

the one hand, they could be maliciously used by attackers to refine and enhance their malicious strategies, fine-tuning their approaches to bypass detection more effectively. This raises significant ethical concerns, as it could lead to the development of even more sophisticated and undetectable attacks, putting sensitive systems and data at greater risk. Moreover, the knowledge gained from identifying the more easy-to-attack CWE could also be used to attack systems more precisely. On the other hand, the insights gained from this study provide significant benefits for defenders. They can use this knowledge to develop more robust detection systems and countermeasures capable of withstanding sophisticated poisoning attempts.

Table XIV
TABNET AND TABPFN POISON-FREE RESULTS

	TabNet			TabPFN		
			C#			
CWE	$Mean\_acc$	%FN	%FP	$Mean\_acc$	%FN	%FP
22	0.31	33.36	35.61	0.95	3.63	0.85
78	0.42	37.81	20.49	0.97	2.36	0.19
89	0.94	1.79	3.69	0.97	1.30	1.41
90	0.39	39.90	21.74	0.98	1.63	0.06
91	0.54	13.77	32.53	0.99	0.85	0.08
			PHP			
78	0.55	36.87	7.73	0.97	1.11	1.71
79	0.92	5.36	2.72	0.97	1.82	1.68
89	0.79	3.55	17.86	0.98	1.64	0.62
90	0.81	11.69	6.97	0.98	0.83	0.67
91	0.50	37.22	13.03	0.93	2.42	4.51
95	0.62	31.36	7.29	0.97	1.93	1.48
98	0.54	38.98	7.31	0.96	2.23	1.39
601	0.67	0.91	32.50	0.96	1.15	3.04
			C/C++			
22	0.49	19.22	31.90	0.74	8.59	17.59
94	0.49	16.52	34.51	0.66	22.03	12.19
120	0.43	21.25	35.83	0.67	18.84	14.03
189	0.53	15.04	31.76	0.69	15.76	15.27
269	0.42	20.55	37.31	0.69	12.79	18.24
287	0.43	17.81	39.77	0.61	16.67	22.09
295	0.49	21.41	29.81	0.75	17.07	7.95
310	0.47	19.66	33.33	0.81	9.73	9.35
369	0.45	18.41	36.19	0.77	15.03	7.83
401	0.53	13.95	32.77	0.68	16.63	14.85
617	0.46	18.28	35.88	0.72	12.42	15.74
770	0.48	15.92	35.92	0.80	11.23	8.89
772	0.48	18.88	33.34	0.81	6.19	12.88
835	0.49	17.97	32.90	0.70	12.12	17.60

Table XV
TABNET AND TABPFN BENCHMARK DETECTOR PER POISON ATTACK AND $\%p$ 35

		TabPFN							
		$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
C/C++	LF_a	0.12	0	0	50	0	50	100	0
	DI_a	0.11	0	0	12.50	87.50	0	100	0
	FR_a	0.09	0	0	37.50	50	12.50	100	0
C#	LF_a	0.02	80	0	20	0	0	20	80
	DI_a	0.20	0	0	0	0	100	100	0
	FR_a	0.15	20	0	40	0	60	100	20
PHP	LF_a	0.25	12.50	0	62.50	0	25	87.50	12.50
	DI_a	0.20	12.50	0	62.50	0	25	87.50	12.50
	FR_a	0.22	12.50	0	62.50	0	25	87.50	12.50
					TabNet				
C#	LF_a	0.05	100	0	0	0	0	0	100
	DI_a	0	100	0	0	0	0	0	100
	FR_a	0	100	0	0	0	0	0	100
PHP	LF_a	0.09	0	0	33.33	33.33	33.33	100	0
	DI_a	0.10	33.33	0	0	33.33	33.33	66.67	33.33
	FR_a	0.10	33.33	0	0	33.33	33.33	66.67	33.33

VIII. FUTURE PROSPECTS: THE APPLICABILITY OF NEW MODELS

Although this paper focuses on traditional AI models, given the widespread use of deep learning nowadays, this section presents some preliminary results of the use of a couple of deep learning models and input features for vulnerability detection.

Models such as BERT or LLAMA [56][57][58] have already been used for vulnerability detection. In [59], a highly simplistic comparison is made between traditional AI methods and LLMs for software vulnerability detection. Nevertheless, these approaches use code as input data and classify vulnerabilities on a binary basis, without considering CWEs. In fact, [60] is the only work which proposes the use of LLMs for detecting 13 types of vulnerabilities.

By contrast, this proposal applies code and token features, being unsuitable the use of models trained to learn from text. Thus, tabular models, namely TabNetClassifier[21] (called TabNet) and TabPFN[22] are applied herein. TabPFN uses a transformer-based model, learns from a large amount of prior tabular data and generalizes to new tabular tasks, being quite similar to a LLM. TabNetClassifier is a deep learning method that uses attention mechanisms to select important features dynamically, can be trained end-to-end like a deep neural network and competes with traditional AI models.

Concerning experimental settings, after trial and error, default TabNetClassifier parameters are left, considering the smallest learning rate (i.e. 1e-2) due to the size of the dataset (i.e. small than <10K inputs). Using the library pytorch_tabnet 4.1.0, the remaining experimental settings and environment are analogous to the ones described in Section VII-A. The same applies to TabPFN, except for the use of library tabpfn 2.0.8 and GPU 24 GB NVIDIA L4 in Google Colab because higher computing power is demanded.

The outcomes for poison-free datasets are shown in Table XIV, where performance is lower in C/C++, likely due to the smaller training dataset size. Additionally, it is evident that TabPFN outperforms TabNet in a larger number of CWEs, with particularly strong performance in C# and PHP for both models.

To complement the study and provide some initial results, but without the intention of being exhaustive, we have analysed proposed attacks under $\%p$ 35, presenting computations just with those CWE in which accuracy is 70% or higher (recall Section VI-B). Table XV depicts results. Interestingly, C# in TabNet is not affected by any of the attacks. On the contrary, the affection is significant in the rest of settings, being remarkable in PHP for TabPFN, where $Satt$ is 62.5% in all cases.

In general, results of these models against traditional ones do not differ significantly (compared with Tables VI and VII), i.e. in C/C++ worse outcomes are reached. Nonetheless, this is just a preliminary analysis and much more work is devised as future work. In fact, the field of software vulnerability detection through deep learning models has many challenges to deal with [61], such as the size of the datasets required for training and the demanding time and computing requirements.

IX. CONCLUSION

In light of the number of existing vulnerabilities and the need of their detection, lots of works have been developed

[21] https://pypi.org/project/pytorch-tabnet/, last access July 2025.
[22] https://github.com/PriorLabs/TabPFN, last access July 2025.

in the area. Most of them use machine learning algorithms and despite problems linked to their use, this proposal is the first one to characterize the resilience of poisoning attacks in traditional AI algorithms, also considering simple code features in the detection process. Different CWE of PHP, C# and C/C++ programming languages are considered in this study. Results show that poisoning attacks are stealthy and affect the system but to a different extent depending on algorithms and languages, as long as the poisoning remains under 50%. In general, it is noticed that KNN and SVM are more appropriate for system protection in C# and C/C++, while MLP in PHP.

This study could be extended to other cybersecurity areas which use AI algorithms, for instance, network flow analysis in software-defined networking. In the same way, an ensemble learning approach could be enforced to get a combined output for different CWEs instead of a binary classification. Besides, the use of generative adversarial networks is an attack type to be also developed as a future step. These algorithms have not been considered in current works for vulnerability management and it is a challenge considering the amount of existing programming languages. In the same vein, our preliminary analysis has shown the need to study poisoning attacks using deep learning models with input features. Finally, once highlighted the problem of poisoning attacks in the vulnerability detection field, countermeasures should be devised, apart from those used in literature (i.e. spectral signatures and activation clustering), either to reduce the impact of attacks or to help in the identification of malicious samples. For example, some defences or mitigation strategies against poisoning attacks could be the use of L2 regularization [62] in SVM and MLP to prevent overfitting to poisoned samples; bootstrap aggregation [63] in RF to improve generalization; the use of autoencoders to reconstruct clean inputs and flag deviations in MLP, SVM or RF; or the use of weighted distance voting to assign higher weights to trusted samples and lower weights to new or rare ones in KNN [64].

DECLARATIONS

Ethical Approval and Consent to participate

Authors declare they have no conflict of interest.

Availability of supporting data

Data will be freely available.

Competing interests

The authors declare that they have no competing interests

ACKNOWLEDGEMENTS

This work has been supported by Grant PID2023-150310OB-I00 (MORE4AIO, funded by the Spanish Ministerio de Ciencia, Innovación y Universidades, the Spanish Agencia Estatal de Investigación and FEDER funds. J. Garcia-Alfaro is partially supported by the French National Research Agency under the France 2030 label (NF-HiSec ANR-22-PEFT-0009). Additionally, Lorena Gonzalez has also received support from UC3M's Requalification programme, funded by the Spanish Ministerio de Universidades with EU recovery funds.

A. APPENDIX

Although the metrics used in the analysis are deemed the most suitable for this study, the following metrics will also be made available in our companion Zenodo repository:[23]

- 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 FP.
- 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 (FP), negatives (FN), true positives (TP) and negatives (TN). In this case, as bad samples are labelled with 0 and good ones with 1 (recall Section VII-A), FP are the most relevant because it means that a vulnerability is missed, while FN affect usability.
- Matthews Correlation Coefficient (MCC): is a measure of the quality of binary classification and produces a high score ($+1$ to -1) only if the prediction obtained good results in all of the four confusion matrix categories.

B. APPENDIX

Tables XVI, XVII and XVIII depicts the complete version of results, namely, $Diff_{acc}$, $closeB$, $Uatt$, $Satt$, $HighS$, $HighU$, DSR and NER for each algorithm, parameter and language.

C. APPENDIX

PERMANOVA is computed taking pairs of $\%p$ and algorithms as independent variables and $Diff_{acc}$ and DSR as dependent ones. This analysis contributes to ensure the robustness of results. Table XIX presents results which show that randomness is discarded in all cases except for $\%p$={20,50} and LF_a in C/C++. However, this result does not follow the general trend. Although the p-value is 0.11, which is slightly above the conventional threshold of 0.05, it still suggests that the observed differences may plausibly be due to random variation [65].

REFERENCES

[1] Y. Huang, F. Xu, H. Zhou, X. Chen, X. Zhou, and T. Wang, "Towards exploring the code reuse from stack overflow during software development," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 548–559.

[2] Z. Zhang, H. Ning, F. Shi, F. Farha, Y. Xu, J. Xu, F. Zhang, and K.-K. R. Choo, "Artificial intelligence in cyber security: research advances, challenges, and opportunities," *Artificial Intelligence Review*, pp. 1–25, 2022.

[3] Y. Hu, W. Kuang, Z. Qin, K. Li, J. Zhang, Y. Gao, W. Li, and K. Li, "Artificial intelligence security: Threats and countermeasures," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–36, 2021.

[4] A. E. Cinà, K. Grosse, A. Demontis, S. Vascon, W. Zellinger, B. A. Moser, A. Oprea, B. Biggio, M. Pelillo, and F. Roli, "Wild patterns reloaded: A survey of machine learning security against training data poisoning," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–39, 2023.

[23] https://doi.org/10.5281/zenodo.16922431

Table XVI
C/C++ - BENCHMARK DETECTOR PER POISON ATTACK

				LF_a								FR_a								$D1_a$							
Alg.	%p			$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20			0.03		0	0	0	100	100	0	0.05	66.67	0	0	33.33	0	33.33	66.67	0.05	83.33	0	0	0	16.67	16.67	83.33
	35			**0.1**	**25**	**0**	**0**	**0**	**75**	**75**	**25**	0.09	25	0	41.67	0	33.33	75	25	0.08	58.33	0	16.67	16.67	8.33	41.67	58.33
SVM	50			0.23	0	0	91.67	0	8.33	100	0	0.25	0	0	100	0	0	0	0	0.22	0	0	50	50	0	100	0
Alg.	%p	K		$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20			**0.21**	**0**	**50**	**0**	**0**	**50**	**50**	**50**	0.03	**100**	**0**	**0**	**0**	**0**	**0**	**100**	0.01	50	0	0	0	0	50	50
	35			0.15	0	0	100	0	0	0	0	0.11	0	0	0	100	0	0	0	0.08	50	0	0	50	0	50	50
	50	3		0.21	0	0	50	0	50	100	0	0.2	0	0	0	100	0	100	0	0.22	0	0	0	100	0	100	0
	20			***0.33***	***0***	***0***	***0***	***0***	***100***	***100***	***0***	0.05	0	0	0	100	0	100	0	**0.04**	**100**	**0**	**0**	**0**	0	0	**100**
	35			***0.18***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	0.11	0	0	0	100	0	100	0	0.06	100	0	0	0	0	0	100
	50	9		0.17	0	0	100	0	0	100	0	0.05	0	0	0	100	0	100	0	**0.33**	**0**	**0**	**0**	***100***	***0***	***100***	***0***
	20			0.29	0	0	0	0	100	100	0	0.01	0	0	0	100	0	100	0	-0.01	0	0	0	100	0	100	0
KNN	35			0.14	0	0	100	0	0	100	0	0.06	0	0	0	100	0	100	0	**0.03**	**100**	**0**	**0**	**0**	**0**	**0**	**100**
	50	15		0.12	0	0	0	100	0	100	0	0.27	0	0	0	100	0	100	0	0.28	0	0	0	100	0	0	0
Alg.	%p	ne		$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20			0.08	0	0	0	0	100	100	0	0.08	25	0	0	75	0	75	25	0.05	75	0	0	25	0	25	75
	35			0.12	0	0	25	0	75	100	0	0.1	50	0	25	25	0	50	50	0.09	0	0	50	50	0	100	0
	50	5		0.2	0	0	100	0	0	100	0	0.2	0	0	100	0	0	100	0	0.2	0	0	0	100	0	100	0
	20			0.05	0	0	0	0	100	100	0	0.07	57.14	0	21.43	14.29	7.14	42.86	57.14	0.06	85.71	0	0	14.29	0	14.29	85.71
	35			0.12	0	0	42.86	0	57.14	100	0	0.1	50	0	42.86	7.14	0	50	50	0.11	0	0	0	100	0	100	0
	50	50		0.25	0	0	100	0	0	100	0	0.22	0	0	64.29	35.71	0	100	0	0.23	0	0	0	100	0	100	0
	20			0.05	0	0	0	0	100	100	0	0.06	85.71	0	7.14	7.14	0	14.28	85.71	0.05	85.71	0	0	14.29	0	14.29	85.71
	35			0.13	0	0	35.71	0	64.29	100	0	**0.1**	**57.14**	**0**	**28.57**	**7.14**	**7.14**	**42.85**	**57.14**	0.11	0	0	0	100	0	100	0
	50	100		0.26	0	0	100	0	0	100	0	0.23	0	0	78.57	21.43	0	100	0	0.24	0	0	0	100	0	100	0
	20			0.06	0	0	0	0	100	100	0	0.06	78.57	0	7.14	7.14	7.14	21.42	78.57	0.05	92.86	0	7.14	0	0	7.14	92.86
RF	35			0.13	0	0	50	0	50	100	0	**0.1**	**57.14**	**0**	**21.43**	**7.14**	**14.29**	**42.86**	**57.14**	0.11	0	0	0	100	0	100	0
	50	500		***0.27***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	0.23	0	0	71.43	28.57	0	100	0	0.25	0	0	0	100	0	100	0
Alg.	%p	#N	#HL	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$close.B$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20			0.08	0	0	0	0	100	100	0	0.09	42.86	0	28.57	0	28.57	57.14	42.86	0.07	57.14	0	28.57	0	14.29	42.86	57.14
	35			0.13	14.29	0	42.86	0	42.86	85.72	14.29	0.11	0	0	57.14	28.57	14.29	100	0	0.11	42.86	0	42.86	14.29	0	57.15	42.86
	50	5		0.23	0	0	85.71	14.29	0	100	0	0.21	0	0	100	0	0	100	0	0.23	0	0	28.57	71.43	0	100	0
	20			0.09	0	0	0	0	100	100	0	0.09	16.67	0	50	16.67	0	33.33	83.33	16.67	0.08	50	0	16.67	0	33.33	50
	35			0.13	16.67	0	16.67	0	66.67	83.34	16.67	0.11	16.67	0	50	16.67	0	16.67	83.34	16.67	0.11	16.67	0	16.67	0	83.34	16.67
	50	50		0.23	16.67	0	83.33	0	0	83.33	16.67	0.19	0	0	83.33	0	16.67	100	0	0.22	0	0	0	100	0	100	0
	20			0.08	0	0	0	0	100	100	0	0.1	40	0	40	0	20	60	40	0.1	20	0	0	40	0	80	20
	35			0.13	20	0	0	0	80	80	20	0.12	20	0	40	0	20	80	20	0.22	20	0	40	20	20	80	20
	50	100		0.23	0	0	0	0	20	100	0	0.19	0	0	80	0	20	100	0	0.22	0	0	0	100	0	100	0
	20			0.07	0	0	0	0	100	100	0	0.1	40	0	40	0	20	60	40	0.1	20	0	40	0	80	20	0
	35			0.14	20	0	0	0	80	80	20	0.12	20	0	40	0	20	80	20	0.11	40	0	40	20	0	60	40
	50	150		0.24	0	0	80	0	20	100	0	0.2	0	0	80	0	20	100	0	0.22	0	0	0	100	0	100	0
	20			0.1	0	0	0	0	100	100	0	0.08	50	0	50	0	0	50	50	0.06	50	0	33.33	0	16.67	16.67	50
	35			0.11	16.67	0	33.33	0	50	83.33	16.67	0.11	16.67	0	66.67	16.67	0	83.34	16.67	0.11	33.33	0	33.33	16.67	16.67	66.67	33.33
	50	5		0.24	0	0	100	0	0	100	0	0.2	0	0	83.33	16.67	0	100	0	0.21	0	0	0	100	0	100	0
	20			0.11	0	0	0	0	100	100	0	0.1	25	0	50	0	25	75	25	0.08	50	0	0	25	25	50	50
	35			0.13	25	0	25	0	50	75	25	***0.12***	***0***	***0***	***75***	***25***	***0***	***100***	***0***	0.11	0	0	75	25	0	100	0
	50	50		0.25	0	0	100	0	0	100	0	0.21	0	0	100	0	0	100	0	0.23	0	0	0	100	0	100	0
	20			0.08	0	0	0	0	100	100	0	0.1	40	0	40	0	20	60	40	0.09	40	0	0	20	40	60	40
	35			0.13	20	0	0	0	80	80	20	***0.12***	***0***	***0***	***60***	***20***	***20***	***100***	***0***	0.11	20	0	0	20	0	80	20
	50	100		0.23	0	0	0	0	20	100	0	0.19	0	0	80	0	20	100	0	0.22	0	0	0	100	0	100	0
	20			0.06	0	0	0	0	100	100	0	0.1	25	0	50	0	0	50	50	0.07	50	0	0	25	25	50	50
	35	2		0.15	0	0	25	0	75	100	0	0.11	25	0	25	25	25	75	25	***0.12***	***0***	***0***	***75***	***25***	***0***	***100***	***0***
	50	150		0.24	0	0	100	0	0	100	0	0.18	0	0	75	0	0	100	0	0.21	0	0	0	100	0	100	0
	20			0.09	0	0	0	0	100	100	0	0.08	80	0	20	0	0	0	0	0.08	20	0	60	0	20	80	20
	35			0.11	0	0	40	0	60	100	0	0.11	40	0	60	0	0	60	40	0.11	20	0	60	20	0	80	20
	50	5		0.23	0	0	100	0	0	100	0	0.22	0	0	100	0	0	100	0	0.22	0	0	20	80	0	100	0
	20			0.12	0	0	0	0	100	100	0	0.09	50	0	25	0	25	50	50	0.07	50	0	25	0	25	50	50
	35			0.14	0	0	25	0	75	100	0	0.11	25	0	75	0	25	75	25	0.11	0	0	75	25	0	100	0
	50	50		0.24	0	0	100	0	0	100	0	0.21	0	0	100	0	0	100	0	0.24	0	0	0	100	0	100	0
MLP	20			0.09	0	0	0	0	100	100	0	***0.1***	***0***	***0***	***60***	***0***	***0***	***40***	***100***	***0***	0.09	40	0	0	20	40	60	40
	35			0.14	0	0	20	0	80	100	0	***0.12***	***0***	***0***	***80***	***0***	***0***	***20***	***100***	***0***	0.12	20	0	60	20	0	80	20
	50	100		0.23	0	0	80	0	20	100	0	0.19	0	0	80	0	0	20	100	0	0.23	0	0	0	100	0	100	0
	20			0.06	0	0	0	0	100	100	0	0.1	25	0	50	0	25	75	25	***0.1***	***0***	***0***	***50***	***50***	***0***	***100***	***0***
	35	3		0.14	25	0	0	0	75	75	25	***0.12***	***0***	***0***	***75***	***0***	***25***	***100***	***0***	0.11	0	0	75	25	0	100	0
	50	150		0.21	0	0	100	0	0	100	0	0.18	0	0	75	0	25	100	0	0.22	0	0	0	100	0	100	0

Table XVII
C# - BENCHMARK DETECTOR PER POISON ATTACK

				LF attack								FR attack								DI attack							
Alg.	%p			DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER
	20			**0.01**	**100**	**0**	**0**	**0**	**0**	**0**	**100**	**0.01**	**100**	**0**	**0**	**0**	**0**	**0**	**100**	**0.01**	**100**	**0**	**0**	**0**	**0**	**0**	**100**
	35			0.02	100	0	0	0	0	0	100	0.03	100	0	0	0	0	0	100	0.08	40	0	60	0	0	60	40
SVM	50			0.47	0	0	100	0	0	100	0	0.28	0	0	80	20	0	100	0	0.44	0	0	100	0	0	100	0
Alg.	%p	K		DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER
	20			0.06	0	0	100	0	0	100	0	**0.01**	**100**	**0**	**0**	**0**	**0**	**0**	**100**	0.02	100	0	0	0	0	0	100
	35			0.14	0	0	60	0	40	100	0	**0.01**	**100**	**0**	**0**	**0**	**0**	**0**	**100**	0.04	100	0	0	0	0	0	100
	50	3		0.47	0	0	100	0	0	100	0	0.32	0	0	100	0	0	100	0	0.26	0	0	100	0	0	100	0
	20			0.02	100	0	0	0	0	0	100	0.01	**100**	**0**	**0**	**0**	**0**	**0**	**100**	0.01	100	0	0	0	0	0	100
	35			0.09	0	0	60	0	40	100	0	0.01	**100**	**0**	**0**	**0**	**0**	**0**	**100**	0.04	100	0	0	0	0	0	100
	50	9		0.47	0	0	100	0	0	100	0	0.32	0	0	100	0	0	100	0	0.27	0	0	100	0	0	100	0
	20			0.02	100	0	0	0	0	100	0	0.01	**100**	**0**	**0**	**0**	**0**	**0**	**100**	**0.01**	**100**	**0**	**0**	**0**	**0**	**0**	**100**
KNN	35			**0.06**	**20**	**0**	**40**	**0**	**40**	**80**	**20**	**0.02**	**100**	**0**	**0**	**0**	**0**	**0**	**100**	**0.03**	**100**	**0**	**0**	**0**	**0**	**0**	**100**
	50	15		0.46	0	0	100	0	0	100	0	0.32	0	0	100	0	0	100	0	0.28	0	0	100	0	0	100	0
Alg.	%p	ne		DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER
	20			0.1	0	0	40	0	60	100	0	**0.13**	**20**	**0**	**0**	**0**	**80**	**100**	**0**	0.12	0	0	40	0	60	100	0
	35			0.19	0	0	20	0	80	100	0	**0.14**	**20**	**0**	**0**	**0**	**80**	**80**	**20**	0.16	0	0	20	0	80	100	0
	50	5		**0.48**	**0**	**0**	**100**	**0**	**0**	**100**	**0**	**0.42**	**0**	**0**	**100**	**0**	**0**	**100**	**0**	0.43	0	0	100	0	0	100	0
	20			0.05	60	0	0	0	40	40	60	0.11	20	0	0	0	0	80	20	0.08	40	0	20	0	40	60	40
	35			0.15	0	0	0	0	100	100	0	0.1	40	0	0	0	0	60	60	0.14	20	0	40	0	40	80	20
	50	50		**0.48**	**0**	**0**	**100**	**0**	**0**	**100**	**0**	0.38	0	0	100	0	0	100	0	0.44	0	0	100	0	0	100	0
	20			0.05	60	0	0	0	40	40	60	0.1	40	0	0	0	60	60	40	0.08	20	0	0	0	80	80	20
	35			0.15	0	0	0	0	100	100	0	0.1	40	0	0	0	60	60	40	0.14	20	0	20	0	40	60	20
	50	100		0.46	0	0	0	0	0	100	0	0.37	0	0	100	0	0	100	0	0.44	0	0	100	0	0	100	0
	20			0.05	60	0	0	0	40	40	60	0.1	40	0	0	0	60	60	40	0.08	20	0	0	0	60	60	20
RF	35			0.15	0	0	0	0	100	100	0	0.1	40	0	0	0	60	60	40	0.15	20	0	20	0	60	80	20
	50	500		0.45	0	0	100	0	0	100	0	0.38	0	0	100	0	0	100	0	0.45	0	0	100	0	0	100	0
Alg.	%p	#HL	#N	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER	DiffA	closeB	Uatt	Satt	HighS	HighU	DSR	NER
	20			0.02	100	0	0	0	0	0	100	0.08	40	0	40	0	20	60	40	0.13	40	0	40	0	20	60	40
	35			0.08	20	0	60	0	20	80	20	0.13	40	0	20	0	0	60	40	0.23	0	0	60	0	40	100	0
	50		5	0.48	0	0	100	0	0	100	0	0.37	0	0	100	0	0	100	0	0.5	0	0	100	0	0	100	0
	20			**0.15**	**0**	**0**	**100**	**0**	**0**	**100**	**0**	0.08	40	0	40	0	20	60	40	**0.15**	**0**	**0**	**80**	**0**	**20**	**100**	**0**
	35			0.2	0	0	60	0	20	80	0	0.08	60	0	0	0	40	40	60	**0.24**	**0**	**0**	**60**	**0**	**40**	**100**	**0**
	50		50	0.44	0	0	100	0	0	100	0	0.36	0	0	100	0	0	100	0	0.5	0	0	100	0	0	100	0
	20			0.13	0	0	100	0	0	100	0	0.08	40	0	40	0	20	60	40	0.12	20	0	60	0	20	80	20
	35			0.18	0	0	80	0	20	100	0	0.09	60	0	20	0	20	40	60	0.24	0	0	40	0	40	80	0
	50		100	0.46	0	0	100	0	0	100	0	0.38	0	0	100	0	0	100	0	0.5	0	0	100	0	0	100	0
	20			0.13	0	0	100	0	0	100	0	0.08	40	0	20	0	20	40	60	0.12	0	0	80	0	20	100	0
	35			0.19	0	0	80	0	20	100	0	0.11	40	0	20	0	40	60	40	0.23	0	0	80	0	20	100	0
	50		150	0.45	0	0	100	0	0	100	0	0.4	0	0	80	20	0	100	0	**0.51**	**0**	**0**	**100**	**0**	**0**	**100**	**0**
	20			0.03	80	0	20	0	0	20	80	0.07	60	0	0	0	20	40	60	**0.15**	**0**	**0**	**60**	**0**	**40**	**100**	**0**
	35			0.09	20	0	60	0	20	80	20	0.13	40	0	20	0	40	60	40	0.23	0	0	40	0	40	80	0
	50		5	0.44	0	0	100	0	0	100	0	0.35	0	0	100	0	0	100	0	0.47	0	0	100	0	0	100	0
	20			0.12	0	0	80	0	20	100	0	0.07	60	0	20	0	20	40	60	0.14	0	0	40	0	60	100	0
	35			0.19	0	0	60	0	40	100	0	0.08	60	0	20	0	20	40	60	0.23	0	0	40	0	60	100	0
	50		50	0.46	0	0	100	0	0	100	0	0.35	0	0	60	40	0	100	0	0.49	0	0	100	0	0	100	0
	20			0.11	0	0	80	0	20	100	0	0.1	40	0	20	0	40	60	40	0.13	20	0	20	0	60	80	20
	35			**0.21**	**0**	**0**	**60**	**0**	**40**	**100**	**0**	0.12	40	0	40	0	20	60	40	**0.24**	**0**	**0**	**60**	**0**	**40**	**100**	**0**
	50		100	0.46	0	0	100	0	0	100	0	0.39	0	0	100	0	0	100	0	0.49	0	0	100	0	0	100	0
	20			0.12	0	0	60	0	20	100	0	0.1	40	0	20	0	20	40	60	0.12	20	0	60	0	20	80	20
	35			0.17	0	0	80	0	20	100	0	0.11	60	0	20	0	20	40	60	**0.24**	**0**	**0**	**60**	**0**	**40**	**100**	**0**
	50		150	0.44	0	0	100	0	0	100	0	0.39	0	0	80	20	0	100	0	0.51	0	0	100	0	0	100	0
	20			0.03	100	0	0	0	0	0	100	0.06	60	0	20	0	20	40	60	0.13	20	0	60	0	20	80	20
	35			0.07	60	0	40	0	0	40	60	0.1	60	0	40	0	0	40	60	0.21	20	0	20	0	60	80	20
	50		5	0.47	0	0	100	0	0	100	0	0.33	0	0	100	0	0	100	0	0.5	0	0	100	0	0	100	0
	20			0.12	0	0	60	0	0	60	40	0.07	40	0	60	0	0	60	40	0.14	0	0	80	0	20	100	0
	35			**0.21**	**0**	**0**	**20**	**0**	**80**	**100**	**0**	0.1	60	0	20	0	20	40	60	0.23	0	0	20	0	80	100	0
	50		50	0.45	0	0	100	0	0	100	0	0.36	0	0	80	20	0	100	0	0.49	0	0	100	0	0	100	0
	20			0.12	0	0	100	0	0	100	0	0.11	40	0	40	0	20	60	40	0.13	20	0	60	0	20	80	20
	35			**0.21**	**0**	**0**	**60**	**0**	**40**	**100**	**0**	0.12	40	0	20	0	40	60	40	0.23	0	0	40	0	60	100	0
	50		100	0.46	0	0	100	0	0	100	0	0.39	0	0	100	0	0	100	0	0.47	0	0	80	0	0	80	0
	20			0.12	0	0	80	0	20	100	0	0.12	40	0	20	0	40	60	40	0.12	20	0	40	0	40	80	20
	35			0.15	0	0	60	0	40	100	0	0.12	60	0	20	0	20	40	60	0.23	0	0	40	0	60	100	0
	50		150	0.45	0	0	100	0	0	100	0	0.39	0	0	100	0	0	100	0	**0.51**	**0**	**0**	**100**	**0**	**0**	**100**	**0**

Table XVIII
PHP - BENCHMARK DETECTOR PER POISON ATTACK

				LF_a							FR_a							DI_a						
Alg.	%p		$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20		0.03	87.5	0	0	0	12.5	12.5	87.5	0.06	75	0	25	0	0	25	75	0.05	62.5	0	0	12.5	25	37.5	62.5
	35		0.13	37.5	0	25	0	37.5	62.5	37.5	0.05	50	0	25	12.5	12.5	50	50	0.1	37.5	0	0	50	12.5	62.5	37.5
SVM	50		0.4	0	0	87.5	12.5	0	100	0	0.39	0	0	100	0	0	100	0	0.31	0	0	25	75	0	100	0
Alg.	%p	K	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20		0.07	50	0	37.5	0	12.5	50	50	0.09	75	0	12.5	0	12.5	25	75	0.06	75	0	12.5	0	12.5	25	75
	35		***0.21***	***0***	***0***	***87.5***	***0***	***12.5***	***100***	***0***	0.14	0	0	87.5	0	12.5	100	0	0.12	0	0	25	62.5	12.5	100	0
	50	3	***0.45***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	***0.41***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	0.29	***0***	***0***	62.5	37.5	0	100	0
	20		0.03	100	0	0	0	0	0	100	0.07	62.5	0	25	0	12.5	37.5	62.5	0.05	87.5	0	0	0	12.5	12.5	87.5
	35		0.16	0	0	87.5	0	12.5	100	0	0.11	12.5	0	75	0	12.5	87.5	12.5	***0.13***	***0***	***0***	***0***	***87.5***	***12.5***	***100***	***0***
	50	9	0.43	0	0	100	0	0	100	0	0.38	0	0	100	0	0	100	0	0.29	0	0	37.5	62.5	0	100	0
	20		0.03	100	0	0	0	0	100	0	0.06	62.5	0	25	0	12.5	37.5	62.5	0.04	87.5	0	0	87.5	12.5	12.5	87.5
	35		0.14	0	0	75	0	25	100	0	0.09	50	0	37.5	0	12.5	50	50	0.12	0	0	0	87.5	12.5	100	0
KNN	50	15	0.42	0	0	100	0	0	100	0	0.37	0	0	100	0	0	100	0	0.28	0	0	37.5	62.5	0	100	0
Alg.	%p	ne	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20		***0.09***	***25***	***0***	***62.5***	***0***	***12.5***	***75***	***25***	***0.12***	***0***	***0***	***87.5***	***0***	***12.5***	***100***	***0***	0.09	50	***0***	***37.5***	12.5	***12.5***	***50***	***50***
	35		0.21	0	0	75	0	25	100	0	***0.16***	***0***	***0***	***75***	***0***	***25***	***100***	***0***	0.09	50	0	25	12.5	25	50	50
	50	5	0.4	0	0	100	0	0	100	0	0.38	0	0	100	0	0	100	0	0.33	0	0	25	75	0	100	0
	20		0.07	62.5	0	25	0	12.5	37.5	62.5	0.11	12.5	0	62.5	0	25	87.5	12.5	0.09	50	0	37.5	0	12.5	50	50
	35		0.2	0	0	62.5	0	37.5	100	0	0.13	25	0	50	0	25	75	25	0.08	50	0	12.5	25	12.5	50	50
	50	50	0.43	0	0	100	0	0	100	0	***0.41***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	0.37	0	0	12.5	87.5	0	100	0
	20		0.07	62.5	0	25	0	12.5	37.5	62.5	0.12	12.5	0	62.5	0	25	87.5	12.5	0.09	50	0	37.5	0	12.5	50	50
	35		0.2	0	0	62.5	0	37.5	100	0	0.13	25	0	50	0	25	75	25	0.08	50	0	12.5	25	12.5	50	50
	50	100	0.43	0	0	100	0	0	100	0	***0.41***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	0.37	0	0	12.5	87.5	0	100	0
	20		0.07	62.5	0	25	0	12.5	37.5	62.5	0.11	12.5	0	62.5	0	25	87.5	12.5	0.09	50	0	37.5	0	12.5	50	50
RF	35		0.2	0	0	50	0	50	100	0	0.14	25	0	50	0	25	75	25	0.08	37.5	0	12.5	37.5	12.5	62.5	37.5
	50	500	0.44	0	0	100	0	0	100	0	***0.41***	***0***	***0***	***100***	***0***	***0***	***100***	***0***	***0.39***	***0***	***0***	***12.5***	***87.5***	***0***	100	0
Alg.	%p	#N	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER	$Diff_{acc}$	$closeB$	$Uatt$	$Satt$	$HighS$	$HighU$	DSR	NER
	20		0.03	50	12.5	12.5	25	0	37.5	62.5	0.04	50	12.5	0	25	12.5	37.5	62.5	0.09	37.5	0	25	12.5	25	62.5	37.5
	35		0.14	25	0	37.5	12.5	25	75	25	0.05	37.5	0	37.5	0	25	62.5	37.5	0.07	25	12.5	12.5	37.5	12.5	62.5	37.5
	50	5	0.33	0	0	100	0	0	100	0	0.3	0	0	62.5	37.5	0	100	0	0.27	0	0	12.5	87.5	0	100	0
	20		0.1	37.5	0	25	0	37.5	62.5	37.5	0.12	25	0	62.5	50	0	25	75	0.1	37.5	0	12.5	12.5	37.5	62.5	37.5
	35		0.2	0	0	75	0	25	100	0	0.09	12.5	0	37.5	25	25	87.5	12.5	0.1	37.5	0	12.5	37.5	12.5	62.5	37.5
	50	50	0.08	62.5	0	12.5	12.5	12.5	37.5	62.5	0.11	25	0	25	12.5	37.5	75	25	0.1	0	0	12.5	37.5	50	100	0
	20		0.18	12.5	0	50	0	37.5	87.5	12.5	0.05	12.5	0	62.5	25	37.5	87.5	12.5	0.08	12.5	0	12.5	62.5	12.5	87.5	12.5
	35	100	0.29	0	0	50	50	0	100	0	0.27	0	0	50	50	0	100	0	0.26	0	0	12.5	87.5	0	100	0
	50		0.1	57.14	0	14.29	0	28.57	42.86	57.14	0.13	28.57	0	28.57	0	42.86	71.43	28.57	0.12	14.29	0	14.29	71.43	85.72	14.29	
	20		0.18	0	0	57.14	0	42.86	100	0	0.05	42.86	0	0	28.57	28.57	57.14	42.86	0.08	14.29	0	0	71.43	14.29	85.72	14.29
	35		0.3	0	0	71.43	28.57	0	100	0	0.28	0	0	57.14	42.86	0	100	0	0.26	0	0	0	85.71	14.29	100	0
	50	150	0.06	50	0	25	12.5	12.5	50	50	0.05	37.5	0	0	50	12.5	62.5	37.5	0.07	25	0	25	12.5	37.5	75	25
	20		0.12	25	0	62.5	0	12.5	75	25	0.03	62.5	0	0	25	12.5	37.5	62.5	0.07	62.5	0	0	25	12.5	37.5	62.5
	35		0.3	0	0	87.5	12.5	0	100	0	0.28	0	0	62.5	37.5	0	100	0	0.26	0	0	12.5	87.5	0	100	0
	50	5	0.1	37.5	0	25	0	37.5	62.5	37.5	0.1	62.5	0	12.5	0	25	37.5	62.5	0.1	25	0	12.5	0	62.5	75	25
	20		0.19	0	0	75	0	25	100	0	0.08	37.5	0	37.5	25	0	62.5	37.5	0.1	50	0	12.5	25	12.5	50	50
	35		0.29	0	0	100	0	0	100	0	0.28	0	0	75	25	0	100	0	0.25	0	0	12.5	87.5	0	100	0
	50	50	0.11	42.86	0	28.57	14.29	14.29	57.15	42.86	0.09	28.57	0	14.29	28.57	28.57	71.43	28.57	0.11	28.57	0	14.29	28.57	28.57	71.43	28.57
	20		0.18	14.29	0	57.14	0	28.57	85.71	14.29	0.06	57.14	0	0	42.86	14.29	42.86	57.14	0.06	14.29	0	28.57	57.14	0	85.71	14.29
	35	100	0.29	0	0	71.43	28.57	0	100	0	0.27	0	0	42.86	57.14	0	100	0	0.25	0	0	14.29	85.71	0	100	0
	50		0.11	50	0	33.33	0	16.67	50	50	0.12	33.33	0	0	16.67	50	66.67	33.33	0.11	33.33	0	16.67	16.67	33.33	66.67	33.33
	20	2	0.2	0	0	83.33	0	16.67	100	0	0.06	16.67	0	0	66.67	16.67	83.34	16.67	0.07	16.67	0	33.33	50	0	83.33	16.67
	35		0.2	0	0	66.67	33.33	0	100	0	0.28	0	0	33.33	66.67	0	100	0	0.26	0	0	16.67	83.33	0	100	0
	50	150	0.03	50	0	12.5	25	12.5	50	50	0.06	62.5	0	12.5	12.5	12.5	37.5	62.5	0.1	12.5	0	25	12.5	50	87.5	12.5
	20		0.12	50	0	37.5	12.5	0	50	50	0.06	62.5	0	12.5	25	0	37.5	62.5	0.06	37.5	0	12.5	37.5	12.5	62.5	37.5
	35		0.32	0	0	87.5	12.5	0	100	0	0.28	0	0	25	62.5	12.5	100	0	0.24	0	0	25	75	0	100	0
	50	5	0.1	50	0	25	0	25	50	50	0.1	50	0	25	12.5	12.5	50	50	0.11	37.5	0	25	12.5	25	62.5	37.5
	20		0.18	12.5	0	50	0	37.5	87.5	12.5	0.09	50	0	12.5	25	12.5	50	50	0.07	37.5	0	25	37.5	0	62.5	37.5
MLP	50		0.3	0	0	100	0	0	100	0	0.29	0	0	62.5	37.5	0	100	0	0.24	0	0	12.5	87.5	0	100	0
	20		0.09	50	0	37.5	0	12.5	50	50	0.1	25	0	12.5	25	37.5	75	25	0.11	37.5	0	12.5	0	50	62.5	37.5
	35		0.17	12.5	0	62.5	0	25	87.5	12.5	0.07	37.5	0	0	37.5	25	62.5	37.5	0.07	12.5	0	12.5	62.5	12.5	87.5	12.5
	50	100	0.24	0	0	62.5	37.5	0	100	0	0.25	0	0	50	50	0	100	0	0.24	0	0	12.5	87.5	0	100	0
	20		0.08	50	0	0	25	25	50	50	0.1	25	0	0	25	50	75	25	0.11	25	0	25	25	25	75	25
	35	3	0.19	0	0	75	0	25	100	0	0.04	0	0	0	100	0	100	0	0.04	0	0	25	75	0	100	0
	50	150	0.23	0	0	75	25	0	100	0	0.27	0	0	50	50	0	100	0	0.27	0	0	0	100	0	100	0

Table XIX
PERMANOVA ADDITIONAL TESTS

	%p=[20,35]			%p=[20,50]			%p=[35,50]		
	LF_a	FR_a	DI_a	LF_a	FR_a	DI_a	LF_a	FR_a	DI_a
C/C++	0.0469	0.0029	0.0009	0.1168	0.0009	0.0009	0.001998	0.0009	0.0009
PHP	0.0009	0.0029	0.0009	0.0009	0.0009	0.0009	0.0449	0.0009	0.0009
C#	0.0129	0.0009	0.0009	0.0089	0.0009	0.0009	0.0139	0.0009	0.0009

[5] H. Mozaffari, V. Shejwalkar, and A. Houmansadr, "Every vote counts: Ranking-based training of federated learning to resist poisoning attacks," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[6] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," in *Proceedings of the fourth ACM international workshop on security and privacy analytics*, 2018, pp. 31–39.

[7] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet?" *Journal of Systems and Software*, vol. 188, p. 111283, 2022.

[8] M. Fu, C. Tantithamthavorn, T. Le, Y. Kume, V. Nguyen, D. Phung, and J. Grundy, "Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities," *Empirical Software Engineering*, vol. 29, no. 1, p. 4, 2024.

[9] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "Vulchecker: Graph-based vulnerability localization in source code," in *31st USENIX Security Symposium, Security 2022*, 2023.

[10] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1559–1575.

[11] W. Sun, Y. Chen, G. Tao, C. Fang, X. Zhang, Q. Zhang, and B. Luo, "Backdooring neural code search," *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL 2023)*, 2023.

[12] H. Aghakhani, W. Dai, A. Manoel, X. Fernandes, A. Kharkar, C. Kruegel, G. Vigna, D. Evans, B. Zorn, and R. Sim, "Trojanpuzzle: Covertly poisoning code-suggestion models," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1122–1140.

[13] J. Li, Z. Li, H. Zhang, G. Li, Z. Jin, X. Hu, and X. Xia, "Poison attack and poison detection on deep source code processing models," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 3, pp. 1–31, 2024.

[14] H. Xu, "Thesis: Environment poisoning in reinforcement learning: attacks and resilience," 2023.

[15] Z. Tian, L. Cui, J. Liang, and S. Yu, "A comprehensive survey on poisoning attacks and countermeasures in machine learning," *ACM Computing Surveys*, vol. 55, no. 8, pp. 1–35, 2022.

[16] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," *Advances in neural information processing systems*, vol. 31, 2018.

[17] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. Molloy, and B. Srivastava, "Detecting backdoor attacks on deep neural networks by activation clustering," *arXiv preprint arXiv:1811.03728*, 2018.

[18] G. Severi, J. Meyer, S. Coull, and A. Oprea, "{Explanation-Guided} backdoor poisoning attacks against malware classifiers," in *30th USENIX security symposium (USENIX security 21)*, 2021, pp. 1487–1504.

[19] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," 2023. [Online]. Available: https://github.com/wagner-group/diversevul

[20] Z. Yang, B. Xu, J. M. Zhang, H. Kang, J. Shi, J. He, and D. Lo, "Stealthy backdoor attack for code models," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–21, 5555.

[21] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 526–537.

[22] Y. Wan, S. Zhang, H. Zhang, Y. Sui, G. Xu, D. Yao, H. Jin, and L. Sun, "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*, 2022, pp. 1233–1245.

[23] S. Qi, Y. Yang, S. Gao, C. Gao, and Z. Xu, "Badcs: A backdoor attack framework for code search," *arXiv preprint arXiv:2305.05503*, 2023.

[24] Y. Li, S. Liu, K. Chen, X. Xie, T. Zhang, and Y. Liu, "Multi-target backdoor attacks for code pre-trained models," *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023.

[25] G. Ramakrishnan and A. Albarghouthi, "Backdoors in neural models of source code," in *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE, 2022, pp. 2892–2899.

[26] Z. Sun, X. Du, F. Song, M. Ni, and L. Li, "Coprotector: Protect open-source code against unauthorized training usage with data poisoning," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 652–660.

[27] H. Zhang, S. Lu, Z. Li, Z. Jin, L. Ma, Y. Liu, and G. Li, "Codebert-attack: Adversarial attack against source code deep learning models via pre-trained model," *Journal of Software: Evolution and Process*, vol. 36, no. 3, p. e2571, 2024.

[28] D. Cotroneo, C. Improta, P. Liguori, and R. Natella, "Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 280–292.

[29] M. Zagane, M. K. Abdi, and M. Alenezi, "Deep learning for software vulnerabilities detection using code metrics," *IEEE Access*, vol. 8, pp. 74 562–74 570, 2020.

[30] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: An image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2365–2376.

[31] W. Tang, M. Tang, M. Ban, Z. Zhao, and M. Feng, "Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection," *Journal of Systems and Software*, vol. 199, p. 111623, 2023.

[32] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," in *2022 International joint conference on neural networks (IJCNN)*. IEEE, 2022, pp. 1–8.

[33] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: vulnerability detection with deep learning on a natural codebase for python," *Information and Software Technology*, vol. 144, p. 106809, 2022.

[34] A. Fidalgo, I. Medeiros, P. Antunes, and N. Neves, "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, 2020, pp. 465–476.

[35] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Network and Distributed System Security (NDSS) Symposium*, 2018.

[36] Y. Li, S. Wang, and T. N. Nguyen, "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*, 2021, pp. 292–303.

[37] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 315–317.

[38] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.

[39] A. Gupta, R. Parmar, P. Suri, and R. Kumar, "Determining accuracy rate of artificial intelligence models using python and r-studio," in *2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*. IEEE, 2021, pp. 889–894.

[40] H. Alibrahim and S. A. Ludwig, "Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization," in *2021 IEEE congress on evolutionary computation (CEC)*. IEEE, 2021, pp. 1551–1559.

[41] A. Tharwat, "Principal component analysis-a tutorial," *International Journal of Applied Pattern Recognition*, vol. 3, no. 3, pp. 197–240, 2016.

[42] M. Charikar, V. Guruswami, R. Kumar, S. Rajagopalan, and A. Sahai, "Combinatorial feature selection problems," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 631–640.

[43] F. Pecorelli, S. Lujan, V. Lenarduzzi, F. Palomba, and A. De Lucia, "On the adequacy of static analysis warnings with respect to code smell prediction," *Empirical Software Engineering*, vol. 27, no. 3, p. 64, 2022.

[44] S. Romano, G. Toriello, P. Cassieri, R. Francese, and G. Scanniello, "A folklore confirmation on the removal of dead code," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 333–338.

[45] P. E. Black, "Sard: A software assurance reference dataset," 2017. [Online]. Available: https://samate.nist.gov/SARD

[46] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. Molloy, and B. Srivastava, "Detecting backdoor attacks on deep neural networks by activation clustering," in *SafeAI Workshop, 2019*, 2019.

[47] I. Malavolta and Others, "Javascript dead code identification, elimination, and empirical assessment," *IEEE Transactions on Software Engineering*, 2023.

[48] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[49] S. Saha, T. Zhang, and Others, "Check your other door! creating backdoor attacks in the frequency domain," *arXiv preprint arXiv:2109.05507*, 2021.

[50] V. R. Joseph, "Optimal ratio for data splitting," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 15, no. 4, pp. 531–538, 2022.

[51] N. Virvilis and D. Gritzalis, "The big four-what we did wrong in advanced persistent threat detection?" in *2013 international conference on availability, reliability and security*. IEEE, 2013, pp. 248–254.

[52] N. Carlini, M. Jagielski, C. A. Choquette-Choo, D. Paleka, W. Pearce, H. Anderson, A. Terzis, K. Thomas, and F. Tramèr, "Poisoning web-scale training datasets is practical," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 407–425.

[53] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *Proceedings of the 29th International Coference on International Conference on Machine Learning (ICML'12)*. Omnipress, 2012, pp. 1467–1474.

[54] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[55] M. J. Anderson, "Permutational multivariate analysis of variance (permanova)," *Wiley statsref: statistics reference online*, pp. 1–15, 2014.

[56] J. Haurogné, N. Basheer, and S. Islam, "Vulnerability detection using bert based llm model with transparency obligation practice towards trustworthy ai," *Machine Learning with Applications*, vol. 18, p. 100598, 2024.

[57] A. A. Mahyari, "Harnessing the power of llms in source code vulnerability detection," in *MILCOM 2024-2024 IEEE Military Communications Conference (MILCOM)*. IEEE, 2024, pp. 251–256.

[58] R. I. T. Jensen, V. Tawosi, and S. Alamir, "Software vulnerability and functionality assessment using llms," in *2024 IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE)*. IEEE, 2024, pp. 25–28.

[59] Y. Shamoo, "Application of large language models (llms) for software vulnerability detection," in *Título del Libro*, M. Omar and H. M. Zangana, Eds. IGI Global, 2024.

[60] M. T. Alam, R. Halder, and A. Maiti, "Detection made easy: Potentials of large language models for solidity vulnerabilities," *arXiv preprint arXiv:2409.10574*, 2024.

[61] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "Llms in software security: A survey of vulnerability detection techniques and insights," *arXiv e-prints*, pp. arXiv–2502, 2025.

[62] C. Cortes, M. Mohri, and A. Rostamizadeh, "L2 regularization for learning kernels," *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI2009)*, 2012.

[63] T.-H. Lee, A. Ullah, and R. Wang, "Bootstrap aggregating and random forest," *Macroeconomic forecasting in the era of big data: Theory and practice*, pp. 389–429, 2020.

[64] J. Gou, T. Xiong, Y. Kuang *et al.*, "A novel weighted voting for k-nearest neighbor rule." *J. Comput.*, vol. 6, no. 5, pp. 833–840, 2011.

[65] M. Anderson, "Permanova+ for primer: guide to software and statistical methods." *Primer-E Limited.*, 2008.