ConvXSS: a deep learning-based smart ICT framework against code injection attacks for HTML5 web applications in sustainable smart city infrastructure

Koundinya Kuppa
Anushka Dayal
Shashank Gupta
Amit Dua
Pooja Chaudhary
Shailendra Rathore

Kuppa, K., Dayal, A., Gupta, S., Dua, A., Chaudhary, P. & Rathore, S. (2022) 'ConvXSS: a deep learning-based smart ICT framework against code injection attacks for HTML5 web applications in sustainable smart city infrastructure'. *Sustainable Cities and Society*, 80.
DOI: https://doi.org/10.1016/j.scs.2022.103765

ConvXSS:A Deep Learning-Based Smart ICT Framework against Code Injection Attacks for HTML5 Web Applications in Sustainable Smart City Infrastructure

Koundinya Kuppa, Anushka Dayal, Shashank Gupta, Amit Dua, Pooja Chaudhary and Shailendra Rathore

***Abstract*—There has been a rapid increase in the interest in smart cities owing to the surge in the deployment of Information and Communications Technology (ICT). As a part of this, HTML5-based mobile apps and JavaScript-based apps have become popular owing to the portability of these applications for both Android and iOS platforms. But this web technology also comes with a dangerous feature where code and data can be mixed. This opens up various security and privacy issues in smart cities as the vulnerability could be exploited by the attackers by injecting malicious code into the application. This type of attack, commonly known as a code injection attack has taken the first place in Web app vulnerabilities as published by the Open Web Application Security Project (OWASP). Though many approaches based on machine learning have been proposed in the last few years for their detection, these methods cannot cater to the existing needs well enough. In this paper we propose ConvXSS, a novel deep learning approach for the detection of XSS and code injection attacks, followed by context-based sanitization of the malicious code if the model detects any malicious code in the application. Firstly, we briefly discuss XSS and code injection attacks that might pose threat to smart cities. Along with this, we discuss various approaches proposed previously for the detection and alleviation of these attacks followed by their respective limitations. Then we propose our deep learning model adopting whose novelty is based on the approach followed for Data Pre-Processing. Then we finally propose Context-based Sanitization to replace the malicious part of the code with sanitized code. Numerical experiments conducted on various datasets have shown various results out of which the best model has an accuracy of 99.42%, a precision of 99.81% and a recall of 99.35%. When compared with other state of the art techniques in this domain, our approach shows at par or in the best case, better results in terms of detection speed and accuracy of CSS attacks.**

***Index Terms*—Smart City, Security, Privacy, ICT, CPS, Web Security, Deep Learning, CNN, Malicious Code.**

I. Introduction

Nowadays, everything and everyone in this world is interconnected through the internet. Owing to the massive connectivity the internet provides, these services are widespread in almost all the domains. Along with this, smartphones became popular over the years owing to the portability and the wide range of features available. Smart Energy Meters, Smart Appliances and Security devices have become part of everyday life. This is the first step towards transforming cities into Smart cities. Due to the incapability of the infrastructure of the present cities in terms of scalability, environment, and security, there has been a sudden surge in the demand for transformation towards smart cities.[1] So, in this process of transformation, to grab the advantages of the technological advances, there is a surge in the requirement of development in many smart appliances and applications. Of all these, mobile applications play an important role in ICT. But, as every coin has two sides, to keep up with the rapid growth in the technology to build Smart Cities, many developers today are in the haste to develop mobile applications faster to stay in the competition. This hastiness comes with the price of security. To cater a larger number of people, development of different versions of the apps in different languages has to be done to support different platforms such as Android and iOS which involves too much time and labor.[2] Therefore, in order to reduce the work, developers started building apps using standard web technologies like CSS, HTML5, JavaScript owing to the convenient portability of such apps from one platform to another. Hence, there is a tremendous development of applications in the aforementioned languages. As mentioned earlier, with a surge in the number of applications, there is also an increase in the security risks involved. Such security concerns should also be responsibly dealt by the developers. Presently, web applications are at the risk of many hidden vulnerabilities. The aforementioned development technologies host a dangerous feature where they allow data and code to be mixed, which is exploited by the attackers using sophisticated techniques like Cross-Site Scripting (XSS) and Code Injection attacks. This pose a serious threat to people living in smart cities. Already, many users have fallen prey to such attacks compromising their sensitive information to the attackers in the end.[3] XSS attacks work only when the user knowingly or unknowingly opens other malicious sites. On the other hand, a code injection attack, similar to an XSS attack, takes place in mobile applications and has a larger scope of damage since a mobile application interacts with outside world through many plugins and mediums and the code injection attack can happen through any of these mediums. these attack methodologies will

Koundinya Kuppa, Anushka Dayal, Shashank Gupta and Amit Dua are with the Department of Computer Science and Information Systems, Birla Institute of Technology and Science, Pilani, Rajasthan, India - 333031 (email: f20180283@pilani.bits-pilani.ac.in, f20170902@pilani.bits-pilani.ac.in, shashank.gupta@pilani.bits-pilani.ac.in,amit.dua@pilani.bits-pilani.ac.in)

Pooja Chaudhary is with the Department of Computer Science, NIT Kurukshetra, Kurukshetra, India (email: pooja.ch04@gmail.com).

Shailendra Rathore is the Faculty of Science and Engineering,University of Plymouth, United Kingdom (email: shailendra.rathore@plymouth.ac.uk)

Division of Cyber Security, Abertay University, United Kingdom (email: s520967@uad.ac.uk)

be discussed in the further sections.

Fig. 1 shows statistics taken from the OWASP (Open Web Application Security Project) 2017 shows the ranking of different attacks that take place on the applications where it can be seen that the code injection attack takes the first position and XSS attacks stand in the seventh position. [4]

Clearly, we can understand the severity of the attacks and the increase of the security concerns of the aforementioned apps. This would pose a serious threat when there would be an increase in the number of ICT users during the smart city transformation [5]. So, an extensive research has taken place in the field of detection and mitigation of such attacks. Many methods have been put forward for detection of such malicious code attacks in the past but they also have their limitations.

A. Other Models and their Limitations/ Related Work:

There had been substantial research on the detection and numerous methods were proposed for detecting malicious code previously. One conventional approach followed to identify malicious code is by checking whether the signature of a virus is listed in a malicious virus signatures' list prepared based on the user's perspective. This list of virus signatures, known as a blacklist, is created on the system of the host and an external security server regularly updates these virus signatures. [6] However, owing to the complexities involved and lack of up-to-date signature files, this method can barely identify the malicious JavaScript code. Another method based on the pre-established rules by security experts that determines the malicious code from benign also has drawbacks. It can only identify the familiar malicious code but fails to identify the unfamiliar malicious code. Along with the growth in Information and Communications Technology in smart cities, there is also growth in the different types of attacks attackers are coming with. Hence, the above methodology cannot be used. For malicious code detection purposes, dynamic analysis methods in which execution of the malware takes place in a simulated environment is also used widely [7]. Though these methods could detect the malware easily, they are time-consuming, and could not dynamically protect the applications owing to the behavioral changes in malware [8]. Along with these techniques, others such as high-interaction honeypots [9], low-interaction honeypots [10], drive-by downloads [11] [12], heap spraying [13] etc. have relevance, but these methods and approaches are designed for specific attacks; they often consume time, and cannot cater to the ever-evolving attacks. Subsequently, though several methods and approaches have been designed based on machine-learning that could classify malicious and benign code, these approaches require huge amounts of time in designing the features manually for classification purposes which is near to impossible with innovative attack strategies.[14]

Some other methods and approaches proposed by researchers after extensive research include those by [15] in which it was proposed to detect the malicious JavaScript code using a static, non-linear, Support Vector Machine (SVM) and accuracy turned out to be 94.38% which is quite high but the author in [16] did not provide a reason for using a non-linear SVM. The authors of [17] and [18] used a combination of behavioral features and language syntax in machine learning approach to detect XSS attack. This was one of the rudimentary studies using classifier based on Random Forests for detection. The authors in [19] proposed a method in which they considered implementing the binary measures obtained by converting structural features, behavioral features, and classifiers instead of using weighted means. Though this method provided a top rate of accuracy with many models, there is no reasoning mentioned for this particular approach. The authors of [20] discussed identification using word frequency method (TFID) and also used structural as well as functional behavior and Abstract Syntax tree methods for classifying malicious from benign codes but these methods did not play a good role to understand the pattern.[21]

A model which uses deep learning framework was developed in [14]. The author used stacked denoising autoencoders (SDA) feature selection method for classification of malevolent and benevolent codes. It also included many other models such as Logistic Regression and SVM where features are selecting using stacked denoising autoencoders (SDA) and it was concluded that this feature selection is better then that of PCA, ICA and FA. Though this method has better statistical output than other models, the classifier could not effectively classify benign from malicious and the training of neurons take time due to many layers. So, if used a larger dataset, the accuracy of this model can further come down. [22]

Hence, considering all the related work done before and their limitations, we came up with a novel approach of detecting the malicious code using CNN model as a classifier and sanitize the malicious part of the code if it is detected for security and privacy of smart cities.

B. About ConvXSS

Through this paper, we present a procedure that banks on Convolutional Neural Networks- CNN for detection of the malicious code. Our novelty lies in the pre-processing phase, in which each line of code is taken and if encoded, it is decoded, and it is then generalized so that unnecessary randomness can be removed. Then, specifically, each word of the code is semantically labelled based on the type of the string and also using existing labelling information given and the labelled information is converted into binary numbers instead of directly converting the strings into numerical data and the dataset is thus formed. In other words, the code snippets are converted to their ASCII values, followed by the scaling of this numerical data into an image-like format, one ideal as an input to our CNN. A number of comparative studies between different models we have built by tweaking the hyper-parameters is then done to identify the one which gives optimum results on half the data. After singling out the best model as the framework of ConvXSS, the entire Dataset is subsequently used to get our results. The data is divided into 3 sets - training, validation and testing. There is also inclusion of context-based sanitization that sanitizes the malicious code based on the context using a list of sanitizers. Before going to the proposed framework, background knowledge about the attacks is discussed in the next section.

Rank	Name of Risk	Attack Vectors		Security Weakness		Impacts		Score
		Threat Agents	Exploitability	Prevalence	Detectability	Technical	Business	
1	Injection	App Specific	Easy: 3	Common:2	Easy: 3	Severe: 3	App Specific	8.0
2	Authentication	App Specific	Easy: 3	Common:2	Average: 2	Severe: 3	App Specific	7.0
3	Sens. Data Exposure	App Specific	Average: 2	Widespread:3	Average: 2	Severe: 3	App Specific	7.0
4	XML External Entities (XXE)	App Specific	Average: 2	Common:2	Easy: 3	Severe: 3	App Specific	7.0
5	Broken Access Control	App Specific	Average: 2	Common:2	Average: 2	Severe: 3	App Specific	6.0
6	Security Misconfiguration	App Specific	Easy: 3	Widespread:3	Easy: 3	Moderate: 2	App Specific	6.0
7	Cross-Site Scripting	App Specific	Easy: 3	Widespread:3	Easy: 3	Moderate: 2	App Specific	6.0
8	Insecure Deserialization	App Specific	Difficult: 1	Common:2	Average: 2	Severe: 3	App Specific	5.0
9	Vulnerable Components	App Specific	Average: 2	Widespread:3	Average: 2	Moderate: 2	App Specific	4.7
10	Insufficient Logging & Monitoring	App Specific	Average: 2	Widespread:3	Difficult: 1	Moderate: 2	App Specific	4.0

Fig. 1: Ranking of various attacks on applications

II. Background

The aforementioned development technologies host a threatening feature where there is an allowance of blending of data and code together which, is exploited by the attackers using sophisticated attacks like Cross-Site Scripting (XSS). These attacks have to mitigated for the security and the privacy of people living in Smart Cities. For the injection of code, web apps have a single medium; that is via the web site and for this reason, called as "cross-site". Disastrously, this feature's repercussion lies in the fact where if the developers are not cautious, the unpredicted code blended with the data can be triggered impulsively and accidentally. As shown in the Fig. 2 , In the XSS attack, the attackers inject a malicious code into the web app via a web form or web request which is saved on the server later. When the victims access this application, the implementation of the code on the user's application through which the user information is stolen by the attackers. The impact of this will be huge when the number of people using smart mobile applications increases.

Based on the approach of attack, XSS attacks are classified into DOM-based XSS attack, Stored XSS attack, Reflected XSS. While DOM-based XSS attack has the origin of the vulnerability from client's side and not from server side, Reflected XSS is the attack in which the code is passed through request URL such as error message, search result etc. Stored XSS attack is the third type of XSS attack where the malicious code is introduced into the database of the website, such as comment section. Further added to this disadvantage is that the malicious code remains imperceptible for a very long time, possibly forever because the malicious code, if present in

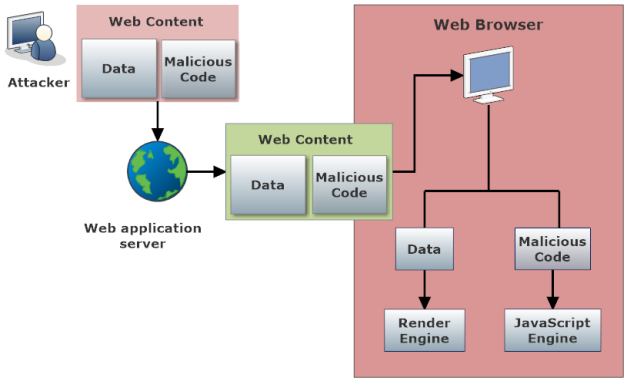

Fig. 2: XSS attacks on ICT Web Applications

the data of the app, is sometimes not displayed but instead executed by the JavaScript engine thus leading to the rise in the number of victims.[14]

A code injection attack is nearly identical to XSS attack. XSS attacks mostly take place in web apps, whereas the code injection attacks takes place in mobile apps. As shown in the Fig. 3 the code injection attack in apps that are hybrid works fundamentally like the XSS attack, but it does not restrict the attack to the data of application only. Since mobile apps required to interact with the external world for better service and functionalities, the attack may use these interaction channels distinctive to mobile phones, like Contacts, SMS, MP4 metadata, MP3 etc., to inject the malicious code.

These channels could be broadly divided into categories like Data Channels, Metadata Channels, ID Channels based

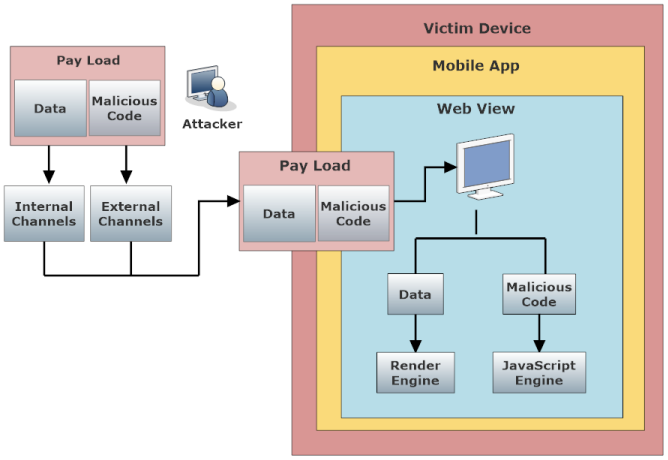

Fig. 3: Code Injection Attack on HTML-5 based Apps to compromise security of Smart Cities

on the approach of the attack. Other than the traditional data sharing channels like Wi-Fi and Bluetooth, smartphones and few smart applications also host data channels like 2D barcodes scanning, RFID tags etc. Since, these Data Channels are quite prevalent in CPS, ICT-technology used in smart cities, these are convenient for the access of info and data for the users, attackers use these channels quite often to inject the code and thus, could compromise the privacy and security of the population in smart cities. The following example shows how a malicious code can be embedded in 2D barcodes upon scanning and attacker could access the victim's location. Since, attacks through data channels have become obvious, attackers also use metadata channels and ID channels. In the approach of using Metadata channels, the attacker might introduce the malicious code into the metadata fields like title, artist name etc., of MP3, MP4, and JPEG files and upon downloading or opening these files, the execution of the malicious code takes place upon displaying this metadata. The app could also be attacked using ID channels i.e. by inserting the malicious code in the place of Wi-Fi or Bluetooth's name. So, whenever a person scans for Wi-Fi or nearby Bluetooth connection, the mere display of the name during the scan could execute the

malicious code in the user's phone. This type of method for injection of code is least obvious. And once injected and executed by the aforementioned methods, the code could access the system resources with the help of plugins that are allowed.[23]

In smartphones and other smart devices involving ICT, along with the interaction with external world, the apps also interconnect with other apps in the device. This interaction enables the spreading of the malicious code into other apps too. If the functions and permissions of the vulnerable app is limited, the app could also inject the code into more privileged app thus escalating the access to more vulnerable information of the victim. These internal channels can be classified into Content Provider, File System, and Intent. Android apps use the features of content provider, file system to store data that could be shared with other apps. Hence, if the malicious app injects the code into the content provider, it could be easily injected into other apps that mutually share these resources. Intent is also used by apps to interact with other apps by passing data. So, if the malicious code is injected into the intent, it is also passed with the existing data thus infecting other apps. Once the device is compromised, it could also act like an attacking device that tries to inject a duplicate of the malicious code into other mobile phones through data sharing like SMS, Bluetooth, sharing media files etc.,[23]

To reduce the number of such attacks and also the number of victims, such vulnerabilities and malicious codes have to be detected and such apps have to be sanitized to make them malicious code free. So, we have chosen a novel approach built on deep learning application to fulfill the above need.

A. Why Deep Learning?

Deep Learning, a looming field of research in machine learning, endeavors to learn sophisticated potrayal of data progressively using deep neural networks. Deep learning could be supervised, semi-supervised or unsupervised. According to [24], unsupervised learning is precarious, it creates many false positives through detection of any type of network anomalies, hence, critical post processing of the output is required, as in DarkTrace [25]. The mentioned method does not need any exemplar datasets since it always learns from progression of previous data. On the other hand, semi-supervised learning infers a limited quantity of known correlations, prior to continuing to the clustering of the obscure data, and for larger networks, the longer they are active inside a particular network, its application can take place in network intrusion detection systems utilizing pre-trained models, that enables in the improvement of their performance. The training of each layer of the network is done via unsupervised learning and the fine tuning of the entire network is carried out in supervised mechanism [26]. So, in this way, low-hierarchy features help in the learning of high-hierarchy features and in the end, proper features could be applied for the classification of patterns.

As stated in the Neural Networks' Universal Approximation theorem [27], the models that are deep have better potentiality than shallow ones to constitute the non-linear functions, enabling the achievement of better results on exhaustive training data. The deep learning framework also assimilates a classifier and feature extractor into a single framework, that spontaneously learns representations of features, thus sparing the effort for feature design manually.

In deep learning, Artificial Neural Network (ANN) was put forward initially and accordingly, the development of deep learning frameworks like CNN and RNN are developed. A Convolutional Neural Network incorporates multiple convolutional layers (Conv1d or Conv2d layer) succeeded by a subsampling layer (pooling) and then associating with a single or multiple fully connected layers. CNN was selected owing to its low computational prerequisites for this particular assignment because it does not need consecutive output for this particular execution, and the dataset given as an input is not a single-dimensional. Convolutional Neural Networks are uncommon feedforward networks with layers having a diminished parameter set because of the training of filters

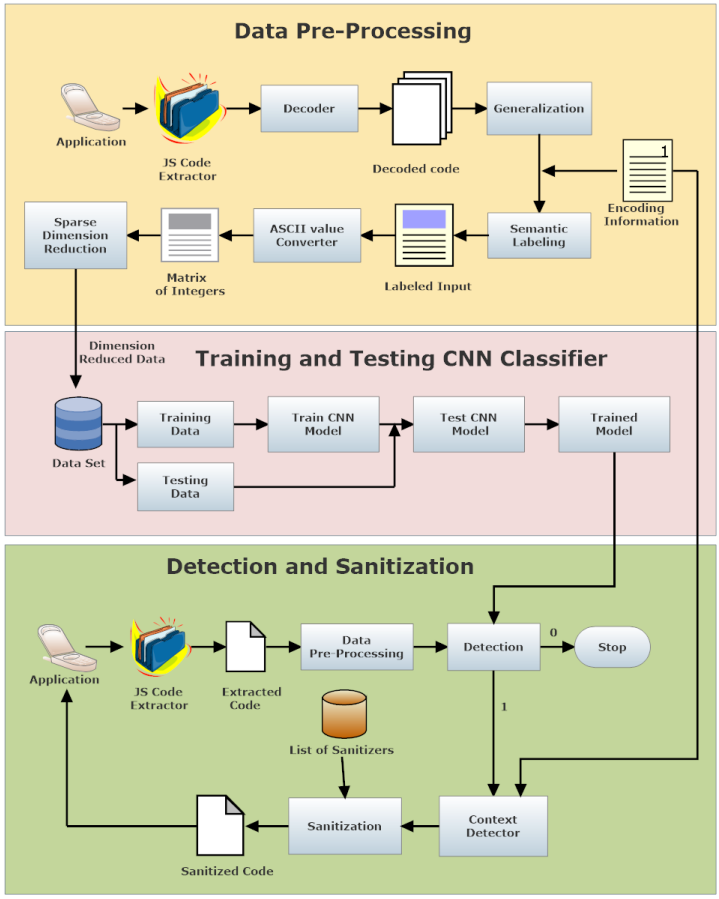

Fig. 4: Architecture of ConvXSS

that are invariant to translation with a provincially restricted receptive field. Adding to this, in the research done by authors of [28], [29] it is shown that Convolutional Neural Nets shows reasonable constructions and are invertible. These algorithms are useful as they are better at the efficiency when compared to the matrix multiplication.

Saying this, we will further move to our next section in which we will discuss about our proposed model in detailed manner.

III. Proposed Framework : ConvXSS

In this section, we propose our model for detection and explain it in a detailed manner. Along with this, we also introduce and explain the context-based sanitization where the malicious code is replaced by sanitized code if any.

The suggested approach is as follows. Initially, we preprocess the data of the input that is the JavaScript code, which includes steps like decoding, generalization, semantic labelling of input and then conversion of the input into binary vectors based on the ASCII values and forming a dataset of 2d matrix of integers. This dataset is broken down into two data sets – one for training and the other for testing. The data set for training is used to train the CNN classifier that we have proposed for the detection purposes and test data is used to find out the accuracy. After the accuracy is maximized based on the change in the layers and epoch values, the model is used to detect malicious code in the app. If found malicious, the code is sent for the context-based sanitization to get rid of the code. The detailed Architecture of the same is given in Fig. 4.

Given in Fig. 5 is the workflow of the model in which the order of execution of the steps involved is explained in a detailed manner. The Start and End states indicates the beginning and the end of data pre-processing and training and testing of the model.

The following subsections contain the detailed discussion of each stage of the above architecture.

A. Data Pre-Processing

In this stage of research, by removing the unnecessary and redundant data and by decoding the encoded data, the cleaning of the data was carried out. Both malicious and benign data are considered and except the blank spaces, each string of the code is labelled based on the semantics and then converted from text to binary matrix. Binary vectors are considered instead of the word vectors as training of word vectors consumes tremendous amount of time and monotonous due to the massive strings. [30] To maintain the consistency in the length of each row, the maximum length of the rows is taken and 0 is appended with other rows in the dataset . Since the dimension of the resulting dataset is huge, we perform sparse dimension reduction to

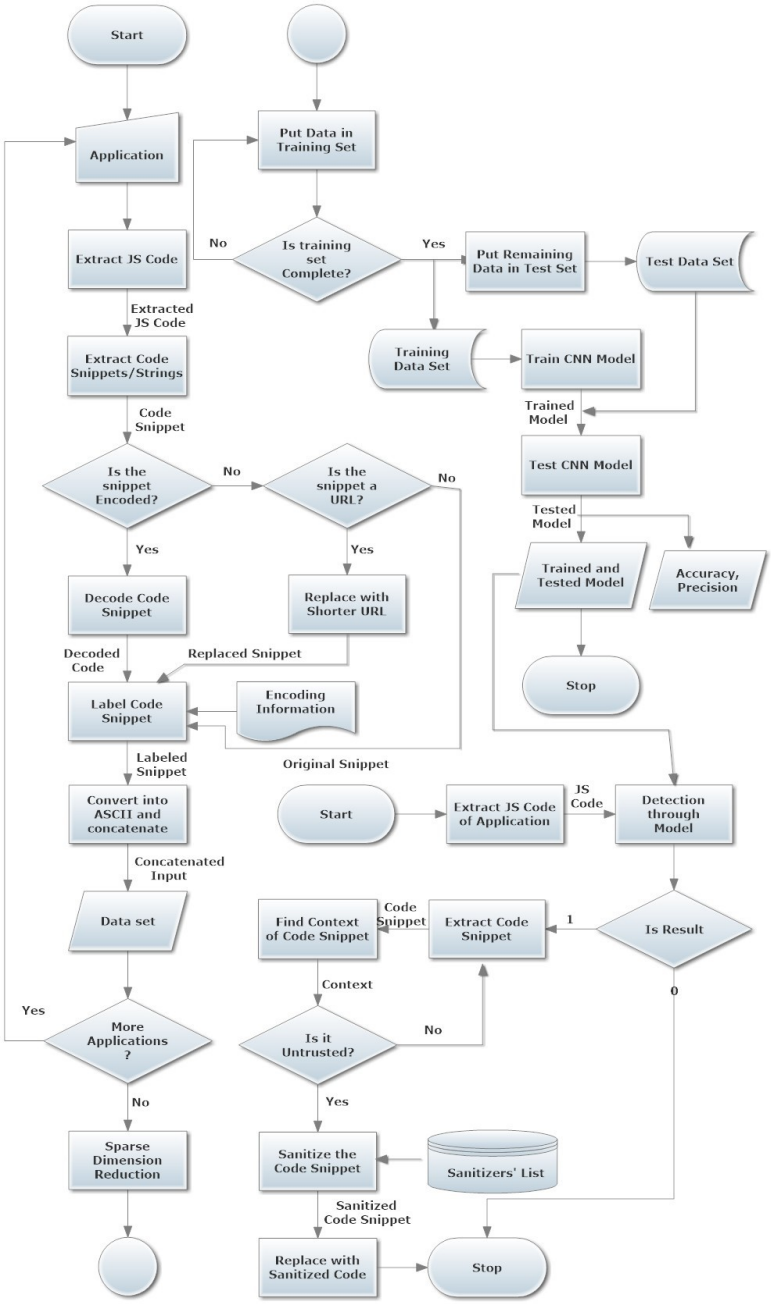

Fig. 5: Work Flow of the Model

finalize the dataset required which is going to be trained and modelled.

The detailed algorithm of the data pre-processing is given in Algorithm 1.

In algorithm 1, the set of JavaScript codes are taken and each code is divided into lines and then into words and depending on the context of the word, each word is either decoded or generalized and then it is labeled based on its context. These labeled words are further converted into ASCII values and then appended into the dataset. Approaches involved in the preprocessing stage like decoding of encoded data, generalization, semantic labeling of strings etc., will be further discussed in detailed manner.

1) Decoder: As mentioned in the Cheat Sheet of XSS Filter Evasion [31], Attackers who inject malicious code try to sidestep conventional filters or validation techniques by utilizing encoding strategies like URL Encoding, Unicode Encoding, Hex Encoding, HTML Entity Encoding , UTF-7 Encoding etc. The following example, which shows HTML encoding is given:

document.write(”<script onmouseover=”alert(1)”>test </script>”);

The character count of the above string is 64. After HTML entity coding, it changes into

document.write("<script onmouseover="alert(1)">test </script>");

Therefore, the character count of the code after encoding is 94. HTML n-coding generally consists of HTML decimal coding and HTML hexadecimal coding. HTML entity has a coding format such that it consists of symbols such as "& lt;& quot& gt;". Regarding HTML decimal coding, the decimal coding is given by appending '& #' with the ASCII value of the character for every character. For example, the code is shown below:

<article draggable="true" ondragstart="alert(1)" >test </article >

The character count of the above code snippet is 63 but after performing the HTML decimal coding, the above code transforms into the snippet as mentioned below:

<article draggable="true" ondragstart="alert(1)">test </article >

The new character count for the same code snippet after decimal coding is 99. Instead if HTML hexadecimal encoding is used, the code snippet transforms into

**<article draggable="true" ondragstart="alert(1)
">test </article >**

whose size is 108. The entire code which used HTML n-coding and HTML entity coding may result in code injection attacks.

So, going through many examples like above patterns, a decoding algorithm is proposed to decode the HTML encoding and convert the code back into its original form. As the encoding involves three characters ';','#','&' along with a number which is the ASCII value of the character that has to be present in the place of the encoded region. Therefore, in the process of decoding, the special characters are ignored and the number is replaced with the corresponding character and the string thus formed is returned. Algorithm 2 is the algorithm for the same.

2) Generalization: In order to reduce the redundant and unnecessary information like the name of websites etc., the string of websites, if present are replaced with "http://website" and then put back into the code.

3) Semantic Labeling of Strings: After the encoded code is decoded and generalized, the code snippet i.e. the string, is semantically labeled based on its context with the help of encoding information provided. This is implemented in our model for the purpose of boost the training and boost the speed of detection. Because the role of the well-trained neural network is to comprehend the character of a particular symbol or operator. Labeling of the string based on its context thus accelerate the process of neural network understanding. The idea of labeling is to transform the original string by appending the type label to it, which therefore will occur more often in the dataset, thus enabling the neural network model to run faster. This approach, quantitatively found to better than simple binary conversion of strings, need not require any modifications to CNN as the neural network continue to be oblivious about the arrangement of the labeled string [30]. This algorithm in which the respective type number based on the labeling information is appended along with the string is given below:

4) Conversion of Strings into the Required Input: In this stage, each character of the string is taken and if uppercase letter, converted into lower case and ASCII character is generated and this process is recurrent for each character in the string and it is stored in the dataset. Then the maximum length of the row is taken and zeroes are appended in the other rows to equalize the length of all rows. The algorithm for the same is mentioned below in algorithm 4:

5) Dimensionality Reduction: As the code for applications are usually more that the dataset generated is enormous and is of high dimension. So, conducive to decrease the computation time of the training of neural network classifier, dimensionality of the dataset should reduce. Though numerous dimension reduction techniques such as Factor Analysis, ICA, PCA can be utilized for diminishing the dimension, Sparse Dimension Reduction technique was used, considering the sparse dataset. This reduction technique helps in reducing the raw data of high dimension to a data of lower dimension by utilizing a sparse random matrix that enables speedy computation of the data projected. Components in the matrix that is random are taken from a distribution over fi1, 0, 1; the probability for elements 1 and -1 is equal to $\frac{1}{2\sqrt{d}}$ each, meanwhile the probability for 0 is $1-(\frac{1}{\sqrt{d}})$, where the raw data's dimensionality is denoted by d. [14]

B. Training and Testing of CNN Model for Classification

Presently, in this part of the process, we divided the reduced dimension data into training and testing data which we used as an input to train and test our deep learning classifier. We have used Convolutional Neural Network for deep learning model as it automatically selects the required features itself and is efficient in dealing with large number of features.[32] Along with this, many models like Naïve Bayes, kNN, Random Forest are also trained and tested for the purpose of comparison. In this section, we will discuss in detail about the CNN classifier model and different layers used in the neural network.

The Convolutional Neural Network consists of either a single or multiple layer of convolutional, succeeded by a pooling layer and it is connected with a single or multiple fully connected layers.[33] The convolutional layer is the elementary unit of Convolutional Neural Network. In the convolutional layer, each neuron is linked to small set of input

Algorithm 1: Data Pre-Processing

Input: Set of JS Codes (Set)

Output: Dataset of Integers (3D array of integers where each 2D array is pre-processed input.)

```
1  begin
2      Dataset ← NULL; // Declaration of Required Variables for further computation.
3      String A ← NULL;
4      String [ ] sa ← NULL;
5      int k ← 0;
6      for ( S ϵSet ) {
           // Application of the algorithm for each JavaScript Code in the set.
7          String [ ][ MAX]B ← NULL;
8          int i ← 0;
9          for ( st ϵS ) {
               // Application of Division of JS code into lines.
10             int m ← 0;
11             String st1 ← StringTokenizer(st,"");
12             int i ← 0;
13             while (st1.hasMoreTokens( )) do
                   // Conversion of lines into array of words for further computation.
14                 sa[ i] ← st1.nextToken();
15                 i ← i + 1;
16             end while
17             int j ← 0;
18             while (sa[ j]! = " < script > ") do
                   // Avoiding Unnecessary tags for computation.
19                 j + +;
20             end while
21             j + +;
22             while (sa[ j]! = " < /script > ") do
                   // Conversion of strings into required format.
23                 if (sa[ j].substring(0, 2) == "&#") then
24                     A ← Decode(sa[j]); // Decoding of the Encoded Strings
25                 end if
26                 else if (sa[ j].substring(0, 7) == "http : //") then
27                     A ← "http : //website"; // Replaced with shorter string to reduce data.
28                 end if
29                 else
30                     A ← sa[ j];
31                 end if
32                 Label(A);// Semantic labelling of the string using encoding information.
33                 B[ l][ m] ← Convert(A); // Conversion of the labelled string into required
                       input.
34                 m + +;
35             end while
36             Dataset[ k] ← B[ l]; // Entry of the converted data into dataset.
37             l + +;
38             k + +;
39         }
40     }
41     Append_Zero(Dataset); // Appending zeroes to each row of dataset for consistent
           length.
42     Sparse_Dimension_Reduction(Dataset); // Reduction technique to reduce dimensionality.
43     return Dataset ;
44 end
45
```

Algorithm 2: Decoding of Data

Input: String (The String to be Decoded)
Output: Decoded String after the computation of the algorithm

```
begin
    String a, Num ← NULL; // Declaration of the variables for computation.
    int[ ]Number ← 0;
    for ( ch ϵS ) {
        if (ch ==′ &′||ch ==′ #′) then
            // Taking each character and ignore if character is '&' or '#'
            Ignore ch;
        end if
        else if (ch>='0'&&ch<='9') then
            // If the character is a number, then appending it with the existing string of numbers.
            Num.append(ch);
        end if
        else if (ch==';') then
            // If the character reaches ';', the string is converted to number and put into the array Num.
            Ignore ch;
            Number ← (int)Num;
            Num ← NULL;
        end if
    }
    for ( n ϵNumber ) {
        // Each number is taken and converted to its respective ASCII and appended to form a string.
        char c ← n;
        a.append(c);
    }
    return a; // Output is the decoded string.
    end
```

Algorithm 3: Labelling of Data

Input: String S, Encoding Information EI

```
begin
    String A ← NULL; // Declaration of a String for Modification
    if (EI[0].contains(S)) then
        // Labelling based on the context of the given string.
        A ← "0";
    end if
    else if (EI[1].contains(S)) then
        A ← "1";
    end if
    else if (EI[2].contains(S)) then
        A ← "2"; // Concatenation of a label based on the context.
    end if
    A.append(S); // Appending the existing string with the label.
    S ← A;
end
```

neurons and the parameters have learnable filters included in them and they get extended through the full depth. Finally, the dot product of the filter's content and input is evaluated for the formation of a two dimensional activation map after which the pooling layer comes into picture. This pooling layer helps in the subsampling of the output produced by the convolutional layer. Max pooling is the largely utilized process for the purpose of pooling. Then the fully connected layer, the layer whose neurons are connected completely to all the previous layer's activation, is used to finally process the output.[34]

The reason to choose CNN among other deep learning classifiers. Since the performance of CNN for image recognition is better than the other classifiers where the pixels of image are converted to numerals and given as an input to the network, it is better to use it for malicious code recognition as the text could also be converted into numerals and given as an input to it. The algorithm for ConvXSS training and testing is as given in algorithm 5.

C. Context Based Sanitization

After the optimization of the model considering the accuracy obtained after the evaluation, ConvXSS is used to detect the malicious code in most of the apps. At this stage, the JavaScript code of the app is taken and after the preprocessing and detection using the CNN classifier, if the code in the app is detected malicious, we sanitize the code. This is a method used to replace the untrusted variables present in the app with the sanitized variables. These variables are sanitized based on their context, thus the name, context-based sanitization. For this, we made sure that all the encoded portion is decoded so that no part of the code could evade the sanitization. For this algorithm to work, we initially provided the algorithm with the list of sanitizers and encoding information and then the code for which the classifier detected as malicious is taken and each string of the code is taken and based on the context of the variable provided by the encoding information, if the context is untrusted, the string is sanitized using the relevant sanitizer from the list of sanitizers. After the sanitization, the sanitized string is replaced with the original string.[35] The algorithm for the same is as mentioned in algorithm 6.

Algorithm 4: Conversion of String

Input: String S (The string to be converted into the required input format)

Output: Integer Array (An array of integers formed after putting every character's ASCII value)

```
begin
    Int[]num ← NULL; // Declaration of Integer Array to store the output of the algorithm.
    if (S[0]=="0") then
        num.push(0); // Putting the value of label into the array based on the context.
    end if
    else if (S[0]=="1") then
        num.push(1);
    end if
    else if (S[0]=="2") then
        num.push(2);
    end if
    for ( ch ϵS ) {
        // Taking each character of the string and converting them into ASCII value
        num.push(ASCII(ch));// Putting the ASCII value into the integer Array.
    }
    return num;
end
```

IV. Implementation:

A. *Dataset Description*

For the training and testing purposes, malicious and benign code strings are collected and a CSV file is formed using the above strings. It is made sure that malicious codes taken contains both regular and encrypted text contents. These codes are further sent for data pre-processing. The dataset consists of 105,470 samples, where 31,407 are benign, and 74,063 are malicious samples.

B. *Implementation Details*

For the implementation of the model, we used Windows 10, Intel(R) Core(TM) i7 CPU @ 1.8 GHz, 16 GB RAM. We have used Kaggle and Google Colab for the training and testing different models mentioned in this paper. We have used python language for the coding in which Keras, an API interface of high-level networks developed with Python, has the advantages of speed, simplicity, convenience of usage and modularity, has been chosen on TensorFlow for training and evaluating the classifier models. After training and testing, we have chosen the best model as ConvXSS based on the maximum evaluative measures and minimal loss value obtained after evaluation using test dataset and comparing the predictions with the given labels in dataset. As mentioned above, for this process, a dataset having malicious and benign code together was taken for drawing comparisons between the different models.After the collection of the data, the following steps are done for the experimentation purpose:

1) Data Pre-Processing
2) Identification of the most appropriate count of hidden layers for the CNN.
3) Maximization of the size of batch with minimization of the epoch numbers
4) Evaluation of the precision, recall and accuracy of the model with the testing dataset and inspection of the models statistically.

For data preprocessing, as mentioned above, we have taken two csv files containing malicious and benign codes and post merging, shuffled them to make the training and testing data set more random. Then, in the aforementioned process we first extracted the code and after applying the respective decoders if needed, we took the binary vectors of the ASCII values of the characters of the string of the codes and then reduced the dimensions of the dataset to avoid the curse of dimensionality and thus formed a dataset.

For dividing the dataset into testing and training sub datasets, and for the evaluation of the model, we used K-fold cross validation technique. In this technique, the performance of classifier model is estimated by dividing the original dataset randomly into k equal sized sub datasets and using k-1 sub datasets for training and 1 sub dataset for testing. This step is repeated for k times such that every sub dataset is used exactly once for testing dataset by the end of k cycles and the model is estimated by taking the average of all the values obtained in k cycles. The edge and superiority of this model lies in the fact that every observation in the dataset is used for testing and also exactly once. Since the accuracy remains similar for different folds, the probability of overfitting in this technique is minimal. For this paper, we have chosen a value of 10 for the K in this cross validation technique and used and chosen the average of evaluative measures for each fold in the model for the final analysis.

For the generation of the model, tuning parameters of appropriate measures are chosen by using different techniques of evaluation. For the CNN model, the count of neurons in the input layer is given by the dataset's largest item. In the convolutional neural network, for this particular implementation, the initial deep layer have to be more extensive than the input layer, and as the model progresses to next layers, the no of neurons in each layer is gradually reduced and when the output layer is reached, it is reduced to one. Though the neural network's mathematical model remains same, the training parameters are changed for purpose of the optimization and to land at the final framework for ConvXSS. This triangular pattern of the CNN ensures that the output layer has only one value.

The epoch value used for our experimentation is 20. Firstly, we fixed the count of hidden layers and changed the quantity of filters in the convolutional layer. We chose the count of hidden layers to be 8 and the initial layer is 1d convolutional layer followed by a pooling layer (max-pooling), and the subsequent layers contain 1d convolutional layer with 100

Algorithm 5: CNN Classifier

Input: Dataset (The dataset taken after the data pre -processing step is completed)
Output: Accuracy, Precision, Recall

begin
  $convnet \leftarrow input\_data(shape = [None, SIZE, SIZE, 1], name =' input');$
  $convnet \leftarrow conv\_1d(convnet, No.of Features, FeatureVectorSize, activation =' relu');$
  $convnet \leftarrow max\_pool\_1d(convnet, FeatureVectorSize);$
  $convent \leftarrow conv\_1d(convnet, No.of Features, FeatureVectorSize, activation =' relu');$
  $convnet \leftarrow max\_pool\_1d(convnet, FeatureVectorSize);$
  $convnet \leftarrow dropout(dropoutfraction);$
  $convnet \leftarrow flatten();$
  $convnet \leftarrow dense(number, activation =' relu');$
  $convnet \leftarrow dense(number, activation =' sigmoid');$
  $model \leftarrow tflearn.DNN(convnet, tensorboard\_dir =' log');$ `// Training of the model in which x is the dataset and y is the corresponding output;`
  $model.fit(Training\_Data(x, y), Parameters);$`// Training the model created.`
  $TP, FP, TN, FN \leftarrow NULL$ `// Declaring the variables for true positive, false positive ,true negative, false negative`
  for (*x,y* $\epsilon Test\_Data$ **)** {
    `// Testing the trained model for every value.`
    $output \leftarrow model.predict(x);$
    if *(output ==1 && y==1)* **then**
      $TP++;$`// Incrementing the respective values based on the output given by model and the actual output.`
    end if
    else if *(output == 0 && y==1)* **then**
      $FN++;$
    end if
    else if *(output == 1 && y==0)* **then**
      $FP++;$
    end if
    else if *(output == 0 && y==0)* **then**
      $TN++;$
    end if
  }
  $Accuracy \leftarrow (\frac{(TP+TN)}{(TP+TN+FP+FN)}) * 100;$ `// Accuracy Calculation based on the values of TN, TP, FP, FN`
  $Precision \leftarrow (\frac{TP}{(TP+FN)}) * 100;$ `// Precision Calculation based on the values TP, FN.`
  $Recall \leftarrow (\frac{TP}{(TP+FP)}) * 100;$ `// Calculation of Recall based on the values TP,FP.`
  return *Accuracy, Precision, Recall;* `// Output values of this algorithm is Accuracy,Precision and Recall.`
end

neurons followed by max-pooling layer, a dropout layer have a dropout value of 0.2, a flatten layer and two dense layers of which one has 250 neurons while the other -the output layer has only one. This model will be further referenced as Model 1. Secondly, we devised a Model 2 in which we didn't change the count of hidden layers but changed the quantity of neurons of the convolutional layer to 200 followed by a pooling layer and the other layers remained to be same. The experimental result for the same is shown in table. Thirdly, for the model referenced as Model 3, the count of hidden layers are unaltered and the count of neurons of fully connected layers are changed to 300 retaining the count of neurons in the rest, i.e, the convolutional layers, max pooling layers, dense layers same. Subsequently, the number of layers is changed in the model and the accuracy and precision for these models are as mentioned in the Table VI. A detailed evaluation of the above models is discussed in the next section. (various number of hidden layers spanning from 8 to 10 are used, changing the values for every iteration.).

C. *Experimental Evaluation of ConvXSS*

After the training, the testing part is done to calculate the accuracy, precision and recall of ConvXSS. These performance evaluation metrics along with others such as the misclassifica-

Algorithm 6: Context Based Sanitization

Input: JavaScript Code, List of Sanitizers, Encoding Information

Output: JS Document (After the sanitization of the untrusted variables)

```
begin
    String[ ]U ← NULL // Declaration of a string for purpose of modification.
    for ( line sl ϵJS ) {
        // Taking each line of the code into consideration
        for ( S ϵsl ) {
            // Considering each word in the line taken
            int l ← label_integer(S) // Taking the labelled integer of the word into account
            if (l==untrusted_num) then
                U.push(S);// If the context of the string is untrusted, the word is listed
                Replace(Sanitize(S),S);
                    // Respective sanitizer is applied based on context and replaced.
            end if
        }
    }
    return JS // After the completion of sanitization of the variables and replacement, the code is returned.
end
```

tion rate (Error rate) and AUC curve are calculated using the True Negative (TN),True Positive (TP), False Positive (FP) and False Negative (FN). The performance of the proposed framework is evaluated using the Confusion Matrix, given in image and tabular form in V and Figure True Negative(TN) is the count of benign codes rightly classified as benign, True Positive (TP) is the count of malicious code samples that are exactly classified as positive. False Positive (FP) is the inaccurate categorization of benign code samples as malicious and False Negative (FN) is the inaccurate classification of malicious code as benign. For the establishment of additional clarification, Fig. 6, is presented to bring four different cases as mentioned by [44], where these four cases can represent the detection models that use artificial intelligence techniques. The four cases are as follows, Fig.14 (a) represents low recall-low precision which leads to a high FP and FN, Fig.14 (b) represents high recall low precision resulting in high Fp rate and low FN rate, Fig.14(c) represents low recall-high precision leading to high FN and low FP, or Fig.14 (d) represents

TABLE I: Evaluative Measures of the Model 1 for different folds

CNN Model 1			
Evaluation	**Accuracy**	**Precision**	**Recall**
Fold 1	97.7866352	97.7655231	98.0902314
Fold 2	98.3178437	98.3915687	98.4461665
Fold 3	98.8784015	98.8002300	99.0766644
Fold 4	98.6865461	98.5807896	98.9860237
Fold 5	99.3358970	99.4206965	99.3384838
Fold 6	99.2178321	99.2961645	99.2692828
Fold 7	99.2916226	99.2161274	99.4388402
Fold 8	99.2621064	99.5977521	99.0664184
Fold 9	99.3801713	99.5587468	99.2849290
Fold 10	98.9817083	98.5597790	99.5608091
Average	98.9138764	98.9187378	99.0557849

TABLE II: Evaluative Measures of the Model 4 for different folds

CNN Model 4			
Evaluation	**Accuracy**	**Precision**	**Recall**
Fold 1	98.1260180	97.5709617	98.9482403
Fold 2	98.6719787	98.2957661	99.2230773
Fold 3	99.0112245	98.7218678	99.4124234
Fold 4	99.1292894	99.1511524	99.2326677
Fold 5	99.4539618	99.2863119	99.6968031
Fold 6	99.4096875	99.4052529	99.5128572
Fold 7	99.2325902	99.0502775	99.4949520
Fold 8	99.4982362	99.6524990	99.4398534
Fold 9	99.4687200	99.5321989	99.4774461
Fold 10	99.4982362	99.4248211	99.6431589
Average	99.14999425	99.0091109	99.4081479

high recall-high precision which is the best case scenario, representing low FP and FN rates as given by our classifier.

Fig. 6: The Four Cases of Model Performance

Precision is the percentage of true positives in the total of false positive and true positive. Therefore, a low precision indicates higher percentage of false positives. Accuracy is the percentage of total true positives and true negatives among all

TABLE III: Evaluative measures for models with different Hidden Layers and Number of Neurons

Model No.	Number of Hidden Layers	Number of Neurons*	Overall Accuracy	Precision	Recall
1	8	100c,0.2dp,100c,fl,250d,1d	98.9138764	98.9187378	99.0557849
2	8	200c,0.2dp,100c,fl,250d,1d	98.8976467	98.8437295	99.102354
3	8	300c,0.2dp,100c,fl,250d,1d	98.6497152	98.5406047	98.9487636
4	10	100c,100c,0.2dp,100c,fl,250d,1d	99.1499942	99.0091109	99.4081479
5	10	200c,100c,0.2dp,100c,fl,250d,1d	99.0791547	99.0867633	99.1955662
6	10	300c,100c,0.2dp,100c,fl,250d,1d	99.0319312	99.0377957	99.1575849

*c: Conv1D + MaxPool1D, dp: Dropout, fl: Flatten, d: Dense

TABLE IV: TP, FP, FN, TN

	Actual Malicious Code	Actual Benign Code
Predicted Malicious Code	True Positive (TP)	False Positive (FP)
Predicted Benign Code	False Negative (FN)	True Negative (TN)

the cases given. Recall is the probability of the retrieval of an accurate item. Putting it in another way, precision calculates the classifier's exactness whereas recall can be considered as the classifier's completeness. As mentioned above, these are given by the formulas:

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

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

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

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

$$F-Score = \frac{2*Precision*Recall}{Precision+TPR}$$

$$MisclassificationRate = \frac{FP+FN}{TP+TN+FP+FN}$$

$$AreaUndertheCurve(AUC) = \frac{1}{2} \times (\frac{TP}{TP+FN} + \frac{TN}{TN+FP})$$

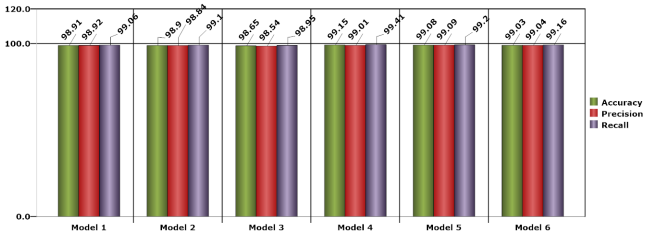

Fig. 7: Evaluation of Different Models

After the implementation of different of models, the testing data is used to test the aforementioned models for calculating the parameters mentioned above. The accuracy, precision, recall for the model 1,comes out to be 98.92%, 98.92% and 99.06% respectively, with 100 filters in the first convolutional layer and 100 in the second one. Whereas the accuracy, precision and recall for the model 2 comes out to be 98.90%, 98.84%, 99.10% respectively, with 200 number of filters in the convolutional layer keeping the rest same. The accuracy, precision and the recall for the other models are mentioned using a bar graph against their respective models in Fig 7.

D. *Performance Analysis*

From this analysis of different models, we can understand that the features that might influence the process of optimization include count of neurons in one or more hidden layers in a neural network, a number of training cycles, size of the batch of the patterns forwarded from the dataset to the input of the neural network, quantity of hidden layers, types of the optimizers. The aforementioned indicators such as True Positive Rate (TPR) or Recall, F- Score, Precision, and Overall Accuracy (Acc) are used to assess if a the algorithm of detection is good or bad. The system could be evaluated as a better model when the TPR is higher and FPR is lower while the precision indicates the predictive power of the algorithm. So going through the evaluative measures mentioned in the previous table and after evaluating the accuracy, precision, recall of the different models, we could say that the Model 4 has performed well among all the models as it has better accuracy, larger precision, greater recall (and smaller FPR). The accuracy of this Model 4 is far better than the models mentioned in papers like [14], [30], and thus it forms the basis of implementing ConvXSS here onwards. A comparative analysis of our model with models of other papers is in the Table. V below.

TABLE V: Comparision with Models of other papers

Model	Accuracy Overall	Precision	Recall
Our Model - ConvXSS	99.42	99.81	99.35
CODDLE - S. Abaimov and G. Bianchi [30]	95.7	99.0	91.2
Selvam, M. K. [21]	98.54	98.65	98.40
Wang, Y., Cai, W. D., Wei, P. C. [14]	94.82	94.9	94.8
DeepXSS - Fang, Y., Li, Y., Liu, L., Huang, C. [36]	99.5	97.9	98.7
Adaboost - Wang, R., Jia, X., Li, Q., Zhang, S. [37]	94.1	93.9	93.9

Also, considering the time for prediction and the pre-processing time, the average time taken to recognize an application is far lesser than the time taken for prediction.

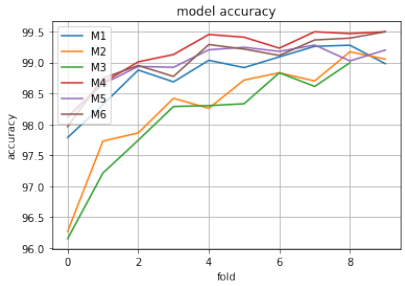

Fig. 8: Accuracy vs Fold value

To corroborate our results about Model 4 being optimum, the Accuracy, Precision and Recall values for the 6 models were plotted against the Fold value, which was varied from 1 to 10 as mentioned earlier. A comparative analysis is given via these curves in Fig 8 , 9 and 10.

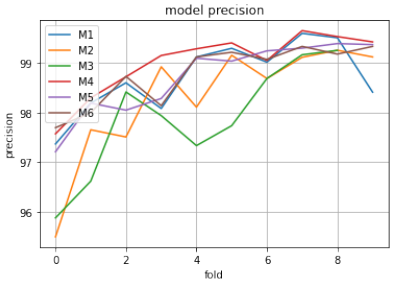

Fig. 9: Precision vs Fold Value

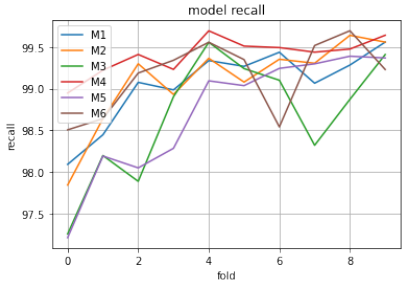

Fig. 10: Recall vs Fold Value

E. A Comparative study with other models

Along with the CNN models mentioned above, we have also trained and tested other classifiers for the purpose of comparison. We have trained and tested models like Naive Bayes, K-nearest neighbors and Random Forest (RF), SVM. Naïve Bayes classifiers are simple probabilistic classifiers that run on the application of Bayes' theorem with naïve independent assumptions considered between the features. They are convenient for any problem of classification. As the Gaussian Naïve Bayes executes well with data that is normally distributed , we have chosen it over other Naïve Bayes models. The other model used widely for classification problems is the K nearest neighbors (KNN). But it is a challenge to choose the optimal K value. After comparing the accuracy, training error rate and validation, we chose the optimal K which turned out to be 9 after training different models with different numbers and considered the evaluation measures for comparison. Random Forest is also chosen by many for the classification problems as the model constructs many hierarchical decision trees classifiers taking various sub samples of data-set for prediction. The tuning parameters for the decision tree are the tree depth and number of trees. (SVM, a supervised learning model is associated with learning algorithms for analyzing data and recognition of patterns). After training and testing all these models, we have compared these models with CNN model and the comparison for the same is given in Fig 11. As we can see in the above bar graph,

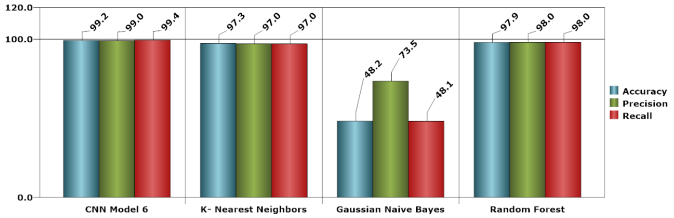

Fig. 11: Comparison with other models

CNN provides slightly higher accuracy than other models. All the CNN models performed better than KNN, Random Forest and Gaussian Naïve Bayes models out of which Model 4 has better performance among all the CNN models.

F. Final Evaluation and Performance Statistics

An extended adaptation of Model 4 was trained on our main dataset, that comprised of about double the number of code strings as used before to obtain the deisred model - ConvXSS. We collected 31,407 benign and 74,063 malicious samples with a total size of about 2 GB. To prevent duplication of data from multiple data sources, we calculated the MD5 of each sample to ensure that each sample is unique.The data extracted consisted of 1,05,470 attack vector code snippets fed as [Code,Label] pairs to our Model.

For an unbiased sense of proposed model efficiency, the dataset was split randomly and separately into three parts with a partition ratio of 60%: 20%: 20% for training, validation, and testing sets. The training set includes 63,281 samples labeled as [0: Benign, 1: Malicious], the validation set contains 21,094 samples, and the holdout set consists of 21,095 samples, which is used only to estimate the effectiveness of the final and fully trained model. Table I shows the detailed partitions of the dataset.

The pre-processing steps remained similar to Model 4, with application of decoders and converting the code strings to the binary vectors of their ASCII values. Resizing and scaling this processed data into a 100x100 image like format using OpenCV led to an "image" like format, one that was easily fed into the model.The next step involved using a Keras data generator for generating multiple cores in real time and

TABLE VI: Dataset Subdivisions

DATA			
Name	**Benign**	**Mali-cious**	**Total**
Training Dataset	18,844	44,437	63,281
Validation Dataset	6,281	14,813	21,094
Testing Dataset	6,282	14,813	21,095
Total Dataset	31,407	74,063	1,05,470

feeding it into our deep learning model to keep the process memory efficient and batch-wise operable. The convolutional neural network is implemented in Python with Tensor Flow 2.0 and Keras. Since malicious JavaScript detection is a binary classification task, we use cross-entropy as the loss function, i.e.

loss = -ylogy' + (1 - y)log(1 - y')

where y (0 for benign and 1 for malicious) is the label of the sample and y' is the output probability of the sample being malicious. We adopt Adam as an optimizer, which is one of the most efficient optimizers at present. The model architecture involved 3 convolutional layers and 11 layers in total, and the features include using an Adam optimizer and a binary cross-entropy loss as the best suited for our use-case (2 labels).

ConvXSS was trained for 20 epochs post which the loss curves showed signs of over-fitting. The maximum validation accuracy achieved for comes out to be comparable with the state of art papers we have explored. The performance of this profound model is better than six ML algorithms in Table V. The proposed model achieved an accuracy rate of 99.42%, recall rate of about 99.35% and precision value of about 99.81%. Figures 16 and 17 represent the Loss and Accuracy curves that can help gauge the performance of the proposed approach. Further, we also studied the effect of the learning rate versus the loss-function as shown in Fig. 12, and plotted both the Cross-Entropy and the Classification Error against the Epochs for which the model was trained and tested. They are depicted via Figures 13 and 14.

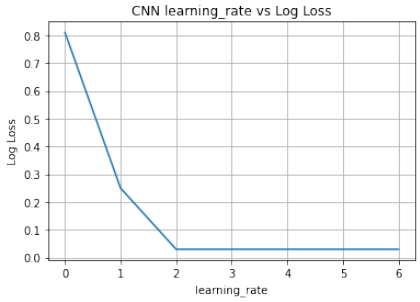

Fig. 12: Learning Rate vs Log Loss

Fig. 15 shows the result of the confusion matrix with a complete statistical view. We can observe that ConvXSS is at par with the state of the art for a Deep Learning Classifier. It also steadies significantly of achieving high precision and high recall simultaneously.

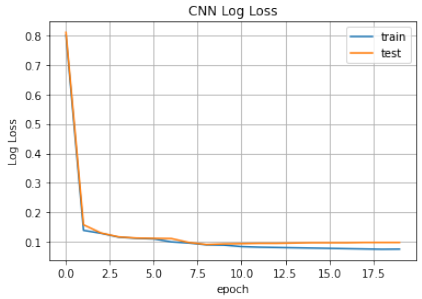

Fig. 13: Cross Entropy vs Epochs

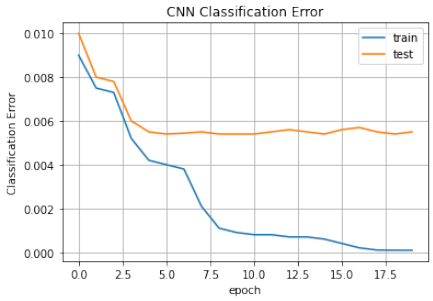

Fig. 14: Classification Error vs Epochs

G. *Evaluation of the Context-Based Sanitization:*

After the pre-processing of the data and the evaluation of ConvXSS post training and testing, we used the model for detection of malicious code and used context-based sanitization if the model detected the code as malicious. For this, the code is extracted from the application and pre-processed and sent to the model for the purpose of detection. If the model classifies the code as malicious , the semi pre-processed code (the code taken after it is decoded) is taken and is tokenized. Each token is taken, and the context is determined using the semantic labeling information used in the semantic labeling stage of the data pre-processing and also using the features in Table.VIII. Table.VIII highlights the features that have been extracted. If the context is untrusted, appropriate sanitizers are applied to sanitize the variable and replace it with the existing variable.

For this process, the list of the context-based sanitizers has been taken from the ESAPI library provided by OWASP which consists of all the possible sanitizers. Post Sanitization, we put the sanitized code back in our database for ConvXSS which tested as benign. This proved that the sanitization technique has been effective and the model is also detecting the malicious and benign code effectively.

V. Conclusion

One of the important factors smart cities is the security and the privacy of users living in it. So, as a part of the issues of security and privacy, we have chosen one of the key topic of Web Application Security, the detection of the Cross-Site Scripting. With the XSS Vulnerabilities, looting user info, and potentially significant security issues happen in the web applications. In this paper, we studied the potential

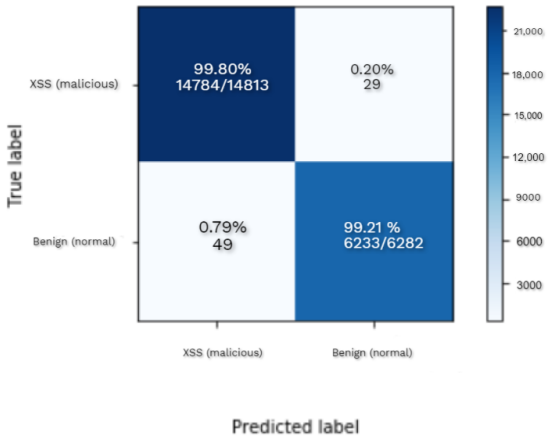

Fig. 15: Confusion Matrix on the Test Dataset

TABLE VII: The performance of existing learning models with ConvXSS

Comparison of the Proposed Model			
Techniques	**Accuracy**	**Precision**	**Recall**
Random Forest	97.90	98.00	98.00
Naive Bayes	48.20	73.50	48.10
Support Vector Machines	95.80	95.50	95.80
k-Nearest Neighbors	97.30	97.00	97.00
BLSTM [38]	92.01	94.23	93.03
CNN-LSTM [39]	99.30	99.90	99.10
Proposed: ConvXSS	**99.42**	**99.81**	**99.35**

risk by XSS attacks imposed on mobile applications, their impact on users of ICT and also introduced code injection attacks which are more hidden. We identified many unique channels like Barcode, Contacts, File Contents etc., through which the code could be injected and spread, apart from the obvious attacks like displaying of code on the URL or on the screen of the application. After that, we introduced a COnvXSS, a novel approach for detection of the malicious code injected that uses Convolutional Neural Networks at the core of it's detection mechanism. This approach has its novelty in the pre-processing stage of semantic labeling of the input which enabled the reduction of the pre-processing time as the labeling of each token of the code helps the CNN model to understand the role of the particular token. In this model, initially, both malicious and benign code strings are taken for the dataset and after decoding, generalizing, and labeling, the code is converted into the binary vectors and the dataset thus formed is divided into the training, validation testing datasets which is used to train the CNN model that we proposed for the detection purposes. The accuracy and the precision of the model came out to be 99.42% and 99.81% as mentioned in the section of evaluation analysis above which we proved that is better than the other models like kNN, Random Forest in the comparative analysis section using various mathematical models like graphs and relevant parameter-plots. These evaluative measures proved that the aforementioned novelty of the model gave better results than that of the previous models, and even those using BiDirectional LSTMs[38] and hybrids of CNN-LSTMs[39] as mentioned in Table V. After the evaluation of the model, we have also

TABLE VIII: Name of Different Features

No.	**Feature Name**	**No.**	**Feature Name**
F0	url_size	F25	html_blur_attr
F1	url_symbols_special	F26	html_onclick_evnt
F2	url_script_tag	F27	html_onchnge_evnt
F3	url_source_attr	F28	html_onerr_evnt
F4	url_cookies	F29	html_onfcs_evnt
F5	url_onmouse_evnt	F30	html_onkeyprss_evnt
F6	url_keyword_number	F31	html_onkeyup_evnt
F7	url_cache	F32	html_onload_evnt
F8	url_domnum	F33	html_onmousedwn_evnt
F9	html_script_tag	F34	html_onmouseup_evnt
F10	html_embed_tag	F35	html_onmouseovr_evnt
F11	html_site_tag	F36	html_onsubmit
F12	html_frame_tag	F37	html_evilkeynum
F13	html_form_tag	F38	html_size
F14	html_object_tag	F39	js_file
F15	html_style_tag	F40	js_psdprtcl
F16	html_data_tag	F41	js_domlction
F17	html_image_tag	F42	js_domdoc
F18	html_areaoftext_tag	F43	js_prpdata
F19	html_bckgrnd_attr	F44	js_prprefrr
F20	html_code_attr	F45	js_writemethod
F21	html_prfl_attr	F46	js_alertmethod
F22	html_source_attr	F47	js_cnfrmmethod
F23	html_usemap_attr	F48	js_deffunc
F24	html_equivhttp_attr	F49	js_funccall
F25	html_blur_attr	F50	js_maxstrlngth

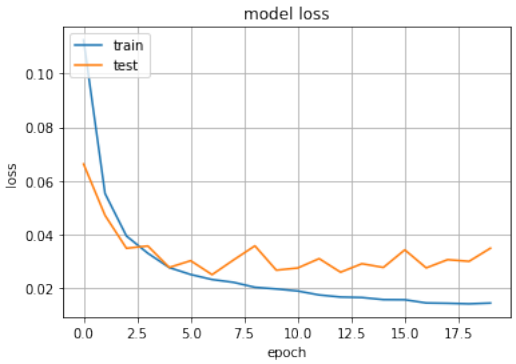

Fig. 16: Loss versus Number of Epochs.

included the Context-Based Sanitization in the paper, in which, after the code is taken from an application, we sanitized the malicious part of the code using the list of sanitizers, if the code is classified as malicious by the model. Some of the prospective possibilities in which the work could be performed in the future, are listed out in the following points:

1) ConvXSS could be utilized to build a powerful projection system for applications in real life for the safety and

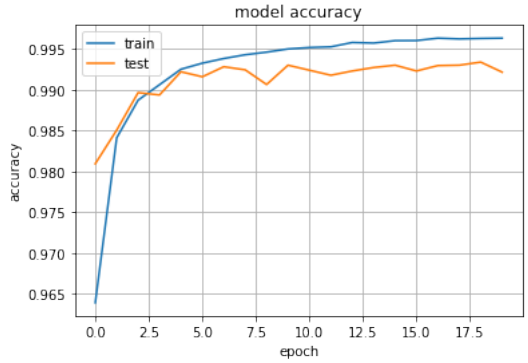

Fig. 17: Accuracy versus Number of Epochs.

privacy of sustainable smart cities. Though, in actual implementation in real life situations, there would be many problems, like insufficient data, disturbances like noise in data , that makes the traditional ML algorithms not practical, our model can be used to make a browser extension or as an app that scans the other downloaded apps in the background.

2) Crawling the website pages in real-time and drawing out the JS code in the website page which can be fed into the model to get notified about the security quotient of the webpage.
3) In general, all learning-based malicious JavaScript detectors do fail to detect some attacks,such as malicious scripts not containing any features in the current training dataset. In this paper,the malicious samples come from a exhaustive mix of data currently present on the Web. As previously mentioned, as the innovation and technology advances in smart cities, innovative attack vector designs keep coming up with time, and learning-based models pose a setback in detecting these new attacks. We plan to continually add new malicious JavaScript samples in the follow-up study to improve the performance of ConvXSS, and work on building an end-to-end Deep Learning based solution for defending Android-based Web Applications against XSS Attack Vectors.

Acknowledgements

The work is partially funded by research grant from Data Science Council of India.

Declarations

Funding: This work is economically supported by Data Security Council of India.
Conflicts of interest/Competing interests: The authors declare that their is no conflict of interests.
Availability of data and material: N/A
Code availability: Code will be available based on the formal request to the corresponding author.

References

[1] R. Khatoun and S. Zeadally, "Cybersecurity and privacy solutions in smart cities," *IEEE Communications Magazine*, vol. 55, no. 3, pp. 51–59, 2017.
[2] A. S. Elmaghraby and M. M. Losavio, "Cyber security challenges in smart cities: Safety, security and privacy," *Journal of advanced research*, vol. 5, no. 4, pp. 491–497, 2014.
[3] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 66–77.
[4] A. van der Stock, B. Glas, N. Smithline, and T. Gigler, "Owasp top 10-2017 the ten most critical web application security risks," *Creative Commons*, 2017.
[5] J. Liu, K. Li, D. Zhu, J. Han, and K. Li, "Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, pp. 1–25, 2016.
[6] G. Schwenk, A. Bikadorov, T. Krueger, and K. Rieck, "Autonomous learning for detection of javascript attacks: Vision or reality?" in *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 93–104. [Online]. Available: https://doi.org/10.1145/2381896.2381911
[7] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 641–654.
[8] M. Alazab, S. Venkatraman, P. Watters, and M. Alazab, "Information security governance: the art of detecting hidden malware," in *IT security governance innovations: theory and research*. IGI Global, 2013, pp. 293–315.
[9] H.-G. Kim, D. Kim, S.-J. Cho, M. Park, and M. Park, "Efficient detection of malicious web pages using high-interaction client honeypots." *Journal of Information Science & Engineering*, vol. 28, no. 5, 2012.
[10] Y. Alosefer and O. Rana, "Honeyware: A web-based low interaction client honeypot," *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 410–417, 2010.
[11] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 281–290. [Online]. Available: https://doi.org/10.1145/1772690.1772720
[12] M. Egele, E. Kirda, and C. Kruegel, "Mitigating Drive-by Download Attacks: Challenges and Open Problems," in *iNetSec Open Research Problems in Network Security*, Zurich, Switzerland, April 2009.
[13] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," in *USENIX Security Symposium*, 2009.
[14] Y. Wang, W.-d. Cai, and P.-c. Wei, "A deep learning approach for detecting malicious javascript code," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534, 2016.
[15] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," in *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2009, pp. 47–54.
[16] S. Rathore, P. K. Sharma, and J. H. Park, "Xssclassifier: An efficient xss attack detection approach based on machine learning classifier on snss." *Journal of Information Processing Systems*, vol. 13, no. 4, 2017.
[17] F. A. Mereani and J. M. Howe, "Detecting cross-site scripting attacks using machine learning," in *International Conference on Advanced Machine Learning Technologies and Applications*. Springer, 2018, pp. 200–210.
[18] M. K. Gupta, M. C. Govil, and G. Singh, "Predicting cross-site scripting (xss) security vulnerabilities in web applications," in *2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 2015, pp. 162–167.
[19] V. K. Malviya, S. Saurav, and A. Gupta, "On security issues in web applications through cross site scripting (xss)," in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1. IEEE, 2013, pp. 583–588.
[20] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[21] M. K. Selvam, "Classification of malicious web code using deep learning," Ph.D. dissertation, Dublin, National College of Ireland, 2018.

[22] K. Li, X. Tang, B. Veeravalli, and K. Li, "Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems," *IEEE Transactions on computers*, vol. 64, no. 1, pp. 191–204, 2013.

[23] X. Xiao, R. Yan, R. Ye, Q. Li, S. Peng, and Y. Jiang, "Detection and prevention of code injection attacks on html5-based apps," in *2015 Third International Conference on Advanced Cloud and Big Data*. IEEE, 2015, pp. 254–261.

[24] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[25] "Darktrace." [Online]. Available: https://www.darktrace.com/en/resources/

[26] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems*, vol. 19, pp. 153–160, 2006.

[27] N. Le Roux and Y. Bengio, "Deep belief networks are compact universal approximators," *Neural computation*, vol. 22, no. 8, pp. 2192–2207, 2010.

[28] A. C. Gilbert, Y. Zhang, K. Lee, Y. Zhang, and H. Lee, "Towards understanding the invertibility of convolutional neural networks," *arXiv preprint arXiv:1705.08664*, 2017.

[29] R. Yan, X. Xiao, G. Hu, S. Peng, and Y. Jiang, "New deep learning method to detect code injection attacks on hybrid applications," *Journal of Systems and Software*, vol. 137, pp. 67–77, 2018.

[30] S. Abaimov and G. Bianchi, "Coddle: Code-injection detection with deep learning," *IEEE Access*, vol. 7, pp. 128 617–128 627, 2019.

[31] X. OWASP, "Filter evasion cheat sheet," 2017. [Online]. Available: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[32] C. Chen, K. Li, S. G. Teo, X. Zou, K. Li, and Z. Zeng, "Citywide traffic flow prediction based on multiple gated spatio-temporal convolutional neural networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 14, no. 4, pp. 1–23, 2020.

[33] J. Chen, K. Li, K. Bilal, K. Li, S. Y. Philip *et al.*, "A bi-layered parallel training architecture for large-scale convolutional neural networks," *IEEE transactions on parallel and distributed systems*, vol. 30, no. 5, pp. 965–976, 2018.

[34] M. Duan, K. Li, and K. Li, "An ensemble cnn2elm for age estimation," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 758–772, 2017.

[35] P. Chaudhary and B. Gupta, "A novel framework to alleviate dissemination of xss worms in online social network (osn) using view segregation," *Neural Network World*, vol. 27, no. 1, p. 5, 2017.

[36] Y. Fang, Y. Li, L. Liu, and C. Huang, "Deepxss: Cross site scripting detection based on deep learning," in *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, 2018, pp. 47–51.

[37] R. Wang, X. Jia, Q. Li, and S. Zhang, "Machine learning based cross-site scripting detection in online social network," *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pp. 823–826, 2014.

[38] X. Song, C. Chen, B. Cui, and J. Fu, "Malicious javascript detection based on bidirectional lstm model," *Applied Sciences*, vol. 10, no. 10, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/10/3440

[39] R. Kadhim and M. Gaata, "A hybrid of cnn and lstm methods for securing web application against cross-site scripting attack," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 21, p. 1022, 02 2021.