HOSTED BY

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

Predicting input validation vulnerabilities based on minimal SSA features and machine learning

Abdalla Wasef Marashdih [a], Zarul Fitri Zaaba [a,*], Khaled Suwais [b]

[a] School of Computer Sciences, Universiti Sains Malaysia, 11800 Pulau Pinang, Malaysia
[b] Faculty of Computer Studies, Arab Open University, Saudi Arabia

ARTICLE INFO

Article history:
Received 8 May 2022
Revised 23 August 2022
Accepted 12 September 2022
Available online 16 September 2022

Keywords:
Input validation vulnerabilities
XSS
SQLi
Minimal SSA
Features extraction
Machine learning

ABSTRACT

Structured Query Language injection (SQLi) and Cross-Site Scripting (XSS) are the most renowned kinds of input validation vulnerabilities. Of late, vulnerability prediction models based on machine learning have been gaining acceptance in the domain of Web security. Such models offer an easy and effective way of dealing with web application security concerns. However, most of them, in particular, rely on complex graphs generated from source code or regex patterns based on expert knowledge. This paper proposed a method for extracting features from source code and predicting input validation vulnerabilities using machine learning algorithms. The proposed method can extract all features related to the flow of vulnerabilities among the programs and remove the features that are irrelevant to the vulnerability flow. In addition, each vulnerability's context has been assigned, providing additional data for our model to use in learning about the vulnerability context. Compared to other related methods, the feature extraction method proposed in this paper has been found to have high reusability and better performance. The best model related to the LSTM classifier had a 98.1% recall rate, a 97.9% precision, an accuracy of 98.67%, and a 99.03% area under the curve (AUC) in the test dataset.

1. Introduction

Web applications are now widely regarded as one of the most common platforms for presenting data and releasing services on the Internet. However, there exist intrinsic security issues, making exploitation of programmatic vulnerabilities relatively easy. Two common exploits are SQL injection (SQLi) and cross-site scripting (XSS). The list of the 10 most common web application vulnerabilities in the Open Web Application Security Project (OWASP) (OWASP, 2020) includes SQLi and XSS. SQLi denotes the introduction of control characters or SQL commands for web inputs in order to adjust how SQL queries can be executed at the backend. The XSS exploit is employed by appending script tags to the web URLs so that individuals can be tricked into clicking the malicious links. To prevent the online applications getting compromised, security organisations, developers and white hat hackers are continuously putting efforts to repair, identify and report these vulnerabilities (CVE, 2022).

Machine learning (ML) is regarded as the optimal technique for identifying vulnerabilities. The extraction of relevant information that may differentiate between vulnerable and benign code is one of the most critical and necessary stages in the construction of a prediction model. Some researchers used static and dynamic code attributes (Shar and Tan, 2013; Shar et al., 2014; Gupta et al., 2015) and data mining (Walden et al., 2014; Medeiros et al., 2015), which can help to extract features based on the applications' source code as well as predict if these applications include any vulnerabilities. Other researchers (Fang et al., 2019) extracted features using data flow analysis. Recently, Intermediate Representation (IR) has been employed in both of the recent studies (Li et al., 2021; Li et al., 2021), to simplify the source code for analysis, like in the form of Static Single Assignment (SSA).

The gap in the previous studies (Shar et al., 2014; Shar and Tan, 2013; Walden et al., 2014; Li et al., 2021; Li et al., 2021; Fang et al., 2019; Li et al., 2020) where thay all extract the features from the source code, which contains all the information about the flow of the program. However, there are a number of features that are unrelated to the flow of the vulnerability in the programs. Removing unrelated features from the programs' extracted features will help to improve the performance of the model (Zhang et al., 2018; Jin et al., 2005; Hong et al., 2020). Spens et al. (2018) stated that reducing the amount of irrelevant features considered for classification increases the classifier's accuracy, whereas Ali et al.

* Corresponding author.
E-mail address: zarulfitri@usm.my (Z.F. Zaaba).

https://doi.org/10.1016/j.jksuci.2022.09.010
1319-1578/© 2022 The Author(s). Published by Elsevier B.V. on behalf of King Saud University.

(2019) discovered that irrelevant features reduce a classification process's precision rate.

In this paper, we first propose a method for extracting the relevant vulnerability features from the source code, and then removing the features that are irrelevant to the vulnerability flow in the programs. The proposed method starts with the transformation of the source code into an Abstract Syntax Tree (AST). Then, each node is compiled exactly into an SSA form. The SSA is then changed to Minimal Static Single Assignment (MSSA), which helps define the control and data dependencies between each of the program paths and gets rid of any duplicate variables and references in the multi-assignment for each variable in the program.

Second, each of the generated paths can be filtered to remove non-representative code as well as the string variable declaration. This approach considerably reduces the vulnerability context terms in the extracted features, thus dismissing the long-term dependency in the vulnerability flow common in the program.

Another issue is that, most of the previous studies (Shar et al., 2014; Shar and Tan, 2013; Li et al., 2021; Li et al., 2021; Fang et al., 2019) did not consider the vulnerability context during the feature extraction of the programs. The consideration of the existence of the vulnerabilities in such a HyperText Markup Language (HTML) context will give the model more knowledge about the vulnerability context and enhance the performance of the prediction results. With regards to the current web applications, a user-input has been employed in an output-statement along with HTML code to yield a dynamic HTML document. This combination can be defined as an HTML context, which also helps the web browsers during interpretation of the HTML document in a different manner. The common HTML contexts wherein a user-input has been referred include HTML Element, Cascading Style Sheets (CSS), Script, HTML Uniform Resource Locator (URL) context and Attribute. Each of the contexts possesses its own characteristics and needs to follow different defense mechanisms against vulnerabilities (Shar and Tan, 2012; Gupta et al., 2018).

The context of vulnerability was not taken into account in the previous studies (Medeiros et al., 2015; Fang et al., 2019; Scandariato et al., 2014; Li et al., 2021; Li et al., 2021) when producing their dataset characteristics. Based on the OWASP prevention rules (OWASP, 2021), there are various contexts with regards to each vulnerability, which need different methods in order to evade and sanitize the user inputs. Thus, we have regarded this issue in our method, in which each of the vulnerability sources has been segmented by conforming to the OWASP prevention rules (OWASP, 2021) and a unique feature has been assigned for that category. Furthermore, all the sanitization functions employed in the program flow have been tokenized based on their block type and name in order to provide additional information to different machine learning (ML) algorithms regarding every case of vulnerability.

The motivation of our study is to propose a method that can extract the whole features from the source code that are related to the vulnerability flow in the programs and eliminate those features that do not make sense for the vulnerability flow in the programs. In addition, the existence of different vulnerability contexts in the programs requires different sanitization methods to be sanitized. Therefore, the proposed method aims to differentiate between the various vulnerability contexts in the programs by assigning a unique token for each vulnerability occurrence case in the program. In summary, the following are the contributions to this paper:

- An enhancement feature extraction method is proposed to extract the relevant vulnerability features from the source code based on MSSA representation.
- A backward slicing approach is carried out for traversing the generated features as well as removing the features unrelated to the vulnerability flow.
- We provide additional unique features for each vulnerability context based on several prevention rules, which gives additional knowledge for the classifiers to learn the vulnerability context in the program.
- We conducted comprehensive experiments with different feature extraction methods, different classifiers, and literature studies.

Overall, the proposed method shows its ability to generate the features that are related to the vulnerability flow in the program, and the results of different ML and neural network classifiers demonstrate the effectiveness of the generated dataset in predicting XSS and SQLi in PHP vulnerable files. It was noted that the best performance results achieved by LSTM and BiLSTM were nearly equal in different evaluation metrics. However, the BiLSTM requires more time than the LSTM to achieve the best results. For those, we considered LSTM classifier to be the best choice for predicting input validation vulnerabilities on the generated dataset.

The rest of the paper is structured as follows. In Section 2, we present the background of input validation vulnerabilities (including SQLi and XSS), SSA representation, and vulnerability prediction models. In Section 3, the related works is discussed. The methodology of the proposed method is covered in Section 4. The experimental results and assessment are indicated in Section 5. Conclusion and future directions are presented in Section 6.

2. Background and Motivation

The risky nature of web technology is well known, which enables combining data and code together. Thus, these web applications may have associated input validation vulnerabilities. In this section, the input validation vulnerabilities are described first, followed by the SSA representation of the source code. Finally, machine learning was discussed.

2.1. Input Validation Vulnerability

User input sets should not be considered trustworthy based on one threat model (Li and Xue, 2011). Validation must be performed before providing data input to an application from unverified sources (e.g., building SQL commands or server response). User input is generally validated by sanitization methods that validate an untrusted data set so that it can be trusted.

Any web application having such weaknesses will not be able to verify input appropriately; hence, such an application can be affected by data flow, input validation, or script injection operations. XSS and SQLi are the most commonly used attack forms. These attacks can be distinguished based on malicious code execution points.

2.1.1. Cross Site Scripting

XSS (Martin and Lam, 2008) is based on injecting JavaScript code, allowing the execution of a script on the target system browser by a malicious user. Web applications are affected by malicious JavaScript code snippets. Script injection is executed so that it appears benign to an unsuspecting individual. Lastly, a trusted domain runs the script compromised using harmful code (Gupta and Gupta, 2017).

Stored, reflected, and DOM-based XSS are three XSS attack categories. Stored XSS attacks comprise code injection into a website, where the script is saved in the database (Yusof and Pathan, 2016). Reflected XSS happens when a web application sends unacceptable user input. A malicious user might disguise the malicious code in the URL so that it is mistaken for user input. DOM-specific XSS attacks are proxies for stored or reflected XSS scripts. In this form, the harmless JavaScript fragments execute on an individual's web browser. Only then does the malicious code get executed (Gupta and Gupta, 2017). Listing 1 specifies a sample XSS attack using source code and several contexts concerning vulnerability presence.

Listing 1: An example of an XSS vulnerability in the source code

```
<?php
$bgcolor = $_GET["color"]; // GET the user input

echo $bgcolor; // Print $bgcolor in HTML body page
echo "<div id='".$bgcolor."'>Text</div>"; // Print $bgcolor
    in the attributes value
echo "<div style='backgroud-color:'".$bgcolor.">Text</div>";
    // Print $bgcolor in CSS style
echo "<div id='name'>$bgcolor</div>"; // Print $bgcolor in
    the body of div tag
```

The program starts with user input received based on the *GET* method; the input is saved in the '$*bgcolor*' variable on Line 2. Subsequently, in Lines 3, 4, 5, and 6, the value of $*bgcolor* is printed in several contexts. For instance, Line 3 requires the program to print $*bgcolor* value in the page body and then print the variable.On Line 4, $*bgcolor* placed inside the HTML div attribute "*id*=". On Line 5, the variable value is printed using the Cascading Style Sheets (CSS) "*background-color*" attribute. Lastly, the HTML div tag within the body comprises the printed value of $*bgcolor*.In every context, the presented variable must have several sanitization techniques for making code safer. If the technique disregards the context of vulnerability presence, there might be more false positives. Presently, web applications have output code that references user input based on HTML commands to output a dynamic HTML page. Such an arrangement indicates an HTML context and facilitates web browsers to process the page differently. Different contexts have distinct properties and need varied defensive techniques to address XSS vulnerabilities (Shar and Tan, 2012; Gupta et al., 2018).For instance, the Hypertext Preprocessor (PHP) function "*htmlspecialchars*" is a common sanitization routine appropriate for sanitizing the initial print command on Line 3. Nevertheless, it cannot clean several vulnerability contexts (like print commands on Lines 4, 5, and 6) since every context needs different sanitization functions based on code presence (Shar and Tan, 2012; Bensalim et al., 2021). Therefore, evaluating particular location-specific characteristics of XSS attacks and creating sanitization routines is critical for evaluating the code for XSS exploits.*2.1.2. SQL Injection*

SQL is the most widely used database language, and it is used to communicate with database servers such as MSSQL, Oracle, and MySQL (Shar and Tan, 2012). Developers usually implement dynamic query building by combining strings to create meaningful SQL commands. Such commands, also known as queries, are created based on input values obtained directly from external sources while the query is executed. Such an implementation allows attackers to intervene since queries may be modified considering particular constraints specified by users. However, since such query building techniques are exposed to SQLi, many developers use parametrised query building techniques or employ stored procedures to improve security. Regardless, code might still be exploited if the correct processes are not followed.

Developers usually create SQL unvalidated query inputs, causing the most common and dangerous coding mistake. A dynamic SQL query is specified in the following PHP code:

$query = "SELECT * FROM user_info WHERE email = '$_GET["email"]' AND password = '$_GET["pass"]'";

Malicious users might take advantage of such an exposed query by using '*1*'='*1*' OR '*v*' value for email, allowing intended data exposure since the WHERE clause has the following condition:

*WHERE email = 'v' OR '1'='1' AND ...";*which is assessed as true.

2.2. Static Single Assignment (SSA) representation

The static single assignment (SSA) (Quiroga and Ortin, 2017) is an intermediate form of the source code considering that every variable is defined before using and only one assignment is performed; it is achieved by renaming variables such that they appear only once on the left side of an assignment operator. Fig. 1(a) exemplifies such a variable name change. Nevertheless, this form creates issues when several flow branches are combined. To address this concern, a ϕ function is employed where the precise version of variables cannot be determined (Rocha et al., 2020; Léchenet et al., 2020). Fig. 1(b) illustrates the usage of the ϕ function based on a basic graph sample.

Since variable assignment in the SSA format along with a ϕ function implemented at a merge point causes a higher time requirement to assess all code instructions. For example, the Low-Level Virtual Machine (LLVM) (Lattner and Llvm, , 2002) program for straightforward implementation requires more than 600 lines (Leißa et al., 2015), necessitating significant time for source code analysis. Hence, eliminating specific instructions incapable of changing program execution and reducing the ϕ function count in the SSA representation created a minimal SSA representation that it may shorten the time it takes to make any progress (Marashdih et al., 2021).

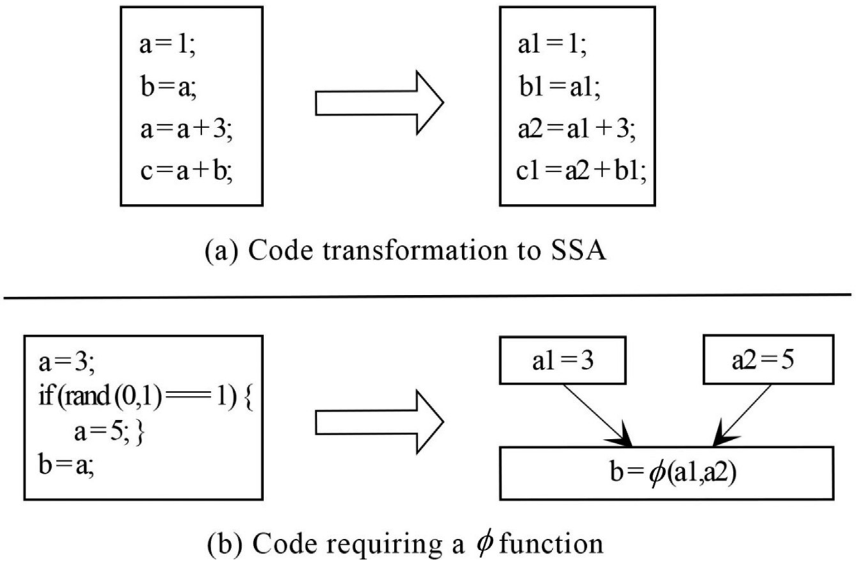


Fig. 1. SSA example code with ϕ function. (**a**) Changing the name of the variables. (**b**) Code based on ϕ function use and a simple graph example.

2.3. Vulnerability Prediction Models

Typically, vulnerability prediction entails analyzing if a source code file is susceptible. Rather than identifying the specific code line where a vulnerability exists, the idea is to alert software developers to areas of the code base that need special attention. As a result, we chose to work at the file level since it is also the scope of recent works (e.g. Walden et al., 2014; Fang et al., 2019) against which we would want to evaluate our method.

The vulnerability prediction model procedure may be summed up as follows (Walden et al., 2014): It first retrieves and identifies program modules from historical repositories, such as the National Vulnerability Database (NVD). As required, the module granularity may be changed to a source code file, object-oriented class, or binary component. Then, these modules are measured using features. Eventually, it marks a software module as vulnerable if it has at least one vulnerability, as determined by analyzing the vulnerability reports in the NVD. Lastly, it employs a particular classification method (i.e., the k-nearest neighbors algorithm) to train a model using the collected information. We might use the same metrics to evaluate a new program module and then use the trained model to predict whether the new module is vulnerable or not.

Machine learning (ML) is an area of Artificial Intelligence (AI) that enables computers to gain information on their own, without having to be explicitly programmed. Classification is thus a kind of data analysis involving the extraction of models defining data classes. Data classification consists of two steps: (1) *learning*, which involves building a classification model, and (2) *classification*, which involves using the model to predict class labels for given input data. This type of ML is called *supervised learning* because the class label of each training instance is known. On the other hand, in *unsupervised learning*, the class labels of each training attribute vector are not known, and it is not known in advance how many classes are being learnt (Alloghani et al., 2020).

Depending on the kind of learning (supervised or unsupervised), each classifier employs a ML algorithm to extract the information necessary to accurately categorize input data from the testing dataset. The kind of issue to be addressed and the nature of the data collection all have a role in the selection of an ML algorithm (Alloghani et al., 2020). Examples of supervised algorithms are linear regression (Money et al., 1982), Support Vector Machine (SVM) (Cortes and Vapnik, 1995), Decision Tree (Quinlan, 2014), Random Forest (Breiman, 2001) and k-Nearest Neigbors (KNN) (Duda and Hart, 2006). Examples of unsupervised algorithms are k-means (Jain et al., 1988), Isolation Forest (Liu et al., 2008), One-Class SVM (Manevitz and Yousef, 2001), Elliptic Envelope (Rousseeuw, 1984), and Principal Component Analysis (PCA) (Jolliffe, 2002).

Deep Learning (DL) is a sub-field of machine learning that focuses on learning successive layers of increasingly meaningful representations of the input data. The learning is performed using Artificial Neural Networks (ANN) composed of multiple layers, and input data travels sequentially across the layers, which transform the data during processing. The presence of several layers allows the model to understand essential and complex requirements using relatively straightforward and non-specific requirements (from prior layers) (Goodfellow et al., 2016).

Convolutional Neural Networks (CNN) and Recurrent Structures are two widely used algorithms in DL for Natural Language Processing (NLP) (Goldberg, 2017; Hanif et al., 2021). These layers are not self-contained, yet they are important feature identifiers. CNN belongs to the feedforward network category and is capable of extracting local data aspects. Recurrent Structures (RS) can process word sequences and every word in prior sequences. These are suitable for sequential datasets and have demonstrated excellent results in several NLP applications (Sutskever et al., 2014; Bahdanau et al., 2014).

Convolutional Neural Networks (CNN): CNNs have a framework that places them in the feedforward model category; such networks are excellent at discovering local pattern recognition using extended sets with different element counts. The primary concept concerning CNN layers is to use a non-linear learning routine (filter) corresponding to a window size k. Window movement with every time step allows processing the entire sequence to output a scalar quantity, determining the features corresponding to the window (Goldberg, 2017). Lastly, the created vectors are merged using pooling layers, and the resultant vector is representative of the entire sequence.

Nevertheless, CNN is unable to identify the global input sequence in cases where the local input sequence is indicated. Therefore, this framework is specifically suited to computer vision scenarios like object identification or image categorisation (Krizhevsky et al., 2012; Simonyan et al., 2014).

Recurrent Structures: Different forms of RS include Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), and Bidirectional Recurrent Neural Network

(BiRNN). RS-related networks performed well in several sequence data modeling tasks. The RNN is a modified version of the classical Feedforward Neural Network (FNN); the units in the RNN feature a self-connection that facilitates the transmission of data from one time step to the next. However, as the number of time steps rises, there is a chance that the RNN network may have issues with vanishing and exploding gradients. A variety of solutions were implemented to address these problems. However, LSTM is the most effective model for replacing basic units with a memory block. A memory block consists of a memory cell and many gating operations. The data is kept in memory cells and controlled by gating processes. However, this memory block is computationally costly since it includes so many parameters. Later, GRU was created; it is a streamlined variant of LSTM that exhibits the same performance as LSTM. BiRNN is an advanced technique in DL architecture that can capture the long-range relationships of temporal patterns in sequence data in both forward and reverse directions.

For web application vulnerability prediction models, source codes must be transformed into a proper form to employ data-driven approaches for automatic vulnerability detection. This form should represent the source code's syntactic and semantic information and be suitable for further analysis in different machine learning and neural network algorithms. The next section shows the related proposed studies for code representation on different machine learning and neural network algorithms.

3. Related Works

In web applications, code build characteristics examine the control or data flow graph, as well as extraction through dynamic and static code analysis. A collection of static code properties was extracted by several research studies (Shar and Tan, 2013; Shar et al., 2014).

Text-mining techniques are used to identify program files classified as a collection of bag-of-words (Gupta et al., 2015). A text mining feature extraction process was employed by Walden et al. (2014) to extract a set of tokens as features to predict XSS vulnerabilities from the dataset that was prepared manually by the authors. Their results indicated that their method had significantly better performance versus software metrics features.

Code vectorisation can be defined as prior research pertaining to machine learning-based code auditing. The most popular technologies and classic code analysis methods include Control Flow Graph (CFG) and AST. They help provide abstract code logic as well as serialisation results, which are deemed appropriate to be fed as input for the machine learning models. Alon et al. (2019) developed a CFG/AST-based data flow technique. However, they all convert the original structure into a new structure. Having to comprehend the new transformed data format adds to the researchers' workload.

Fang et al. (2019) proposed TAP, which is regarded as a useful approach for tokenizing PHP codes. By accounting for the rules that are set by the authors, auditing of different vulnerabilities was completed on the Software Assurance Reference Dataset (SARD) (Database et al., 2021).

A novel feature extraction algorithm was suggested by Gupta et al. (2015) to extract the basic as well as contextual features based on the web applications' source code. The input/output patterns were analysed by Li et al. (2020) as well as constructed an algorithm to recognise the vulnerable paths in the programs. Their approach improved the efficiency of mining the input/output context and constructed exact feature sequences for presentation.

Li et al. (2021) perform slicing on the control flow graph (CFG) of the programs. A bidirectional GRU network is employed as a classifier to predict the vulnerabilities. Their work was enhanced with the inter-procedural dependency analysis in their recent approach (Li et al., 2021) to produce sliced source code. They converted source code into an IR based on the LLVM compiler, expressing semantic and syntax information in an SSA form. Then, in order to learn high-level characteristics, the bidirectional GRU neural networks are used. However, they are deemed to be relatively complex because of the inter-procedural dependency analysis. Table 1 illustrates the related studies methodology and evaluation indicators.

IR is employed in both of the recent studies (Li et al., 2021; Li et al., 2021), to simplify the source code for analysis, like in the form of SSA. Meanwhile, an MSSA construction algorithm was introduced by Braun et al. (2013) to prepare a minimal SSA form that helps in reducing the SSA instructions for analysis. This algorithm can reduce the number of instructions in the SSA form, which will help to handle the issue of the long-term dependency between the elements of the programs.

Table 1
Related studies methodology and evaluation indicators. R: is the recall, P: is precision, and Acc: is the accuracy.

Authors	Features	Vulnerability type	Algorithms	Dataset	Performance
Shar and Tan (2013)	Static attributes	SQLi, XSS	C 4.5, NB, MLP	User defind dataset	R: 78%, P: 80%, Acc: 90%
Shar et al. (2014)	Static/Dynamic attributes	SQLi, XSS	LR and MLP	User defind dataset	R: 77%, P: 75%
Gupta et al. (2015)	Static attributes and HMTL context	XSS	Random Forest, SVM, J48	SARD	R: 85%, P: 88%, Acc: 88%
Walden et al. (2014)	PHP tokens and software metrics (i.e. LOC, cyclomatic complexity)	Code Injection, CSRF, XSS, Path Disclosure	Random Forest	User defind dataset	R: 57%, P: 67%, Acc: 71%
Fang et al. (2019)	TAP	General	LSTM	SARD	R: 93%, P: 92%, Acc: 97%
Li et al. (2021)	Slicing the CFG	General	BiGRU	SARD	R: 88%, P: 83%, Acc: 89%
Li et al. (2021)	Intermediate representation (SSA)	General	BiGRU	SARD	R: 95%, P: 97%, Acc: 97%
Li et al. (2020)	PHP Opcode and tokens	XSS	BiLSTM	SARD	R: 92%, P: 92%, Acc: 92%

In addition, most of the previous studies (Shar et al., 2014; Shar and Tan, 2013; Walden et al., 2014; Li et al., 2021; Li et al., 2021; Fang et al., 2019) did not consider the vulnerability context during the feature extraction of the programs. Taking into account that the vulnerabilities exist in such an HTML context will give the model more information about the context of the vulnerabilities and improve the performance of the prediction results.

Algorithm 1: Pseudocode of the preprocessing stage.

```
Input: $DS \leftarrow PHPdataset$
Function Preprocessing($DS$):
    Generate the AST of the program under test.
    Convert AST to SSA form.
    Produce a minimal SSA (MSSA) form by reducing the number of
      instructions and $phi$ in SSA form.
    Paths ← Generate program paths from MSSA
    while $Path \in Paths$ do
        Extract the instructions from the path.
        Assign the vulnerability source.
        Remove the unrelated instructions for the vulnerability
          source.
        Assign a unique feature for each vulnerability context
          (execution of the vulnerability source statement).
    endwhile
    Features ← A list of all the extracted features from the paths as
      rows.
End Function
```

Furthermore, all the previous studies extract the features from the source code, which contains all the information about the flow of the program. However, there are a number of features that are unrelated to the flow of the vulnerability in the programs. Removing unrelated features from the programs' extracted features will help to improve the performance of the model (Zhang et al., 2018; Jin et al., 2005; Hong et al., 2020). Spens et al. (2018) stated that reducing the amount of irrelevant features considered for classification increases the classifier's accuracy, whereas Ali et al. (2019) discovered that irrelevant features reduce a classification process's precision rate.

4. Methodology

This section presents our proposed approach for predicting XSS and SQLi vulnerable files, where the program code is turned into an AST. Then the AST is directly converted to an SSA form, which involves optimising the SSA form that has to be pruned, along with the MSSA form. The initiation of the feature extraction method includes traversing of the MSSA form by employing forward slicing to create the program paths. During the step of path generation, each of the instructions is stockpiled based on its type as a token (feature). Moreover, in each path, the context of the vulnerability as well as the sanitization functions are determined to provide additional knowledge about the flow of vulnerability within the program for the model. A filtering method is carried out for the extracted features, and these features are used as input for several ML algorithms, which generate the prediction result. Fig. 2 shows the flow of the proposed model, and Algorithm 1 displays the pseudocode of the proposed preprocessing stage to extract the features from the program code.

4.1. Preprocessing

The notion of data preprocessing affecting the outcome of an ML task is widely accepted. The proposed data preprocessing method includes a feature generator, selection, and filtering. As a preprocessing stage, the proposed approach reduces dimensionality, removes unnecessary data, increases learning accuracy, and improves understanding of the results. This method starts by converting the source code into an AST, followed by presenting the tree in an MSSA form. Then, the feature extraction process started by generating the program paths from the MSSA form and eliminating the features that were unrelated to the vulnerability flow among the programs. The proposed method can identify relevant features for further processing.

4.1.1. Tree Generator

Abstract syntax trees (ASTs) are a way to show how programs work. They are often made by compiler frontends. These trees show how code can be broken down into its syntactical parts in a logical way. The trees are abstract in the sense that they don't include all of the specifics of how to write a concrete program. They merely demonstrate how programming constructs are stacked to create the final code. It makes no difference whether the variables are defined as elements of a declaration list or in a stream of declarations. This is a small point in the formulation that doesn't change how the ASTs look.

There are two types of nodes in an abstract syntax tree. Operators like assignments and function calls are represented by inner nodes, whereas operands like constants and identifiers are represented by leaf nodes. Fig. 3 depicts the AST for Listing 2 s running example as an example.

Listing 2: An example of generated features.

```php
<?php
$x = 10;
$y = 0;
if ($x == 5) { $y = 3; }
else { $y = x; }
print $y;
```

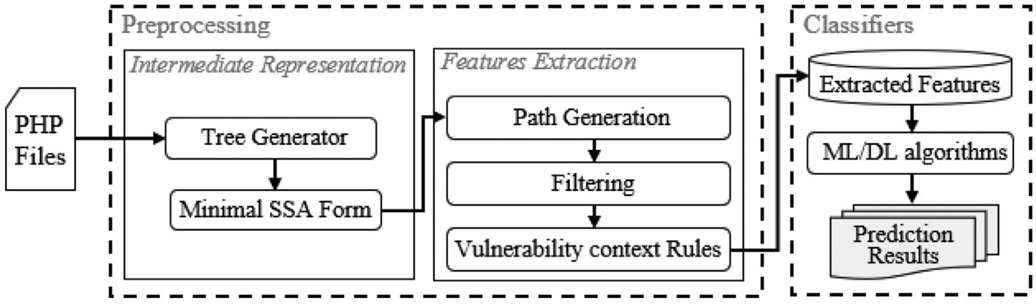


Fig. 2. The flow of the proposed model.

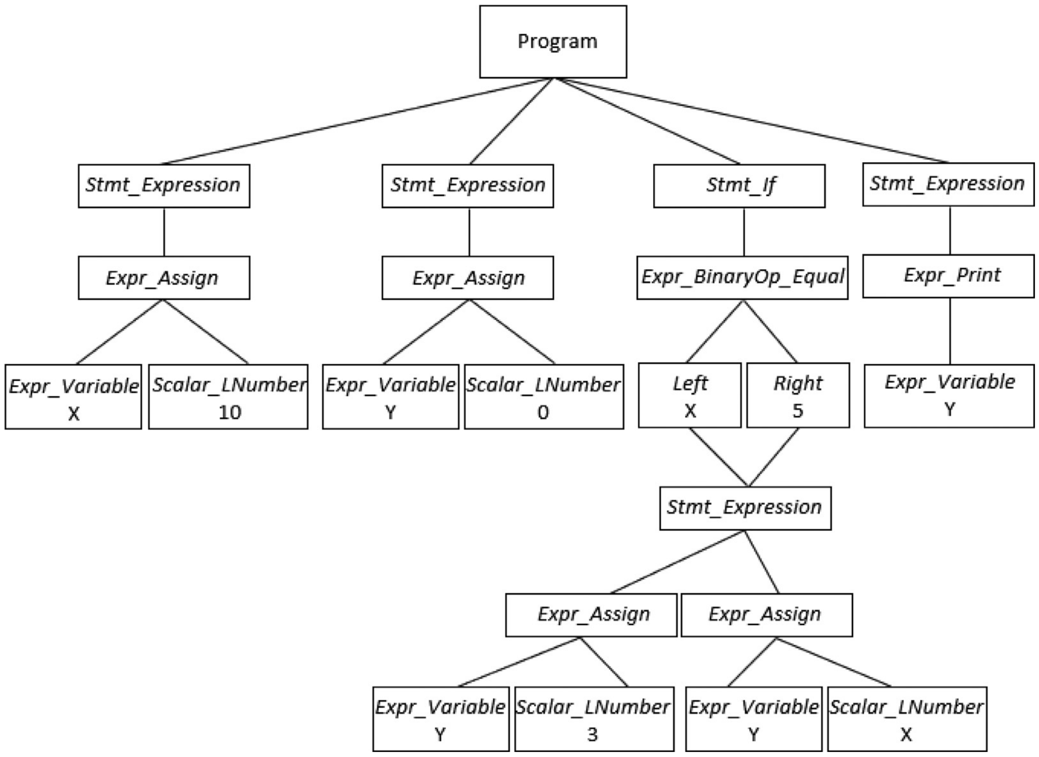


Fig. 3. AST for code in Listing 2.

Semantic information, such as a program's control flow or data flow, is missing from AST (Backes et al., 2017). Thus, they are unable, for example, to reason about data flow inside a program, leading to the requirement for extra structures. Therefore, we convert the PHP source code to AST in order to be easily converted to SSA form in the next section. PHP-Parser (Popov, 2020) has been used to generate the AST which carry out lexical and grammatical analysis of the programs.

4.1.2. Minimal SSA Representation

The SSA (Schardl et al., 2019) form is a type of intermediate representation that ensures that each variable has only been written and defined once.

We have employed Braun et al. (2013) algorithm wherein on-the-fly optimisations have been employed with an SSA compiler for reducing the ϕ functions in the form. Initially, we used a mathematical simplification, in which the constructors of the IR node get exposed to peephole optimisations as well as output simpler nodes, wherever it would seem feasible. Based on an example, the mathematical expression X–X has been found to lead to zero every time. Moreover, common subexpression elimination, reuse of the current values recognised employing local values and an assessment of the constants in compilation are carried out-for instance, 2×3 can also be optimised as 6. Finally, the local variables could also possess unnecessary arguments that need to be eliminated (for example, X = Y). Assignments of this form are not needed for the SSA form; thus, it can be deemed feasible with regard to directly using the right-hand side value. Fig. 4 shows the effectiveness of such optimisations.

Fig. 4(b) illustrates the minimal static single assignment (MSSA) form before the on-the-fly optimisations are applied. After enabling the optimisation, the initial difference can be observed upon constructing value V_2. Such a case will return any comparison with zero as false; thus, during arithmetic simplification, the value is simpler. The next step will show a comparison with zero returning false being converted due to constant propagation. Because of the false state at the jump condition, it is feasible to exclude the code that is inside the "then" block. As for the "else" block, assigning V_0 to Y conducts a copy propagation. In the same way, V_5 disappears and, eventually, the code goes back to V_0. The code's optimised SSA form is depicted in Fig. 4(c). Upon observing the example, it can be seen that the run-time optimisations recommended by Braun et al. (2013), have the potential to decrease the SSA form's instruction count and ϕ functions. The following section illustrates the features extraction method after finding the PHP source code's minimal SSA form.

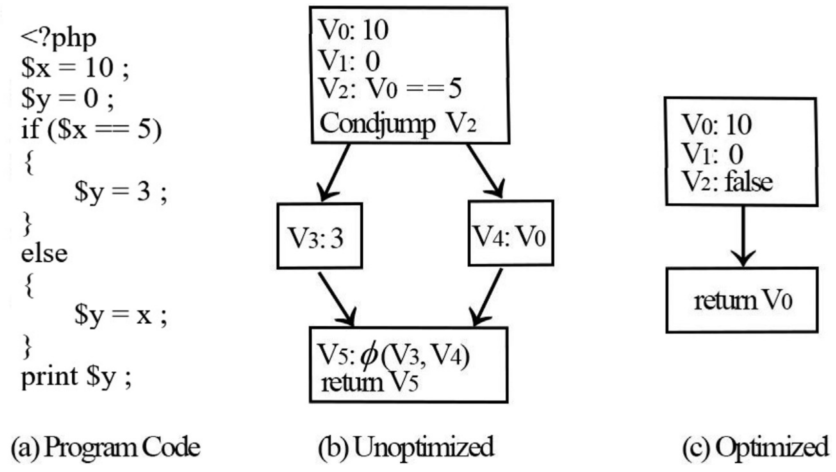
4.1.3. Features Extraction

Fig. 5 shows the structure of the proposed method for extracting the features from the MSSA form. The proposed method is based on three stages: (1) *path generation*: generates the entire paths in the MSSA form; (2) *filtering*: reduces the features in the generated features sequence that are unrelated to the vulnerability flow; and (3) *vulnerability context rules*: assigns unique features for each vulnerability context based on certain rules.

Path Generation: A few instructions about a block's type, variables, and expressions are contained in each block in the MSSA form. The "type" value of each node is utilised as a token to create a structure stream. Since the MSSA form consists of a start block and an end block, it is useful to know when each path begins and ends. Slicing (Weiser, 1984) was performed according to each program block to generate a path of the program. A forward slice was carried out since it always begins as an initial block from the first block. It will initially store each block's type in an array related to the current path. The functions within the MSSA structure are specified as a separate group of blocks. After the generator locates a function call within the block, it will proceed to the function blocks and follow that function's inside instructions. Each instruction type within the function is kept in the array of features for that particular path. The termination of that function is contained in the function block's last block. With this final step, the generator will return to the function call instruction. Upon the termination of the path generator, a stream will be formed using a backslash to concatenate all the generated paths of a program under test. The next stage shows the filtering method for eliminating the generated path blocks that are unrelated to the vulnerabilities.

Filtering: The results of the path generation contain all the blocks generated from the MSSA as a set of stream paths separated by backslash. Nevertheless, several blocks are not related to the flow of the vulnerability in the programs. Therefore, we removed the blocks that were irrelevant to the flow of the vulnerabilities in the programs. The removal stage of these blocks is based on four steps:

1. Tracing the generated paths: tracing of the generated paths should go through the features in the order they were generated. Thus, the source of the vulnerabilities would be identified before the sanitization functions or sink statements. In this step, the process of tracing the paths that were made began with the first identified block and went in order from there.
2. Identifying the sources of vulnerabilities: the generated paths are traced to find any source (user input) in the programs. Once the source is found, the value of that source is stored externally in a database.
3. Removing comment blocks: Developers frequently use comments to make their code more readable and easier to update in the future.However, some comments might contain some information about the variables and how they are used. If a comment contains the name of the vulnerable source, that might lead to storing these comments in the database as a value related to the sources, and it will increase the features that are



```
<?php
$x = 10 ;
$y = 0 ;
if ($x == 5)
{
      $y = 3 ;
}
else
{
      $y = x ;
}
print $y ;
```

Fig. 4. Generating the minimal SSA form of the sample code in Listing 2 (**a**) Source code; (**b**) SSA form; (**c**) SSA form after on-the-fly optimisations.

unrelated to the vulnerabilities. So, we took out all of the comment blocks from the paths that were generated to fix this problem.

4. Regular expression matching source: each block value is evaluated based on a regular expression to check if one of the identified sources in the database matches the current block value. If it is matched (which means it contains the source of the vulnerability in that block, whether in an assignment of another variable, or in function parameters, or in an array), the filter keeps this block without elimination. If it doesn't match, which means it doesn't have any source of vulnerability, the block is removed by the filter.

Listing 3: An example of generated features before and after the filtering step.

```
<html>
<body>
<?php
$tainted = $_GET['UserData'];
$var = 1;
$tainted = htmlspecialchars($tainted,ENT_QUOTES);
echo "<div id = ".$tainted." />";
?>
</body>
 </html>
 - - - - - - - - - - - - - - - - - - - - - - - - -
//  Generated features before filtering (12 tokens):
//  Terminal_Echo(LITERAL)  Expr_ArrayDimFetch(tainted)
   Expr_Assign  Expr_Assign(tainted)  Expr_FuncCall(
   htmlspecialchars/ENT_QUOTES) Expr_BinaryOp_Concat_.
   Expr_Assign Expr_Binary Concat.(tainted) Terminal_Echo(
   LITERAL) Terminal_Echo() Terminal_Echo(LITERAL)
   Terminal_Echo(LITERAL)
 - - - - - - - - - - - - - - - - - - - - - - - - -
 // Generated features After filtering (6 tokens):
 // Expr_ArrayDimFetch(tainted)  Expr_Assign(tainted)
   Expr_FuncCall(htmlspecialchars/ENT_QUOTES)
   Expr_BinaryOp_Concat_.(LITERAL) Expr_BinaryOp_Concat_.(
   tainted) Terminal_Echo()
```

The results of the proposed filtering method are a set of the paths that contain all the blocks that are related to the flow of the vulnerabilities in the programs. The block type and value are concatenated and used as features to represent each block. Listing 3 presents a sample of PHP code with features before and after the filtering method.

The PHP code in Listing 3 begins by getting user inputs based on the *GET* method and assigning the input to the '$*tainted*' variable in Line 4. A second variable is provided with a value of 1 in Line 5. Next, a sanitization function *htmlespecialchars* was employed using the *ENT_QOUTES* arguments to clean up the '$*tainted*' input. Subsequently, Line 7 contains code to output the value of $*tainted* variable. The example code declares the '$*var*' variable on Line 5 and assigns a value of 1. However, this code fragment is not related to the vulnerability flow in the program. Thus, the filtering method removed the features related to this line of code. Additional feature removal is implemented for this code for *Terminal_Echo(LITERAL)* feature associated with the HTML code on Lines 1, 2, 11, 12, 13, and 14 because they are static attributes not associated with vulnerability flow.

The feature results in Listing 3 show that the generated features before filtering contain 12 features for the example program code. On the other hand, the filtering method reduced the number of generated features by 50% without missing any features related to the vulnerability flow in the program.

Currently, the sink (printing statements of the vulnerability source) is tokenized by the type of the block and the value. However, the program might contain vulnerabilities in different printing statements of HTML contexts where all of them are currently tokenized the same. Those statements should be identified and tokenized with unique features in order to provide the model with complete information about the vulnerability context later on.The next step explains the vulnerability context rules that are used to tokenize each case where a vulnerability exists in the source code.

Vulnerability Context Rules: To tag a vulnerability, a tainted variable (i.e., any variable that can be changed by an outside user and, thus potentially poses a security risk), should be consumed before conducting sanitization or filtering. Sources, sinks, and sanitizers constitute the three primary components for static taint analysis (Maskur and Asnar, 2019).

Sources indicates that attackers might employ it for embedding malicious scripts. Normally, PHP web applications are attacked using HTTP request properties, session information, and file uploads.

Sinks are defined as points or end functions that implement actions that can possibly be attacked. Sinks in PHP web apps

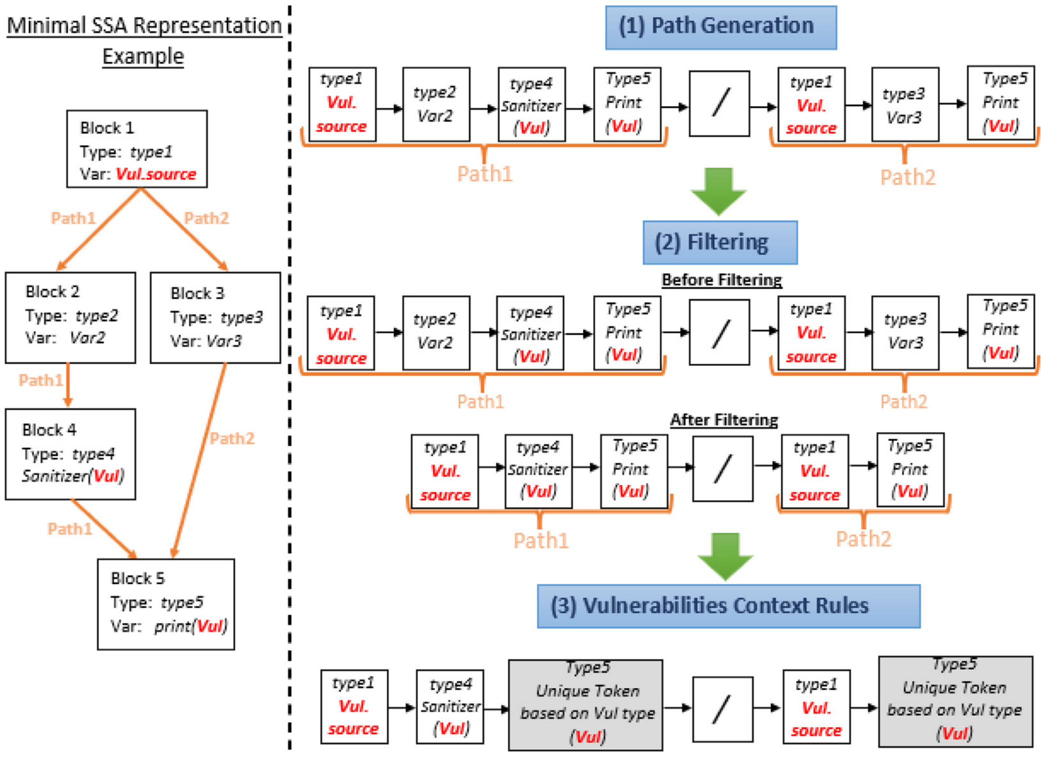

Minimal SSA Representation Example
Block 1 Type: type1 Var: Vul.source
Path1
Path2
Block 2 Type: type2 Var: Var2
Block 3 Type: type3 Var: Var3
Block 4 Type: type4 Sanitizer(Vul)
Block 5 Type: type5 Var: print(Vul)
(1) Path Generation
type1 Vul. source
type2 Var2
type4 Sanitizer (Vul)
Type5 Print (Vul)
type3 Var3
(2) Filtering
Before Filtering
After Filtering
(3) Vulnerabilities Context Rules
Type5 Unique Token based on Vul type (Vul)

Fig. 5. The proposed feature extraction method steps are applied to a MSSA representation example.

include file methods, SQL query methods, and operating system methods, among others.

Sanitizers are code fragments which filter a malicious code and serve to protect a source code (Backes et al., 2017). PHP web application consists of several sanitizers (i.e., the SQL query and HTML escape functions).

PHP sources (input expressions) are built using global *$_REQUEST*, *$_POST*, *$_GET*, *$_SESSION*, *$_FILES*, *$_COOKIE*, and *$_SERVER*. Therefore, a malicious script could be included and PHP sources could be vulnerable should the malware reach a sink statement and a proper sanitization method is not used to sanitize the variable.

OWASP (OWASP, 2021) is an Internet-based forum which provides free knowledge, techniques, software, documentation, and technologies concerned with web applications' security aspects. To prevent XSS attacks, the OWASP XSS cheat sheet series project suggests a series of rules to correctly sanitize or circumvent untrusted inputs in different contexts. Therefore, the XSS problem is solved theoretically. The existence of the injection for SQLi could only be in one location on the website, which is inside the executing SQL queries' sink statements (i.e., *MySQL_query* or *MySQLi_query*).

The OWASP XSS prevention rules (OWASP, 2021) were used to determine each program's type of sink. Table 2 displays the OWASP categories of XSS context along with the new generated features.

As shown in Table 2, the classification of each vulnerability context is in accordance with the vulnerability location in the source code. The context number for every sink statement was designated based on the OWASP classification context. On the other hand, each of the provided contexts requires different sanitization functions to make it safe. Consequently, for each PHP sanitization function that was located in the generated paths, we added the function name as well as the arguments and the block type. This is to pinpoint the sanitization functions suitable to the found contexts. Listing 4 provides an example of a PHP program's generated features following the representation of the vulnerability context.

The *htmlespecialchars* sanitization function on Line 6 is recorded using the *Expr_funcCall* type; next, the arguments and function, i.e., *ENT_QUOTES* and *htmlespecialchars*, are recorded. Line 9 contains code for the second OWASP context; as a result, the print statement is tokenized as *Expr_Assign(C2)* as a unique feature. Therefore, the generated features have all the features that are linked to the vulnerability. The context of the vulnerability and the sanitization functions used are also tracked, which gives the model learning more useful information.

4.2. Classifiers

Researchers have used different machine-learning and DL algorithms in their studies. A feature set with different algorithms builds different prediction models and may have varied performances (Gupta et al., 2015; Medeiros et al., 2015). These algorithms were used to extract knowledge from a set of labeled data. This section presents the algorithms that were studied as classifiers to identify the best approach to classifying candidate vulnerabilities.

4.2.1. Random Forest

A random forest is comprised of random trees. A random forest classifier obtains its prediction by averaging the forecasts of individual decision trees. This avoids overfitting of the model. Each random tree may also be trained on a distinct subset of the training set to further reduce the correlation across decision trees.

4.2.2. Linear Regression

Linear regression is used to estimate a linear hypothesis function between the output and input variables. The goal of linear

Listing 4: An example of generated features following the representation of the vulnerability context.

```
<html>
<body>
<?php
$tainted = $_GET['UserData'];
$var = 1;
$tainted = htmlspecialchars($tainted, ENT_QUOTES);
echo "<div id = ". $tainted ." />" ;
?>
</body>
</html>
----------------------------------
//  Generated Features:
//  Expr_ArrayDimFetch(tainted)  Expr_Assign(tainted)
    Expr_FuncCall(htmlspecialchars/ENT_QUOTES)
    Expr_BinaryOp_Concat_.(LITERAL) Expr_BinaryOp_Concat_.(
    tainted) Terminal_Echo(C2)
```

Table 2
OWASP categories of XSS context.

Context #	Description	Example	New features
C1	HTML encoding prior to saving untrusted information inside an HTML element.	*echo <span> XSS Malicious Script </ span>*	Terminal_Echo (C1)
C2	Attribute encoding prior to saving untrusted information inside HTML attributes.	*echo <input attr="Malicious script">*	Terminal_Echo (C2)
C3	URL encoding prior to saving untrusted information inside HTML URL variables.	*echo <a href=" http:// www. __.com?var = Malicious script"> </a >*	Terminal_Echo (C3)
C4	URL verification and validation processes; permitting only HTTPS URLs and encoding attributes.	*echo <a href="Malicious URL">click here</a>*	Terminal_Echo (C4)
C5	CSS encoding and strict validation before saving untrustworthy information inside HTML style attributes.	*echo <div style="width: Malicious script;"> Selection</div>*	Terminal_Echo (C5)
C6	Ensuring that all JavaScript parameters are quoted.	*echo <script>var currentValue='Malicious script ';</script>*	Terminal_Echo (C6)

regression is to predict future data by fitting a straight line across the existing data.

4.2.3. SVM

SVM is a classification technique that involves displaying raw data as points in an n-dimensional space (where n is the number of features you have). The data may then be easily classified since each feature's value is then connected to a specific coordinate. The data may be divided into groups and shown on a graph using lines known as classifiers.

4.2.4. Decision Tree

A decision tree is a tree structure that looks like a flowchart, with each internal node representing a test on an attribute, each branch representing the test's output, and leaf nodes representing classes. The benefit of employing decision trees in data classification is that they are straightforward to learn and interpret (Ashari et al., 2013).

4.2.5. k-nearest neighbor (KNN)

The KNN is a classification approach based on the majority class of an object's k-nearest neighbors. The KNN is a kind of lazy learning in which the function is only estimated locally and all computation is delayed until classification (Ashari et al., 2013).

4.2.6. Neural Networks Classifiers

Cyclic neural network (Hu, 2019) is a prominent model at the moment, and with good reason. Multi-layer back propagation neural networks improve the transverse connections between hidden layer units. For the neural network to retain its memory, a weight matrix is used to transmit the value of a previously recorded neuron unit to the current neuron unit. Because of this, RNN has had a major impact on text classification. There is a gradient explosion or gradient disappearance if the RNN can't remember the data from a long time ago or a long time ago. As a result, RNN-based deformation structures have been suggested by numerous researchers. As shown in Fig. 6, four neural network models of RNN structure were built based on this information.

- *LSTM*: RNNs have a special structure called Long Short-Term Memory (LSTM). On the basis of LSTM, memory units are added to each neuron unit in the hidden layer. This makes it possible to control the memory information about time series. Each time it goes through a controllable gate (forgetting gate, input gate, candidate gate, and output gate) in the hidden layer, the

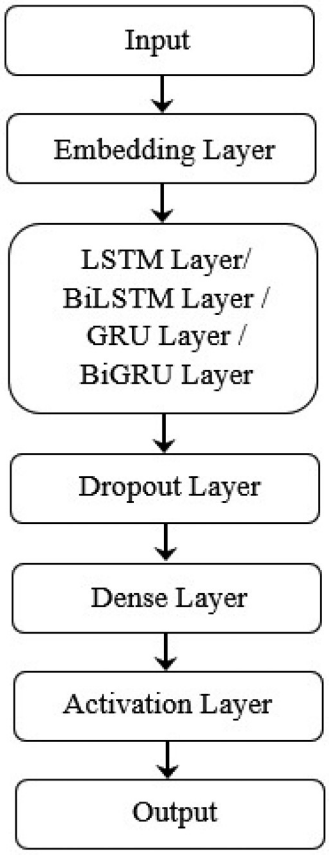


Fig. 6. The general architecture of the neural network model for vulnerability prediction.

amount of information that is remembered and forgotten about past and current information can be changed. This gives LSTM the ability to remember things for a long time.

In this paper's LSTM model, the first step is to get to the embedded layer's processing data. The next layer is the LSTM layer, which has a 16-dimensional output. As was already stated, deep learning networks often have problems with fitting, so a dropout layer is added after the LSTM layer to reduce the effects of overfitting. The dropout layer is followed by a dense layer with an output dimension of 1. Lastly, the predicted results are handled by the *Softmax* function.

Some of the settings for learning LSTM are: batch size of 256, number of LSTM neurons of 128, output dimension of 256, default learning rate of 0.01 when using ADAM for training, maximum sequence length of 350, and dropout of 0.2. After a lot of tests, the author found that these values for the parameters were pretty much efficient for the model.

- *BiLSTM*: LSTM mostly remembers the previous input data, but it is dependent on the following input data for the majority of the time when making a prediction. BiLSTM has been suggested by researchers. BiLSTM trains two rather than one LSTM on the input sequence when all time steps are available. Using the BiLSTM, the first input sequence is replicated, and the second input sequence is reversed. As a result, it is possible for the network to learn quickly or even more effectively using this method of processing.

  The network layout of the BiLSTM model built in this paper is essentially identical to that of the LSTM model, except we replace the LSTM layer with the BiLSTM layer.

  The batch size is 256; the number of BiLSTM neurons is 128; the output dimension is 256; the default learning rate is 0.01 when using ADAM for training; and the maximum sequence length is 350; dropout is 0.2. These are the parameters that the author discovered to be pretty accurate after a significant number of tests.
- *GRU*: The network structure of LSTM is complicated, particularly with numerous gates, which leads to inefficiency and impacts prediction findings. On this premise, GRU, a more practical and simpler structure form that enhances efficiency, combines input gate and forgetting gate, and merges ST and CT, namely, memory unit and output unit, was presented in 2014. As a result, GRU may be seen as an upgrade to RNN and LSTM, as it can resolve the lengthy dependency issue in RNN. At the same time, its network structure is simpler than that of the LSTM network (Bansal et al., 2016).

  Referring back to the notion of building an LSTM model, the GRU model built in this paper is essentially the same as the LSTM network structure, except we replace the LSTM layer with a GRU layer.

  The batch size is 256, the number of GRU neurons is 128, the output dimension is 256, the default learning rate is 0.01 when utilizing ADAM for training, the maximum sequence length is 350, and the dropout is 0.2. These are the parameters that the author discovered to be pretty accurate after doing a significant number of tests.
- *BiGRU*: As with the majority of cyclic neural networks, GRU only examines the initial training dataset and disregards subsequent input datasets. In certain instances, the output of the present instant is tied to both the prior state and the future state. For instance, the predictions of absent words in a phrase must evaluate not just the preceding paragraph, but also the paragraph after it, in order to make a true context-based decision.

  The BiGRU is comprised of two GRUs that are overlaid, and the output is reliant on the state of both GRUs (Luo et al., 2017).

  The batch size of learning BiGRU is 256, the number of BiGRU neurons is 128, and the dimension of output is 256. When utilizing ADAM for training, the default learning rate is 0.01 and the maximum sequence length is 350. The dropout value is 0.2. After conducting a significant number of tests, the author determined these parameters to be reasonably accurate.

5. Experiments and Evaluation

In this section, details regarding the datasets as well as the evaluation metrics employed for the assessment have been provided. The assessment stage starts with performing experiments on various feature extraction methods and then making a comparison based on their performances. Second, a comparative experiment was carried out on a different classifier that had the best performance (i.e., the LSTM classifier). Finally, the performance of our proposed model was compared with the studies carried out on the same datasets in order to highlight its excellent performance.

5.1. Research questions

The goal of the experimental assessment was to answer the following questions:

- Is the proposed feature extraction method able to enhance the prediction of input validation vulnerabilities from source code? (Section 5.5)
- How well is the performance of the generated features in different classifiers? (Section 5.6)
- How effective and precise was our proposed model versus the state-of-the-art models? (Section 5.7)

5.2. Dataset

The Software Assurance Reference Dataset (SARD) (Database et al., 2021) has been employed by numerous studies (Li et al., 2020; Li et al., 2021; Fang et al., 2019) in order to assess the models with regards to the prediction of web application vulnerabilities. Accordingly, we incorporated the SARD dataset into our experiments for predicting input validation vulnerabilities because it includes all of the code files that refer to a user-input as a vulnerability label in the many HTML contexts where it is mentioned. To evaluate the performance of each method, this dataset (SARD) is a better option than, for example, Kaggle (https://www.kaggle.com/datasets). We can't utilize these repositories for our research since they just give information about vulnerabilities and don't provide the source code.

SARD includes software-generated PHP tests to predict SQLi and XSS vulnerabilities (Stivalet et al., 2016). These tests are then organised as being unsafe (vulnerable) and safe (not vulnerable). In SARD, 10,080 files are included in the XSS dataset, of which 4352 files (# of unsafe) are regarded to be unsafe files, while 5,728 files (# of safe) are regarded to be safe files. However, a total of 9144 files are included in the SQLi dataset, of which 912 files (# of unsafe) are regarded to be unsafe files, while 8232 files (# of safe) are regarded to be safe files. In our experiment, the dataset has been segmented into training, validation, and test datasets that have a ratio of 7:1:2. The SARD dataset details for XSS and SQLi have been illustrated in Table 3.

5.3. Experimental Environment

The preprocessing stage starts by converting the source code into an AST, which presents the program in a tree structure of nodes. We used the PHP-Parser (Popov, 2020) tool for grammatical and lexical analysis, and then we used the Braun et al. (2013) algorithm to convert the AST to minimal and pruned SSA form, which reduces the number of instructions in the SSA form and the *phi* functions.Once the MSSA form is produced, a features extraction process begins to convert the source code into features and eliminate the features that are considered unrelated to the flow of the vulnerability in the programs. Also, the HTML context was taken into account by replacing sensitive statements with features that were unique to each statement. This gave the model more information to learn the context of the vulnerability in the programs.

Afterwards, the generated features are sent to the model classifiers as inputs, Word2vec (Gensim, 2022) is used to encode the features into fixed-length vectors. Word2vec uses a single-layer neural network to map one-hot word vectors to distributed word vectors. Training turns the features into vectors of continuous tokens. Token vectors utilized in the experiment have a size of 30 and have decimal ranges ranging from 0 to 1. All classifers have been trained based on the publicly available Google Colaboratory[1], a free Jupyter notebook environment that can be run on the cloud. On the back end, Keras has been used as a DL package along with Tensorflow.

Table 3
SARD dataset for XSS and SQLi.

Vulnerability	# of Safe	# of Unsafe	Total
XSS	5728	4352	10080
SQLi	8232	912	9144

[1] https://colab.research.google.com/notebooks/welcome.ipynb

5.4. Evaluation Metrics

In order to measure our model's performance, several metrics are used for recording incorrect as well as correct predictions. True Positives (TPs) can help to assess if the approach accurately determines the code to be deemed vulnerable; True Negatives (TNs) can help assess if the approach accurately determines the code to be deemed invulnerable. False Positives (FPs) help assess if the approach incorrectly determines the code to be deemed vulnerable, and False Negatives (FNs) can help assess if the approach incorrectly determines the code to be deemed invulnerable. Naturally, the optimal tool offers the highest TN and TP values; concurrently, it also provides the lowest FP and FN values. In order to characterise prediction models, a combination of these metrics can be employed, as presented below:

- *Precision* signifies the true classification with regards to vulnerabilities, determined by employing the following expression:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

- *Recall* indicates the likelihood that an identification tool categorizes present vulnerabilities in the vulnerability category, as specified as follows:

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

- *Accuracy* represents the capability to accurately segment vulnerable and clean samples.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

- *F1-score* pertains to precision and recall and is expressed in terms of the harmonic mean of these metrics. It denotes the level of balance with regard to these metrics.

$$F1 - score = \frac{2 * (Precision * Recall)}{Precision + Recall} \tag{4}$$

- *Receiver Operating Characteristic (ROC)* curve (Fawcett, 2006) is employed for assessing the diagnostic ability of binary classifiers. The y-axis represents the True Positive Rate (TPR), while the x-axis denotes the False Positive Rate (FPR). As per the real label for each test sample as well as the predicted probability value, various points predicted by the model can be obtained via threshold adjustment. Connection of these points can be done with the curve, which is also referred to as the ROC curve. The effect of the model is enhanced with a larger Area Under Curve (AUC).

$$TPR = \frac{TP}{TP + FN} \tag{5}$$

$$FPR = \frac{FP}{FP + TN} \tag{6}$$

5.5. Comparison of different feature extraction methods

The proposed feature extraction method takes into account the flow of each program path, including sanitization functions and the vulnerability context. The results of our method were filtered, with features that appeared unrelated to the vulnerability's flow being removed. Two groups of comparative experiments have been set up based on the proposed method to measure the impact of the carried-out improvement on the MSSA features. The first group refers to the simplest MSSA features, which lack knowledge of vulnerability context and are filtered, whereas the second group refers to our method, which includes information on vulnerability context for each case and filtering for each path feature.

In addition to these two feature groups, we included another three methods based on the literature on publicly available comparative experiments (PHP, 2022; Walden et al., 2014; Fang et al., 2019). All the five methods were applied to the same dataset presented in Section 5.2.

The PHP built-in function *token_get_all()* has been found to parse code into PHP tokens via the Zend engine (PHP, 2022). The translation of each PHP keyword and symbol is done into a word beginning with a capital letter *T*. However, this function has also been found to parse any other strings into the same token *T_CONSTANT_ENCAPSED_STRING*, which makes it difficult to extract information from strings due to the loss of string content details. Also, this function has been found to only recognise PHP keywords as well as a few of the functions. Other identifiers and functions, such as class names, have all been parsed into token *T_STRING* (PHP, 2022).

TAP (Fang et al., 2019) obtains code features from codes written in PHP by making use of a custom tokenizer built based on PHP's token mechanism (*token_get_all()*), and completes data flow examination to discover relevant code lines that have function calls. Walden et al. (2014) proposed a method called text mining features, wherein it starts by tokenizing each source file of PHP with PHP's inbuilt *token_get_all()* function. Furthermore, their feature extraction method processes the group of tokens to discard whitespace and comment tokens. For instance, numeric and string literals are converted to form fixed tokens. For instance, *T_STRING* is used to form a string literal rather than the string contents. Table 4 summarises the datasets generated by these five methods, where the second column shows a sample's average length, which is followed by the size of its vocabulary.

As displayed in Table 4, the *token_get_all()* function's average length was 79, and the size of the vocabulary was 250. After analysis of data flow, the average length was increased to 108 in TAP, with a lower vocabulary size of 92. The text mining method showed the sample's average length to be 83 and its vocabulary size to be 64. In the case of the "MSSA" form, the features produced have an average length of 137 of a sample and a vocabulary size of 112 before becoming aware of the filtering and vulnerability context. The proposed method decreased the sample's average length to 69 and the size of its vocabulary to 57. All 57 features were related to vulnerability flow, and we incorporated the vulnerability context into each of the context cases.

Table 4
A summary of the datasets generated by each method.

Method	Average Length	Vocab
Text mining	83	64
TAP	108	92
token_get_all()	79	250
MSSA without vulnerability context and filtering	137	112
Our method	**69**	**57**

In order to assess every method's performance in predicting the vulnerable files, we apply the generated features from each method to the different classifiers. Table 5 depicts the precision performance of every method on different classifiers.

The results of Table 5 indicated that the PHP built-in function *token_get_all()* achieved the lowest precision results of all classifiers compared to the other methods, where the best results achieved by this function was 77.03% with the LSTM classifier. In addition, the same classifier (LSTM) was shown to be effective for MSSA and text mining methods with a precision rate of 83.19% and 92.95% for each of them, respectively. On the other hand, the best precision result achieved by the TAP was 90.15% with the BiGRU neural network. The proposed method achieved the highest precision results in all classifiers compared to the previous methods, and the highest precision value achieved was 97.90% with the BiLSTM neural network. It was noted that the linear regression was the lowest among the other classifiers for the methods. Furthermore, most of the classifiers achieved good results on the conducted features that were generated by the proposed method. Thus, it indicates that the feature sets generated from the proposed method have the ability to find vulnerable files more accurately than other methods.

The recall is the percentage of different labeled vulnerabilities that our model properly identifies. Table 6 shows the recall results of every method applied to each classifier.

The recall rate of the proposed method was better than the other methods, where the highest recall achieved by the proposed method was 98.25% with BiLSTM, followed directly by TAP with a recall value of 91.97% with GRU. The text mining method achieved 80.82%, which is still better than both of the other methods (MSSA and *token_get_all()*) with recall values of 77.34% and 77.01% for each of them, respectively. Therefore, the proposed method did better than the previous methods in terms of the recall rate, which showed that the proposed method correctly identified more vulnerable files than the other methods.

Considering the precision of one classifier is high, which indicates that it has a low false positive rate. At the same time, this model achieved a low recall rate, which indicates that not all vulnerable files are predicted correctly by the model. Consequently, the F-score is the optimum method for evaluating the performance of any model. It accords equal weight to precision and recall. Table 7 shows the F1-score results of every method on different classifiers.

The F1-score results of the proposed approach were higher than those of other methods in all classifiers, where it reaches the highest F1-score of 98.07% with BiLSTM, followed by the same F1-score of 97.97% for each classifier of LSTM and BiGRU. The decision tree and random forest classifiers were noted to be better than the other ML classifiers with 97.95% and 97.52%, respectively. The second highest F-score value for the other methods was achieved by TAP and LSTM neural network with 89.98%, followed by the text mining method, MSSA, and token_get_all() function with values of 78.19%, 77.69%, 75.07%, respectively.

The accuracy of the proposed method was determined and compared to the other methods. Table 8 shows the accuracy of each method on different classifiers.

Table 5
Precision performance in (%) of each method on different classifiers.

Method	Decision Tree	KNN	Random forest	Linear regression	SVM	LSTM	BiLSTM	GRU	BiGRU
Text mining	77.68	75.99	77.89	69.65	78.30	**92.95**	77.86	80.00	85.83
token_get_all()	70.26	72.32	75.74	69.28	69.74	**77.03**	76.22	74.01	73.84
TAP	85.97	81.49	85.65	72.66	74.79	89.08	87.91	86.40	**90.15**
MSSA	79.88	76.22	80.41	68.32	69.75	**83.19**	81.39	80.77	76.41
Our method	97.72	93.56	97.47	77.85	83.49	97.86	**97.90**	97.15	97.82

Table 6
Recall performance in (%) of each method on different classifiers.

Method	Decision Tree	KNN	Random forest	Linear regression	SVM	LSTM	BiLSTM	GRU	BiGRU
Text mining	76.69	**80.82**	77.01	68.08	68.01	55.12	77.50	75.45	67.09
token_get_all()	71.54	69.19	71.41	52.61	58.66	68.47	65.28	**77.01**	76.45
TAP	88.16	85.81	86.70	60.48	63.29	90.92	91.09	**91.97**	88.26
MSSA	74.15	**77.34**	74.59	61.42	60.10	71.31	75.08	70.99	76.48
Our method	98.17	94.88	97.58	75.71	75.51	98.19	**98.25**	98.02	98.08

It can be noted from Table 8 that the accuracy results of the proposed method with BiLSTM neural network achieved the highest accuracy results compared to other classifiers and is also higher than all the previous methods. The best accuracy result achieved by TAP at 93.30% was with the LSTM neural network. The text mining method and MSSA achieved around 85% accuracy with GRU and BiLSTM classifiers. The lowest accuracy result collected by *token_get_all()* function was 83.24% with the BiGRU classifier.

The performance results of the proposed method are better than the previous methods, and it was noted that both neural network and ML classifiers worked well on the generated features from the proposed method and achieved good results. This is because the proposed method addresses the vulnerability context in the programs and eliminates the features that are unrelated to the vulnerability flows in the programs, which were missed in the previous studies. Thus, it reduces the noise in the classifiers to learn the vulnerability flows.

Various prediction models have different performance measures, and a statistical significance paired t-test is used to measure how significant the variance of a value is to infer if the mean value could be different. It is also used to determine if there are differences between groups of meanings. Thus, we use a t-statistic and compare this to t-distribution values to determine if the results of the models are statistically significant. We conducted the null hypothesis test to evaluate the substantial difference in the output of each classifier with the same dataset. The null and alternate hypothesis statements in our experiment are defined as follows:

- **H0:** means that there is no significant mean difference in the accuracy value of the model evaluated.
- **H1:** means that the accuracy of the model is less than the mean accuracy obtained on the test.

We assume a 95% significance level, which allows for a 5% error rate, where alpha (α) is 5% or less (0.05). It is statistically significant if a corrected standard t-test has a significance level of 0.05 or less (Chowdhury and Zulkernine, 2011). We employed a paired sample t-test approach, which is often used to evaluate significant differences in different classifiers' performance. It is a form of inferential statistic used to predict whether or not there is a significant difference in the means of two groups (models) on the sample data. The estimated probability, also known as the P-value, is utilized to determine the observed or more extreme findings when our null hypothesis (H0) assertion is true at the significance level (α=0.05). As a result, the null hypothesis that both models perform equally well on the same dataset cannot be rejected if P-value > *alpha*. That is, the performance of the two models does not vary statistically substantially. If the P-value $\leqslant\alpha$, the null hypothesis is rejected, indicating that the performance of the two models is significantly different.

We used a one-tailed t-test on the Python stats module (Statistics, 2022). This function gives us both the t-value and the P-value to compare with the α value. We used a KNN-based predictor as the baseline predictor and did a standard t-test with a significance level of 0.05. Table 9 shows the pairwise t-test results of each classifier's mean accuracy, mean standard deviation (std), P-value, and t-statistics for the same dataset (the feature sets that were generated by the proposed method).

All models perform well over 96% accuracy, which shows how well the recommended characteristics work in making a model for predicting vulnerability. Both the linear regression and SVM models have the lowest accuracy rate of 89% and 92%, respectively. An average accuracy of the BiLSTM model of 98% was attained with a very low standard deviation of only 0.1%, making it the most accurate model. In addition, the accuracy results of BiLSTM are very close to those of Random Forest, Decision Tree, KNN, LSTM, GRU, and BiGRU classifiers on the same dataset. The results of the paired t-tests showed that the performance of the various classifiers on our created dataset differed significantly. Due to the fact that all P-values are smaller than α 0.05, the null hypothesis (H0) is rejected.

5.6. Comparison of Different Classifiers

The effectiveness of ML and neural network classifiers may be evaluated using a variety of metrics, including the time it takes to develop a learning model, the time it takes to test the model, the size of the training data set, the consistency and quality of the outputs, etc. Different ML and neural network algorithms are compared in terms of the time needed to train a model, the quantity of training data needed to achieve a specific level of prediction accuracy, and the impact that class imbalance has on training and testing data.

5.6.1. Training Time and the Size of the Training Data

Fig. 7 depicts a comparison of the classifiers in terms of the time required to construct models for various training data sizes. It demonstrates that the neural network classifiers have the longest training time, where the longest training time was recorded for the BiLSTM model, followed by BiGRU, LSTM, and GRU. On the other hand, the training time for traditional ML algorithms did not take as long time as the neural networks, whereas the KNN

Table 7
F-score performance in (%) of each method on different classifiers.

Method	Decision Tree	KNN	Random forest	Linear regression	SVM	LSTM	BiLSTM	GRU	BiGRU
Text mining	77.12	**78.19**	77.44	68.78	71.94	53.51	77.63	77.46	73.25
token_get_all()	70.84	70.65	73.38	53.89	61.33	71.07	69.09	74.94	**75.07**
TAP	86.96	83.49	86.17	63.19	65.95	**89.98**	89.42	88.92	89.12
MSSA	76.72	76.72	77.19	63.88	62.61	76.02	**77.69**	74.84	76.45
Our method	97.95	94.21	97.52	76.62	77.59	97.97	**98.07**	97.87	97.97

model's training time is insignificant, followed by Decision Tree and Random Forest, respectively. In addition, it shows that the time required to develop the model grows as the quantity of the training data increases for each of the evaluated classifiers.

Fig. 8 shows a comparison of the time required by each classifier to construct the model with a 90% training set size. It indicates that the BiLSTM requires the highest amount of time, 869.34 s, to build the model. The other neural network classifiers are followed as BiGRU, LSTM, and GRU, each of which takes 524.94, 438.86, and 308.78 s. Traditional ML classifiers, on the other hand, required less time to complete model building on the given data size (90%), with KNN being the fastest classifier with a time of 0.0028, followed by Decision Tree, Random Forest, and Linear regression with times of 0.13, 1.58, and 2.58, respectively.

It is necessary to upgrade the training model for a good vulnerability prediction system because the training samples are likely to vary over time. We believe that the training time for a prediction model should not grow exponentially with the amount of training data provided. It was noted from the previous results that the neural network models (i.e. BiLSTM, BiGRU, LSTM, and GRU) require more time to build the model than the traditional ML models, because the neural networks have several layers with a large number of neurons in the layers that need to be processed and a greater number of iterations (number of epochs) that need to be tested to find the proper results for the model. Thus, the traditional ML classifiers are more effective in terms of the training time, where the lowest training time was achieved by KNN, Decision Tree, and Random Forest, respectively.

5.6.2. The Influence of Training Data Size on Model Performance

The quality and quantity of training data are often the deciding factors in a model's performance. Using empirical scaling models,

Table 8
Accuracy performance in (%) of each method on different classifiers.

Method	Decision Tree	KNN	Random forest	Linear regression	SVM	LSTM	BiLSTM	GRU	BiGRU
Text mining	85.02	84.31	85.00	80.26	83.85	83.16	85.40	**85.66**	85.35
token_get_all()	82.09	82.09	82.96	78.50	80.21	82.83	82.63	83.01	**83.24**
TAP	91.34	89.07	90.85	82.14	84.21	**93.30**	92.94	92.89	92.99
MSSA	85.48	84.87	85.61	80.77	79.55	85.53	**85.91**	85.05	85.17
Our method	98.51	96.63	98.42	89.83	92.13	98.67	**98.70**	98.28	98.34

Table 9
Pairwise Comparison of T-test for different classifiers' performance.

Classifiers	Mean Accuracy	Mean std.	T-test	Results (P-value)	Reject/Accept
Decision Tree	98.62	0.000250000	−103.49	0.000000000	Reject
KNN	96.63	0.000000000	NA	NA	NA
Random forest	98.42	0.000368000	−50.01	0.000000000	Reject
Linear regression	89.83	0.000000000	2.15e+28	0.000000000	Reject
SVM	92.13	0.000000000	1.53e+28	0.000000000	Reject
LSTM	98.67	0.000527635	−66.36	0.000000000	Reject
BiLSTM	98.70	0.001085989	−32.14	0.000000000	Reject
GRU	98.59	0.000360126	−83.14	0.000000000	Reject
BiGRU	98.67	0.000763942	−45.00	0.000000000	Reject

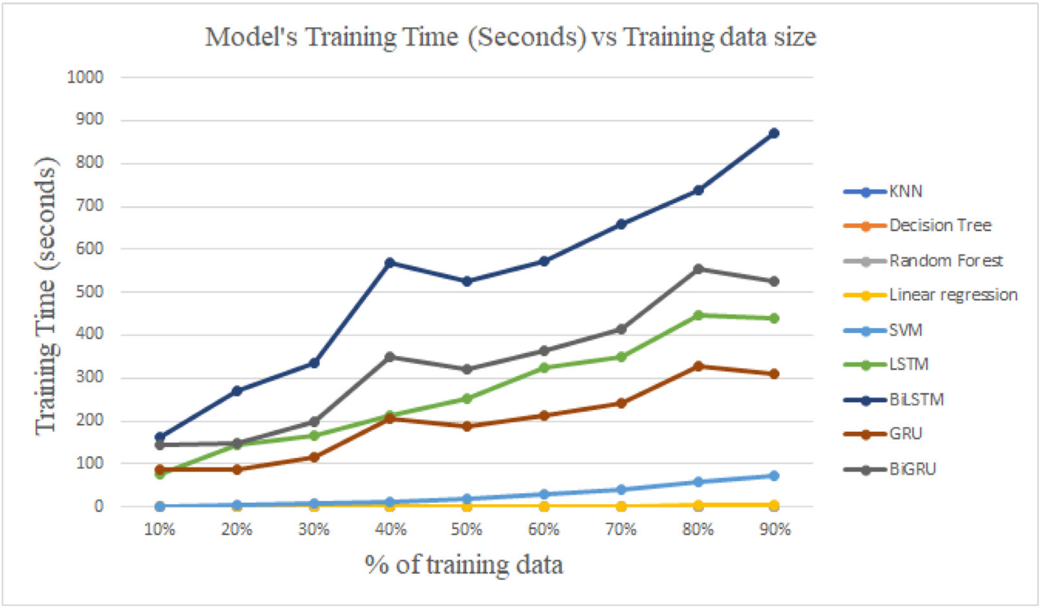


Fig. 7. The effect of different training data sizes on the training time of each model.

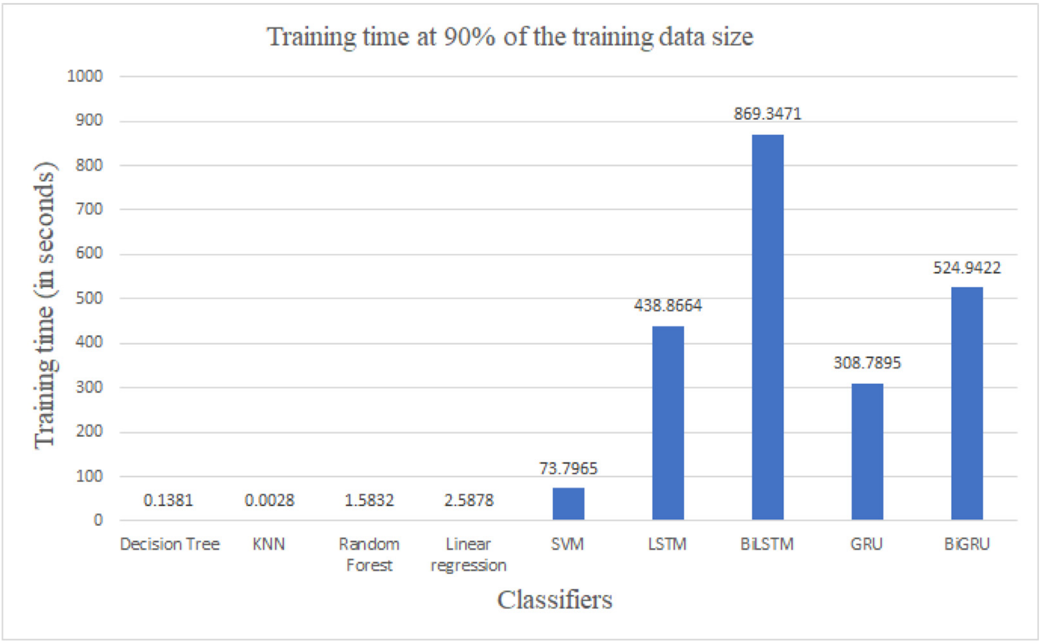


Fig. 8. The effect of 90% training data size on the training time of each model.

known as learning curves, is a natural way to analyze the performance of machine learning algorithms (Cortes et al., 1993). By examining the learning curve, it is possible to assess how much data is required to develop a prediction model using various machine learning algorithms (Perlich et al., 2003). The general properties of a prediction model's learning curve may be summarized as follows (Beleites et al., 2013): At first, when there isn't enough data to learn effectively, a model's performance increases fast. However, as more data becomes available, the learning curve flattens out and the slope gets closer to 0, indicating that further training data isn't adding much new information.

Various models need different amounts of training data in order to reach a specific degree of prediction accuracy. Comparative results for each ML and neural network classifier are shown in Fig. 9. Up to a given amount of training data, we believe that the learning curve for a good vulnerability predictive model should be smooth and monotonic. Fig. 9 demonstrates that the effectiveness of all classifiers varies within a range of outcomes as the training sample amount grows. Furthermore, Table 10 concludes the results of different training data sizes by sorting the models by their accuracy results from 1 to 9, where 1 is the best model in obtaining the highest accuracy rate and 9 is the lowest.

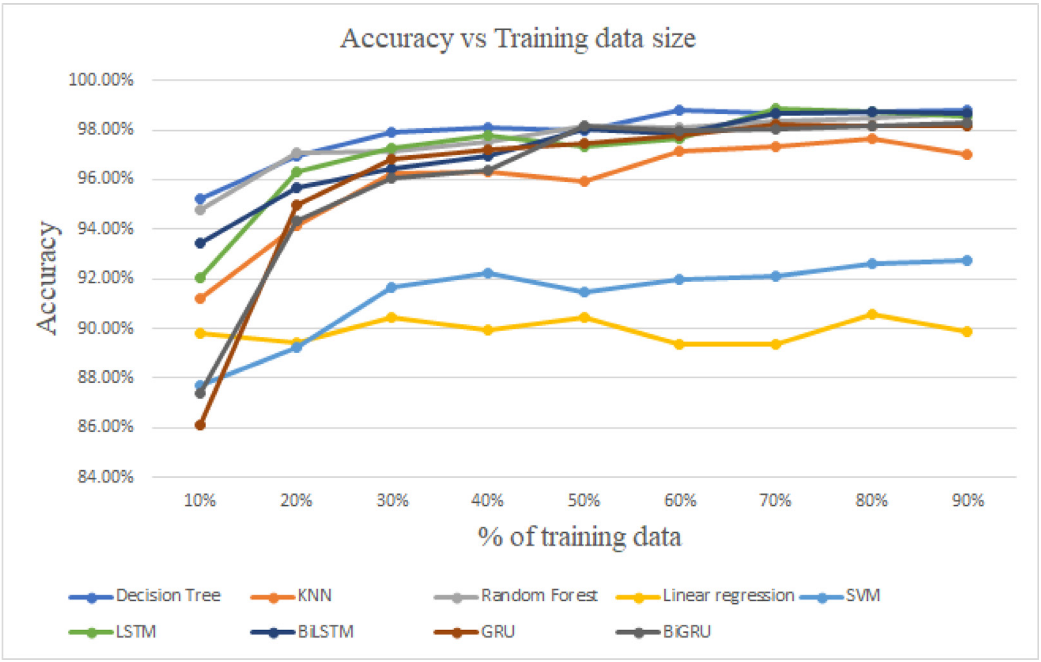


Fig. 9. The effect of training data size on model accuracy.

Table 10
Sorting classifiers by their best performance results at training size.

Rank	Classifier	Accuracy (%)	Training data size (%)	Time (s)
1	LSTM	98.75	80	445.730846
	BiLSTM	98.75	80	738.635573
2	Decision Tree	98.57	70	0.093114
3	Random Forest	98.56	90	1.583257
4	BiGRU	98.27	90	308.78954
5	GRU	98.24	70	242.93161
6	KNN	97.66	80	0.001811
7	SVM	92.72	90	73.796568
8	Linear Regression	90.60	80	2.239999

The following are the findings:

- The performance of linear regression and SVM fluctuated with the change in data size, as it increased in some periods and decreased with the change in the size of the data. In addition, the results of the two models were lower than the rest of the other models in terms of the ability to predict the vulnerabilities.
- The decision tree model has low performance after increasing the data size by more than 70%.
- The performance of Random Forest, KNN, and the neural network models (LSTM, BiLSTM, and BiGRU) was better when the data size increased accordingly.
- A point on the learning curve when the curve starts to flatten reflects the lowest quantity of training data. According to the good results obtained by the neural network models, around 80% of training samples are necessary to construct a good training model.
- Both LSTM and BiLSTM classifiers achieved the best accuracy rate (98.75%) compared to other classifiers. They both achieved these results with an 80% training data set. However, LSTM requires less time to build the model than BiLSTM on the data size (80%). Thus, we can conclude that LSTM performs better than BiLSTM in terms of the time taken to process the model.

5.6.3. Imbalanced Dataset Influence

Imbalanced dataset is a regular occurrence in the machine learning field in which a large number of cases represent one class while fewer cases represent another. As a result, a model properly identifies cases that belong to a class with a higher percentage of samples than the other class.

When compared to safe samples, the vulnerable population is smaller in the current world. As a result, we're curious whether a vulnerability may be found using the given dataset, which includes a greater number of safe samples. Experiments were carried out on the dataset, including 3800 vulnerable safe samples and 3800 safe (not vulnerable) samples. Experiments with varying ratios of vulnerable samples are carried out. The vulnerable sample numbers evaluated in each experiment are V% of safe samples (where V% is 10%, 20%, 30%, and so on). For instance, if there are 3800 safe samples, there are 760 vulnerable samples (V = 20% of safe samples). Fig. 10 displays the performance of each ML and neural network classifier with increasing proportions of vulnerable samples.

There is no significant variation in the TP rate and TN rate values for the neural network classifiers (LSTM, BiLSTM, GRU, and BiGRU) with increasing vulnerability sample values. As a result, the neural network models seem to be unaffected by the issue of class imbalance. On the other hand, increasing the proportion of vulnerable samples leads to enhanced AUC values of random forest, decision tree, and KNN classifiers. It demonstrates that the class-imbalance issue affects those models' performance.

The researchers (Younis et al., 2016; Younis et al., 2016; Chowdhury and Zulkernine, 2011; Scandariato et al., 2014) advise that predicting all vulnerable files is more essential than incorrectly predicting non-vulnerable files as vulnerable, because a single unprotected file might expose a major security threat. Based on

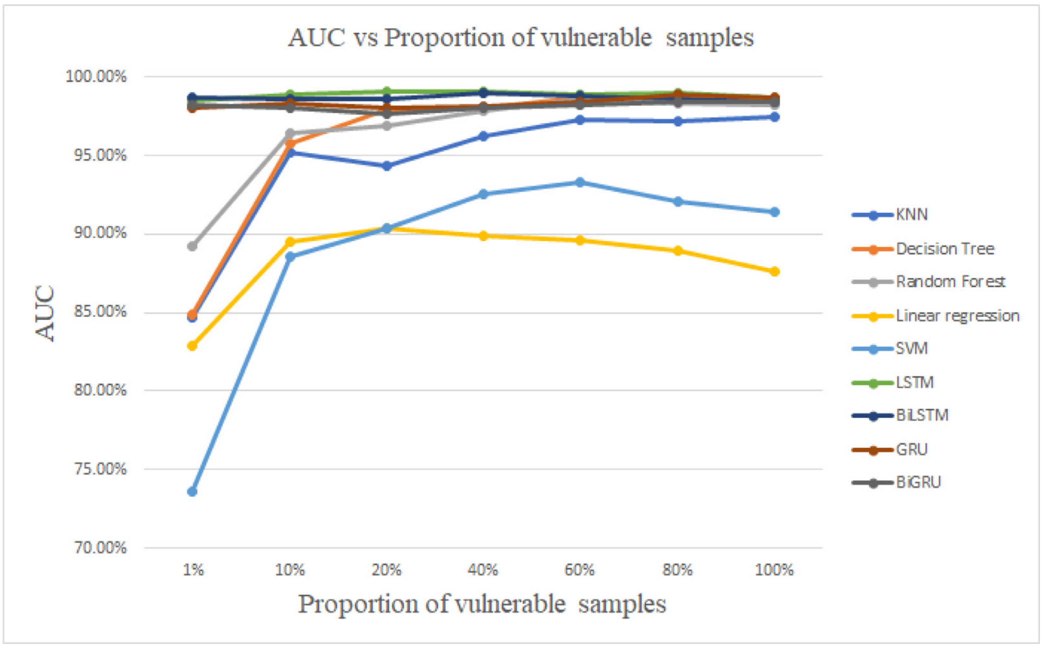


Fig. 10. ROC curve of each classifier on different proportion vulnerable samples.

the conducted experiments, the LSTM classifier outperformed all other classifiers as it has the best AUC values in all proportions of vulnerable samples, with the highest AUC value (99.03%) of LSTM noted in the rate of 40% of vulnerable samples. This indicates that the LSTM model can perform well in predicting vulnerable samples with fewer false positives. Therefore, we can recommend LSTM classifiers as the best for our research since they have a modest training time compared to other neural network classifiers and among the highest performance values.

5.7. Comparison with state-of-the-art methods

For answering RQ3, we made a comparison of our model (with LSTM classifier) and other vulnerability prediction models from the literature. After extracting the source code's fundamental and context features, M. K. Gupta et al. (2015) used several machine-learning models (e.g., Random Forest, SVM, J48, etc.) to predict context-sensitive security vulnerabilities. Li et al. (2020) analyzed the input/output patterns in XSS-related PHP source code; afterwards, they devised an algorithm for identifying the data stream's specific path. By implementing the BiLSTM network, the researchers aimed to learn their triples' relationships from feature sequences.

Li et al. (2021,) proposed two methods. "SySeVr" (Li et al., 2021) is the first method, applying the program slice to extract the features from CFG. Then, a BiGRU network is utilised to automatically determine vulnerable programming patterns. The second method, "VulDeeLocator", (Li et al., 2021) automatically uses the SSA form to represent the program code and etract the program features using slicing and employs attention mechanisms and BiGRU to discover vulnerable programming patterns. "SySeVr" and "VulDeeLocator" are both sequence-based methods. The outcomes of every model are listed in Table 11, where the best case is chosen for every model to be matched against our model.

Based on the results, it can be easily predicted that the proposed model overrides each of the proposed models in the past with a significant margin. The results of Gupta et al. (2015) and Li et al. (2021) were shown to have the lowest performance as shown in the results, with an accuracy rate of 88.9% and 89.1% for each of them, respectively. They extracted the whole program's features. However, some of the features are not related to the vulnerability flow in the programs. Thus, it shows the data dependency issues between the extracted features and those that lead to the lower performance of the models. Li et al. (2020) analysed the input/output stream of the vulnerability in the programs, which helps to add additional information for the model to learn about the vulnerability. However, they consider all the extracted features in the program to be related to the vulnerability flow in the program, which lowers its performance to predict the vulnerabilities with an accuracy rate of 92%. On the other hand, Li et al. (2021) enhanced their previous method (Li et al., 2021) by converting the program code to an SSA form, which reduced the instructions to be analyzed. Thus, they achieved good results (i.e., accruacy of 97.7%) after reducing several instructions by excxtracting the features from the SSA form. However, the SSA form includes a simple representation of each instruction of the program statements and defines it only once. Therefore, considering all the features from SSA form still containing alot of instructions that may not related to the vulnerabilities flow in the programs.

Table 11
Comparison with related studies.

Method	P (%)	R (%)	F1 (%)	Acc (%)
Gupta et al. (2015)	88.7	83.0	85.9	88.9
Li et al. (2021)	83.1	88.5	85.7	89.1
Li et al. (2021)	97.1	95.9	96.8	97.7
Li et al. (2020)	92.0	92.3	92.1	92.3
Our Method + LSTM	**97.9**	**98.1**	**97.97**	**98.67**

Our proposed method filters the extracted features and eliminates the features that are unrelated to the vulnerability flow in the program. The number of features is reduced, thus making the model process of learning the vulnerability context in each program more direct and clear with those connected features that are related to the flow of the vulnerability in the programs. An accuracy rate of 98.76% and the highest precision and high recall were sustained by this model, as evidenced by the F1-score, which is because of the more detailed feature sequence that the method generated.

6. Conclusion and future work

This paper proposes an approach for predicting the vulnerable files against input validation vulnerabilities. Transforming the programs into intermediate representations first (MSSA form) removes irrelevant instructions in the SSA form; it likewise depicts the vulnerabilities with their corresponding explicit dependency relations. Then, a method was proposed to extract the program features and eliminate the features not related to the vulnerability flow in the program. Overall, the proposed method shows its ability to generate the features that are related to the vulnerability flow in the program, and the results of different ML and neural network classifiers demonstrate the effectiveness of the generated dataset in predicting XSS and SQLi in PHP vulnerable files. It was noted that the best performance results achieved by LSTM and BiLSTM were nearly equal in different evaluation metrics. However, the BiLSTM requires more time than the LSTM to achieve the best results. For those, we considered LSTM classifier to be the best choice for predicting input validation vulnerabilities on the generated dataset.

The proposed approach has some drawbacks, which can be further studied. First of all, we applied our method to predict the current vulnerabilities in written source code, particularly in the PHP language. Our approach can likewise be applied theoretically to other programming languages. Thus, applying our approaches to other languages is one of the interesting prospects. Secondly, owing to the lack of labelled vulnerability datasets, our approach was conducted only on the SARD dataset. Formulation of an automated vulnerability prediction technology is restricted by the lack of labeled datasets; this fact is an open industry problem. For this reason, one of the interesting works in the future is to propose a method of generating labels on real-world software and transferring the learned vulnerable patterns in order to predict vulnerabilities in a new project. Thirdly, the program paths' execution feasibility aided in enhancing the outcomes of traditional static analysis tools. Therefore, to eliminate non-executable paths in the program under any user input, a model proposal would be useful.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The authors would like to thank Universiti Sains Malaysia and Arab Open University, Saudi Arabia, for supporting this study.

References

Ali, F., Kwak, D., Khan, P., El-Sappagh, S., Ali, A., Ullah, S., Kim, K.H., Kwak, K.-S., 2019. Transportation sentiment analysis using word embedding and ontology-based topic modeling. Knowledge-Based Systems 174, 27–42.

Alloghani, M., Al-Jumeily, D., Mustafina, J., Hussain, A., Aljaaf, A.J., 2020. A systematic review on supervised and unsupervised machine learning algorithms for data science. Supervised and unsupervised learning for data science, 3–21.

U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning distributed representations of code, Proceedings of the ACM on Programming Languages 3 (POPL) (2019) 1–29.

Ashari, A., Paryudi, I., Tjoa, A.M., 2013. Performance comparison between naïve bayes, decision tree and k-nearest neighbor in searching alternative design in an energy simulation tool. International Journal of Advanced Computer Science and Applications (IJACSA) 4 (11).

Backes, M., Rieck, K., Skoruppa, M., Stock, B., Yamaguchi, F., 2017. Efficient and flexible discovery of php application vulnerabilities. In: 2017 IEEE european symposium on security and privacy (EuroS&P). IEEE, pp. 334–349.

D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, arXiv preprint arXiv:1409.0473 (2014).

Bansal, T., Belanger, D., McCallum, A., 2016. Ask the gru: Multi-task learning for deep text recommendations. In: Proceedings of the 10th ACM Conference on Recommender Systems, pp. 107–114.

Beleites, C., Neugebauer, U., Bocklitz, T., Krafft, C., Popp, J., 2013. Sample size planning for classification models. Analytica Chimica Acta 760, 25–33. https://doi.org/10.1016/j.aca.2012.11.007.

Bensalim, S., Klein, D., Barber, T., Johns, M., 2021. Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis, EuroSec '21. Association for Computing Machinery, New York, NY, USA, pp. 27–33. 10.1145/3447852.3458718.

Braun, M., Buchwald, S., Hack, S., Leißa, R., Mallon, C., Zwinkau, A., 2013. Simple and efficient construction of static single assignment form. In: International Conference on Compiler Construction. Springer, pp. 102–122. https://doi.org/10.1007/978-3-642-37051-9_6.

Breiman, L., 2001. Random forests mach learn 45 (1), 5–32.

Chowdhury, I., Zulkernine, M., 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. Journal of Systems Architecture 57 (3), 294–313.

Cortes, C., Vapnik, V., 1995. Support-vector networks in machine learning 20, 3.

Cortes, C., Jackel, L.D., Solla, S., Vapnik, V., Denker, J., 1993. Learning curves: Asymptotic values and rate of convergence. Advances in neural information processing systems 6.

CVE, Cve details, https://www.cvedetails.com/browse-by-date.php ((accessed January 12, 2022)).

N.V. Database, Nvd - statistics search, https://web.nvd.nist.gov/view/vuln/statistics ((accessed April 30, 2021)).

Duda, R.O., Hart, P.E., et al., 2006. Pattern classification. John Wiley & Sons.

Fang, Y., Han, S., Huang, C., Wu, R., 2019. Tap: A static analysis model for php vulnerabilities based on token and deep learning technology. PloS one 14, (11) e0225196.

Fawcett, T., 2006. An introduction to roc analysis. Pattern recognition letters 27 (8), 861–874.

Gensim, Word2vec model, https://radimrehurek.com/gensim/auto_e xamples/tutorials/run_word2vec.html ((accessed January 15, 2022)).

Goldberg, Y., 2017. Neural network methods for natural language processing. Synthesis lectures on human language technologies 10 (1), 1–309.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep learning. MIT press.

Gupta, S., Gupta, B.B., 2017. Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. International Journal of System Assurance Engineering and Management 8 (1), 512–530.

Gupta, M.K., Govil, M.C., Singh, G., 2015. Predicting cross-site scripting (xss) security vulnerabilities in web applications. In: 2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE). IEEE, pp. 162–167.

Gupta, M.K., Govil, M.C., Singh, G., 2018. Text-mining and pattern-matching based prediction models for detecting vulnerable files in web applications. Journal of Web Engineering, 028–044.

Hanif, H., Nasir, M.H.N.M., Ab Razak, M.F., Firdaus, A., Anuar, N.B., 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. Journal of Network and Computer Applications 179, 103009.

Hong, Y., Liu, Y., Yang, S., Zhang, K., Hu, J., 2020. Joint extraction of entities and relations using graph convolution over pruned dependency trees. Neurocomputing 411, 302–312.

Y. Hu, A framework for using deep learning to detect software vulnerabilities (2019).

A.K. Jain, R.C. Dubes, Algorithms for clustering data, Prentice-Hall Inc, 1988.

Jin, Y., Khan, L., Wang, L., Awad, M., 2005. Image annotations by combining multiple evidence & wordnet, in. In: Proceedings of the 13th annual ACM international conference on Multimedia, pp. 706–715.

Jolliffe, I.T., 2002. Principal component analysis for special types of data. Springer.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems 25.

Lattner, C.A., 2002. Llvm: An infrastructure for multi-stage optimization Ph.D. thesis. University of Illinois at Urbana-Champaign.

Léchenet, J.-C., Blazy, S., Pichardie, D., 2020. A fast verified liveness analysis in ssa form. In: International Joint Conference on Automated Reasoning. Springer, pp. 324–340. https://doi.org/10.1007/978-3-030-51054-1_19.

Leißa, R., Köster, M., Hack, S., 2015. A graph-based higher-order intermediate representation. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, pp. 202–212. https://doi.org/10.1109/CGO.2015.7054200.

Li, X., Xue, Y., 2011. A survey on web application security. Nashville, TN USA 25 (5), 1–14.

Li, C., Wang, Y., Miao, C., Huang, C., 2020. Cross-site scripting guardian: A static xss detector based on data stream input-output association mining. Applied Sciences 10 (14), 4740.

Li, X., Wang, L., Xin, Y., Yang, Y., Tang, Q., Chen, Y., 2021. Automated software vulnerability detection based on hybrid neural network. Applied Sciences 11 (7), 3201.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing.

Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H., 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Transactions on Dependable and Secure Computing.

Liu, F.T., Ting, K.M., Zhou, Z.-H., 2008. Isolation forest. In: 2008 eighth ieee international conference on data mining. IEEE, pp. 413–422.

Luo, X., Zhou, W., Wang, W., Zhu, Y., Deng, J., 2017. Attention-based relation extraction with bidirectional gated recurrent unit and highway network in the analysis of geological data. IEEE Access 6, 5705–5715.

Manevitz, L.M., Yousef, M., 2001. One-class svms for document classification. Journal of machine Learning research 2 (Dec), 139–154.

Marashdih, A.W., Zaaba, Z.F., Suwais, K., 2021. An approach for detecting feasible paths based on minimal ssa representation and symbolic execution. Applied Sciences 11 (12), 5384.

Martin, M.C., Lam, M.S., 2008. Automatic generation of xss and sql injection attacks with goal-directed model checking., in. USENIX Security symposium, 31–44.

Maskur, A.F., Asnar, Y.D.W., 2019. Static code analysis tools with the taint analysis method for detecting web application vulnerability. In: 2019 International Conference on Data and Software Engineering (ICoDSE). IEEE, pp. 1–6.

Medeiros, I., Neves, N., Correia, M., 2015. Detecting and removing web application vulnerabilities with static analysis and data mining. IEEE Transactions on Reliability 65 (1), 54–69.

Money, A., Affleck-Graves, J., Hart, M., Barr, G., 1982. The linear regression model: Lp norm estimation and the choice of p. Communications in Statistics-Simulation and Computation 11 (1), 89–109.

OWASP, Top-10 threats for web application security –2020, https://owasp.org/www-project-top-ten/ ((accessed September 15, 2020)).

OWASP, Cross site scripting prevention cheat sheet, https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Sc ripting_Prevention_Cheat_Sheet.html ((accessed April 15, 2021)).

C. Perlich, F. Provost, J. Simonoff, Tree induction vs. logistic regression: A learning-curve analysis (2003).

PHP, token_get_all, https://www.php.net/manual/en/function.toke n-get-all.php ((accessed January 15, 2022)).

PHP, List of parser tokens, https://www.php.net/manual/en/tokens.php ((accessed January 15, 2022)).

N. Popov, Php-parser, https://github.com/nikic/PHP-Parser ((accessed August 29, 2020)).

Quinlan, J.R., 2014. C4.5: programs for machine learning. Elsevier.

Quiroga, J., Ortin, F., 2017. Ssa transformations to facilitate type inference in dynamically typed code. The Computer Journal 60 (9), 1300–1315. https://doi.org/10.1093/comjnl/bxw108.

Rocha, R.C., Petoumenos, P., Wang, Z., Cole, M., Leather, H., 2020. Effective function merging in the ssa form, in. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 854–868. https://doi.org/10.1145/3385412.3386030.

Rousseeuw, P.J., 1984. Least median of squares regression. Journal of the American statistical association 79 (388), 871–880.

Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W., 2014. Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering 40 (10), 993–1006.

Schardl, T.B., Moses, W.S., Leiserson, C.E., 2019. Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation. ACM Transactions on Parallel Computing (TOPC) 6 (4), 1–33. https://doi.org/10.1145/3365655.

Shar, L.K., Tan, H.B.K., 2012. Automated removal of cross site scripting vulnerabilities in web applications. Information and Software Technology 54 (5), 467–478.

Shar, L.K., Tan, H.B.K., 2012. Defeating sql injection. Computer 46 (3), 69–77.

Shar, L.K., Tan, H.B.K., 2013. Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns. Information and Software Technology 55 (10), 1767–1780.

Shar, L.K., Briand, L.C., Tan, H.B.K., 2014. Web application vulnerability prediction using hybrid program analysis and machine learning. IEEE Transactions on dependable and secure computing 12 (6), 688–707.

K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556 (2014).

H. Spens, J. Lindgren, Using cloud services and machine learning to improve customer support: Study the applicability of the method on voice data (2018).

P. Statistics, Mathematical statistics functions, https://docs.python.org/3/library/statistics.html ((accessed April 3, 2022)).

Stivalet, B., Fong, E., 2016. Large scale generation of complex and faulty php test cases. In: 2016 IEEE International conference on software testing, verification and validation (ICST). IEEE, pp. 409–415.

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. Advances in neural information processing systems 27.

Walden, J., Stuckman, J., Scandariato, R., 2014. Predicting vulnerable components: Software metrics vs text mining. In: 2014 IEEE 25th international symposium on software reliability engineering. IEEE, pp. 23–33.

M. Weiser, Program slicing, IEEE Transactions on software engineering (4) (1984) 352–357.

Younis, A., Malaiya, Y.K., Ray, I., 2016. Assessing vulnerability exploitability risk using software properties. Software Quality Journal 24 (1), 159–202.

A. Younis, Y. Malaiya, C. Anderson, I. Ray, To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit, in: Proceedings of the sixth ACM conference on data and application security and privacy, 2016, pp. 97–104.

Yusof, I., Pathan, A.-S.K., 2016. Mitigating cross-site scripting attacks with a content security policy. Computer 49 (3), 56–63.

Y. Zhang, P. Qi, C.D. Manning, Graph convolution over pruned dependency trees improves relation extraction, arXiv preprint arXiv:1809.10185 (2018).