Sustainable Cities and Society 80 (2022) 103765

Contents lists available at ScienceDirect

Sustainable Cities and Society

journal homepage: www.elsevier.com/locate/scs

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

Koundinya Kuppa [a], Anushka Dayal [a], Shashank Gupta [a,*], Amit Dua [a], Pooja Chaudhary [b], Shailendra Rathore [c,d]

[a] Department of Computer Science and Information Systems, Birla Institute of Technology and Science, Pilani, Rajasthan, 333031, India
[b] Department of Computer Engineering, NIT Kurukshetra, Kurukshetra, India
[c] Faculty of Science and Engineering, University of Plymouth, United Kingdom
[d] Division of Cyber Security, Abertay University, United Kingdom

ARTICLE INFO

Keywords:
Sustainable smart cities
Security
Privacy
ICT
CPS
Web security
Deep learning
Data preprocessing
Training and testing
Neural Networks
Sanitization
CNN
XSS attack
Malicious code
Code injection attack

ABSTRACT

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 sustainable 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.

1. 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 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 sustainable 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 sustainable smart cities (Khatoun & Zeadally, 2017). To improve the social and economic quality of people's lives, improve the efficiency of services, maintain city operations, we have to build smarter and more sustainable cities for which we have to use the Information and Communication Technology (ICT) (Chen, Wawrzynski, & Lv, 2020; Silva, Khan, & Han, 2018). So, in this process of building sustainable smart cities and to grab the advantages of the technological advances, there is an increase in the requirement of the development of many smart appliances and applications. Of all these, mobile applications play an important role in ICT. To cater to a larger number of people in sustainable smart cities, developers have to develop the app in different languages to support platforms like Android and iOS which involves too much time and labor (Elmaghraby & Losavio, 2014). Therefore, 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. But, as every coin has two sides, to keep up with the rapid growth in the technology to build sustainable smart cities, many

* Corresponding author.
E-mail addresses: f20180283@pilani.bits-pilani.ac.in (K. Kuppa), f20170902@pilani.bits-pilani.ac.in (A. Dayal), shashank.gupta@pilani.bits-pilani.ac.in (S. Gupta), amit.dua@pilani.bits-pilani.ac.in (A. Dua), pooja_6180102@nitkkr.ac.in (P. Chaudhary), shailendra.rathore@plymouth.ac.uk, s520967@uad.ac.uk (S. Rathore).

https://doi.org/10.1016/j.scs.2022.103765
Received 16 July 2021; Received in revised form 19 December 2021; Accepted 4 February 2022
Available online 23 February 2022

Table 1
Ranking of various attacks on applications used in sustainable smart cities.

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

developers today are in the haste to develop mobile applications faster to stay in the competition. While issues and challenges like waste management, transportation and environmental protection are in the center of attention for debates about sustainable smart cities' development, the aspects of security and crime prevention are frequently neglected resulting in an increase in the security and privacy risks involved in mobile applications (Laufs, Borrion, & Bradford, 2020). To understand about the risks, we need to first understand what the difference between security and privacy is. Security as a concept, not a whole, is a vigorous, truthful effort for the prevention of harm to the inhabitants of the sustainable smart cities through physical and digital relationships. Regarding privacy, it is the sum whole of the activities that have the substantial level of safety for self (Braun, Fung, Iqbal, & Shah, 2018). 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 sustainable smart cities. Already, many users have fallen prey to such attacks compromising their sensitive information to the attackers in the end (Jin et al., 2014; Rahman et al., 2020). 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. In the case of sustainable smart cities, these attacks are getting more dynamic in nature due to the increased interconnectivity among the users of owing to new ICT technologies available for them, thus making it more difficult for the developers to tackle them (Elnour, Meskin, Khan, & Jain, 2021; Lever & Kifayat, 2020; Liu, Li, Chen, & Hua, 2021; Liu, Li, Zhu, Han, & Li, 2016). These attack methodologies will be discussed in the further sections.

Table 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 from which the severity of the attacks could be understood (van der Stock, Glas, Smithline, & Gigler, 2017). 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.

1.1. About ConvXSS

Through this paper, we present a procedure that banks on Convolutional Neural Networks — CNN for detection of the malicious code for the safety of users residing in Sustainable Smart Cities. Our novelty lies in the pre-processing phase, in which each line of code is taken and if encoded, is decoded, and is then generalized so that unnecessary randomness can be removed. Then, each word of the code is semantically labeled based on the type of the string and using existing labeling information given. This labeled information is then converted into binary numbers instead of directly converting the strings into numerical data. Thus the dataset is 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. Few comparative studies have been done between different models that have been built by tweaking the hyper-parameters for the identification of the optimum results. 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. This paper also includes sanitization of the malicious code based on the context found using a list of sanitizers. Background knowledge about the attacks is discussed in the next section.

2. XSS attacks in sustainable smart cities

Almost all the data in the sustainable smart cities use open or shared medium like Wi-fi for the purpose of communication in sustainable smart cities. Also, for the surveillance of sustainable smart cities, a

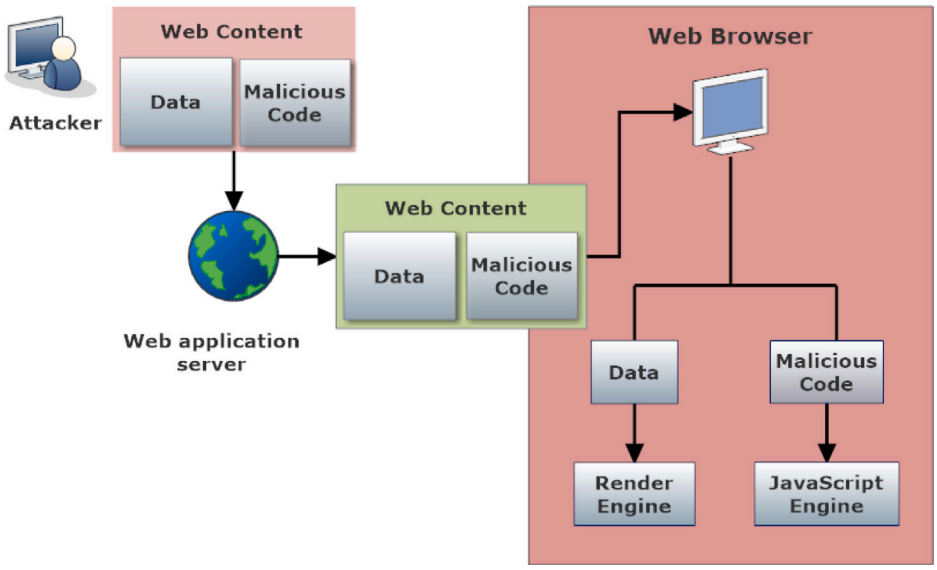

Fig. 1. XSS attacks on ICT web applications in sustainable smart cities.

gigantic volume of confidential data has to be transferred over the network through various applications (Mishra, Puthal, & Tripathy, 2021). But the aforementioned application development technologies host a threatening feature which allows blending of data and code together and could be exploited by the attackers using sophisticated attacks like Cross-Site Scripting (XSS). So, this could be considered a high risk for security and privacy of citizens of the sustainable smart cities. For the injection of code, web apps have a single medium; that is via the web site thus called "cross-site". Disastrously, this feature's repercussion lies in the fact where if the developers are not cautious, the hidden code blended with the data can be triggered impulsively and accidentally. As shown in Fig. 1, In the XSS attack, the attackers inject a malicious code into the web app via a web form or web request which would be 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. For example, in sustainable smart cities, in construction business, there are many applications which help in the automation of many processes. But, people with malevolent intention, can steal the users as well as the app's information to cause deliberate destruction and derange operations and also access unauthorized files (Mantha, de Soto, & Karri, 2021). This might lead to huge loss of property and lives.

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, in Reflected XSS, 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. This may lead to the malicious code remaining imperceptible for a long time, possibly forever. The malicious code present in the data is not displayed. Instead, it is executed by the JavaScript engine leading to the rise in the number of victims. (Mohammadpourfard, Khalili, Genc, & Konstantinou, 2021; Wang, Cai, & Wei, 2016).

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 Fig. 2 the code injection attack in apps works fundamentally like the XSS attack, but does not restrict the attack to the data of application. 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 on the approach of the attack. Other than the traditional data sharing channels like Wi-Fi and Bluetooth, smartphones and smart applications host data channels like 2D barcodes scanning, RFID tags etc. Since these Data Channels are quite prevalent in CPS, ICT, these are convenient for the access of info and data for the users, attackers use these channels quite often to inject the code and could compromise the privacy and security of the smart cities' citizens. 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 through display of 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 (Xiao et al., 2015).

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. If the functions and permissions of the vulnerable app are limited, the app could inject the code into more privileged app thus escalating the access to more vulnerable information of the victim. For example, in sustainable smart cities, in health care applications run on ML/DL which uses enormous data for the betterment of the application, the data like patients eye images could be stolen by a simple XSS attack and thus there is a need for security to protect the privacy of users (Nagarajan, Deverajan, Chatterjee, Alnumay, & Ghosh, 2021; Rahman, Hossain, Showail, Alrajeh, & Alhamid, 2021). 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

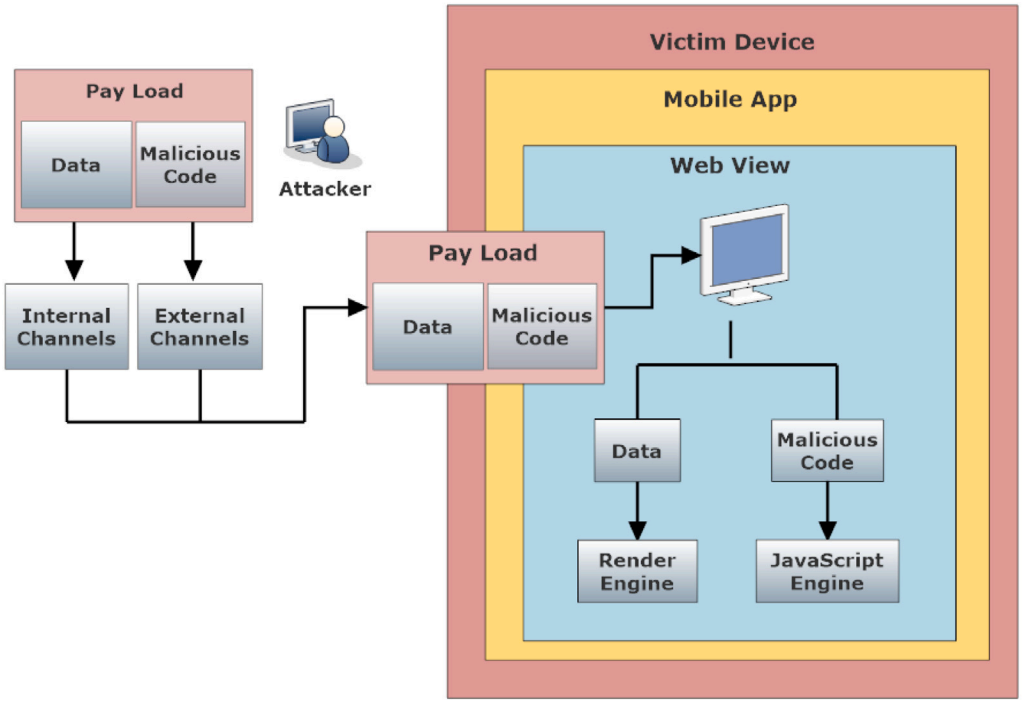

Fig. 2. Code Injection Attack on HTML-5 based Apps to compromise security of sustainable smart cities.

provider, it could be easily injected into other apps that mutually share these resources. If the malicious code is injected into the intent, it is 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., Xiao et al. (2015).

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 chose a novel approach built on deep learning application to fulfill the above need.

2.1. Why Deep Learning?

Deep Learning, a looming field of research in machine learning, endeavors to learn sophisticated portrayal of data progressively using deep neural networks. Generally derived from "Neural Network", deep refers to chain of continuous and deep layers of representation which are considered as filter for each layer to extract needed signals from that of noisy ones (Madu, Kuei, & Lee, 2017; Said & Tolba, 2021). It is used for the conception of the training and its stability and achieves enhancement on huge volumes of data. It utilizes the complicated and non linear group factors of data of training and serves for the target of constructing a pattern that leads from input to output (Chang, Ma, Chen, Gao, & Dehghani, 2021). Deep learning could be supervised, semi-supervised or unsupervised. According to LeCun, Bengio, and Hinton (2015), 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 (dar). The mentioned method does not need any exemplar data sets 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 (Bengio, Lamblin, Popovici, & Larochelle, 2006; Tang, Fonzone, Liu, & Choudhury, 2021). 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 (Le Roux & Bengio, 2010), 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 that are invariant to translation with a provincially restricted receptive field. Adding to this, in the research done by authors of Gilbert, Zhang, Lee, Zhang, and Lee (2017) and Yan, Xiao, Hu, Peng, and Jiang (2018) 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.

2.2. Other models and their limitations/ related work

There had been substantial research on the detection and numerous methods were proposed for detecting malicious code. One conventional approach followed to identify malicious code is to check whether the virus signature is found in a list of malicious virus signatures prepared

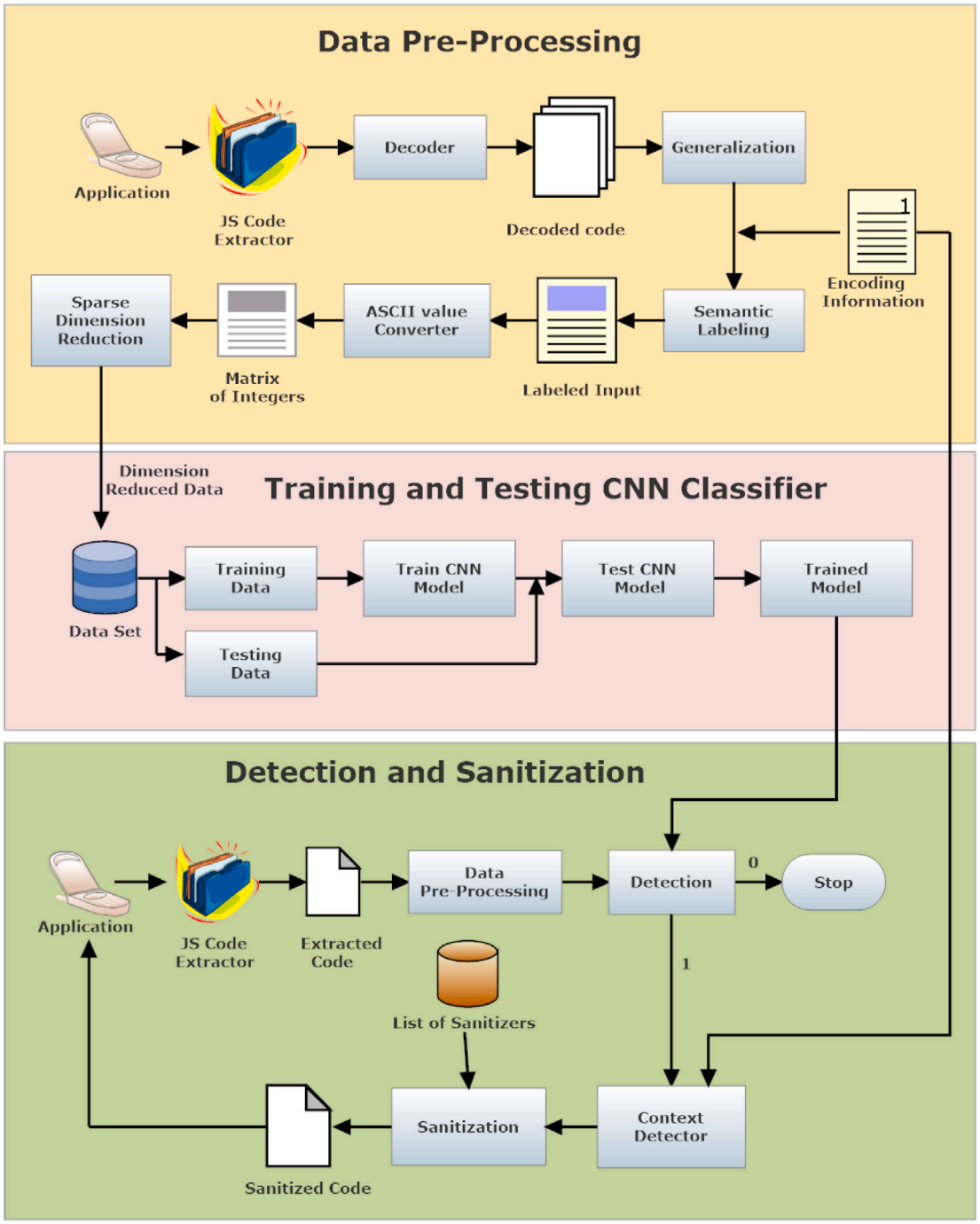

Fig. 3. Architecture of ConvXSS: The solution for XSS attacks on sustainable smart cities.

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 (Schwenk, Bikadorov, Krueger, & Rieck, 2012). 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 to determine the malicious code is based on the pre-established rules by security experts.But even this method has its fair share of drawbacks. It can only identify the familiar malicious code but fails to identify the unfamiliar malicious code. Hence, the above methodology cannot be used owing to the increase in the different types of attacks. For malicious code detection purposes, there is also a wide usage of dynamic analysis methods in which the execution of the malware takes place in a simulated environment (Kapravelos et al., 2014). 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 (Alazab, Venkatraman, Watters, & Alazab, 2013). Along with these techniques, others such as high-interaction honeypots (Kim et al., 2012), low-interaction honeypots (Alosefer & Rana, 2010), drive-by downloads (Cova, Kruegel, & Vigna, 2010; Egele, Kirda, & Kruegel, 2009), heap spraying (Ratanaworabhan, Livshits, & Zorn, 2009) 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 (Wang et al., 2016).

Some other approaches proposed by researchers after extensive research include those by Likarish, Jung, and Jo (2009) 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 Rathore, Sharma, and Park (2017) did not provide a reason for using a non-linear SVM. The authors of Mereani and Howe (2018) and Gupta, Govil, and Singh (2015) 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 Malviya, Saurav, and Gupta (2013) 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 Ghaffarian and Shahriari (2017) 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 (Selvam, 2018).

A model which uses deep learning framework was developed in Wang et al. (2016). 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 (Li, Tang, Veeravalli, & Li, 2013).

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 if detected sanitize the malicious part of the code.

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

3. 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 labeling 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. 3.

Given in Fig. 4 is the workflow of the Data-preprocessing, Training and Testing of the CNN Model. As shown in the workflow, the process is initiated through taking an application from the data pool of applications and extracting JavaScript code. From this JS code, individual code snippets/strings are taken, and this snippet is inspected for encoding. If encoded, the snippet undergoes the process of decoding as shown in the algorithm 2. If the code snippet is not encoded, then the snippet is examined for URLs. If the snippet is a URL, then the URL is replaced with a shorter URL as mentioned in the step 26 and 27 of algorithm 1. Then the decoded snippet or the replaced URL snippet or the original snippet (the snippet that neither contains encoded text or URLs) undergoes the process of labeling using the encoding information provided. Then, as shown in the workflow, this labeled input is converted into ASCII and all such labeled inputs are concatenated into the dataset. And if there are more applications, this entire procedure is done again until there are no more applications left in the data pool. Then the data undergoes the process of Sparse Dimensionality Reduction to overcome the dimensionality constraint of the dataset.

Once the data is dimensionally reduced, as shown in the second part of workflow in Fig. 4, this dataset is divided into training and testing data sets based on few conditions discussed in further sections. The training set, thus created is used to train the CNN model and after the completion of training, the CNN model is tested using test data set.

Given in Fig. 5 is the Work Flow of Code Sanitization. As seen in the figure, an application is taken and JS code is extracted and the above trained and tested CNN model is used to detect if there is any malicious code. If the result is positive then the code is taken and each code snippet is extracted. Now, the snippet's context is found and based on the context, if it is untrusted, the code snippet is sanitized using the list of sanitizers as shown in the algorithm 6. This sanitized code snippet is replaced with the original snippet and by sanitizing every code snippet based on their context, the code is sanitized. The following subsections contain the detailed discussion of each stage of the above architecture.

3.1. 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 labeled 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. Abaimov and Bianchi (2019) 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 data set . Since the dimension of the resulting data set is huge, we perform sparse dimension reduction to finalize the data set required which is going to be trained and modeled.

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

In algorithm 1, each code, from the set of JavaScript codes, is first divided into lines, and then into words. Each word is decoded and/or generalized if necessary. The resulting word is then labeled based on its context. These labeled words are further converted into ASCII values and then appended into the data set. Approaches involved in the pre-processing stage like decoding of encoded data, generalization, semantic labeling of strings etc., will be further discussed in detailed manner.

3.1.1. Decoder

As mentioned in the Cheat Sheet of XSS Filter Evasion (OWASP, 2017), 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

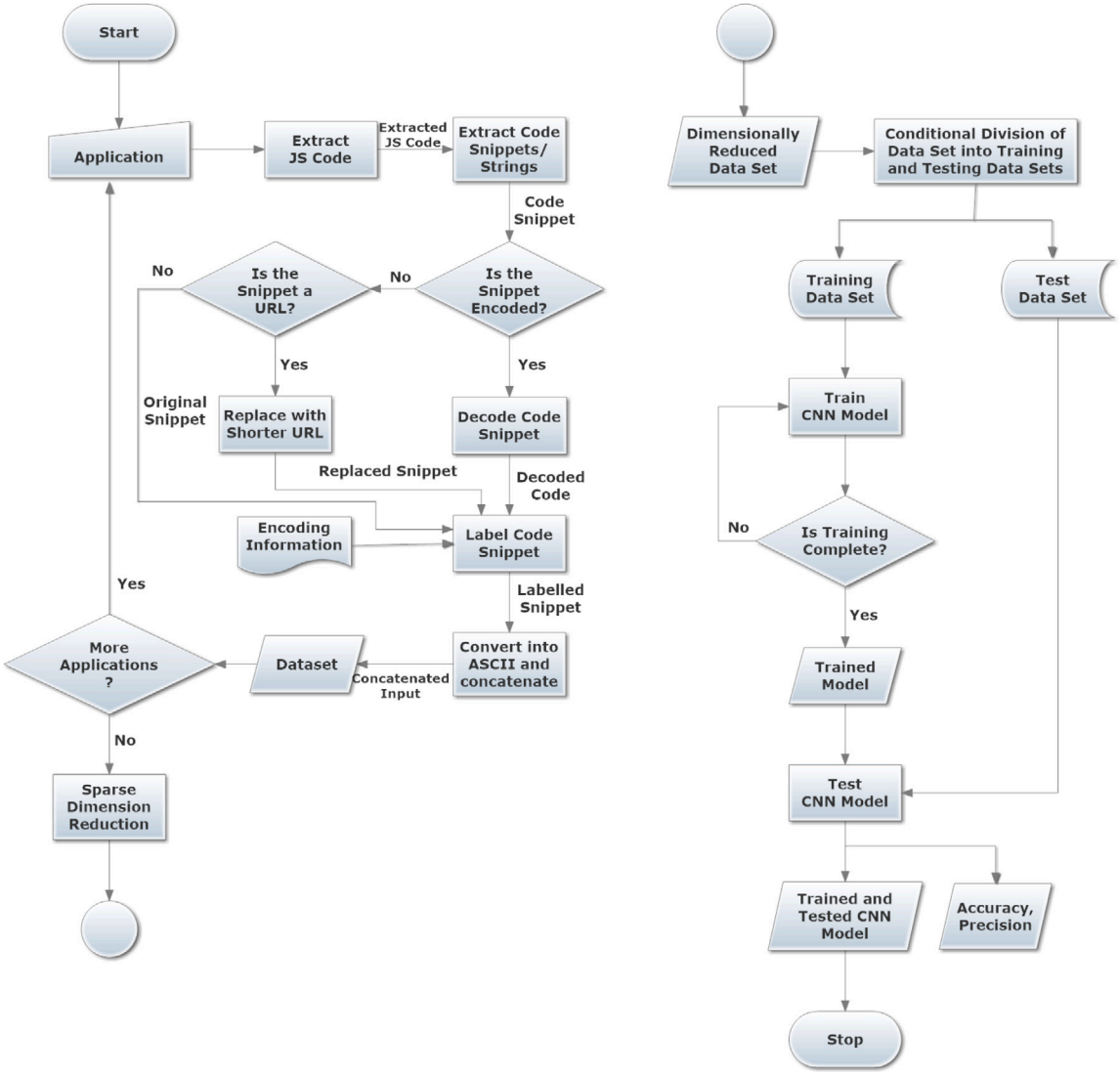

Fig. 4. Work flow of data processing, training and testing of the model.

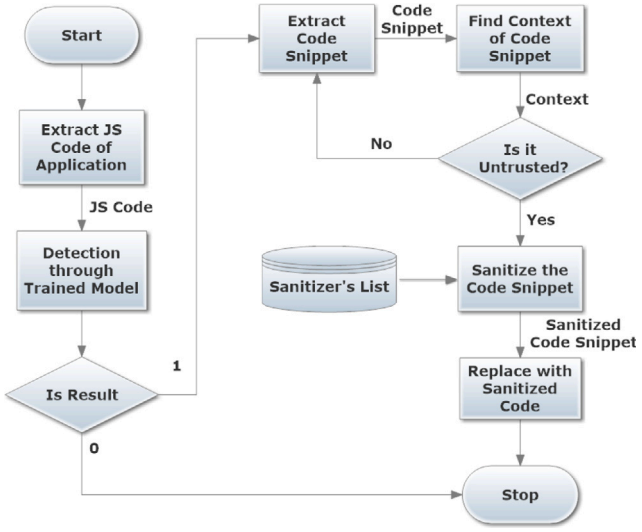

Fig. 5. Work flow of sanitization of code.

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

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="ale rt(&# 49;)">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="ale rt(& #x31;)
">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. Each character in a word is encoded as n; where n is the ASCII value of the character being encoded. 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.

3.1.2. 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.

3.1.3. 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 (Chen, Li, & Ouyang, 2018). 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. Labeling transforms the original string by appending a label to it based on its type. The number of distinct labels is very less compared to the number of distinct words. This helps the neural network 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 (Abaimov & Bianchi, 2019). This algorithm in which the respective type number based on the labeling information is appended along with the string is given below:

3.1.4. 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 data set. Then the maximum length of the row is taken and zeros are appended in the other rows to equalize the length of all rows. The algorithm for the same is mentioned below in algorithm 4:

3.1.5. Dimensionality reduction

Usually, as the applications' codes are enormous in size, the data set generated after initial pre-processing is huge, which results in the higher dimensionality of the data. So as to decrease the computation time of the training of neural network classifier, the dimensionality of the data set should reduce. Though numerous dimension reduction techniques such as Factor Analysis, ICA, PCA can be utilized for diminishing the dimension, Sparse Random Projection, a Dimension Reduction technique was used. 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 random matrix are taken from a distribution over -1, 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 d is the raw data's dimensionality (Wang et al., 2016).This random matrix is used to project the higher-dimension data set as a lower dimensional data set with similar embedding quality.

3.2. 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 (Chen, Li, & Teo, 2020). 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 (Chen, Li, & Bilal, 2018). The convolutional layer is the elementary unit of Convolutional Neural Network. In the convolutional layer, each neuron is linked to small set of input 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 sub-sampling 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 (Duan, Li, & Li, 2017).

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.

3.3. 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

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.)
begin
    Dataset ← NULL; // Declaration of Dataset Variable to add the converted data in the end.
    String A ← NULL; // Declaration of a temporary string for computations.
    String [ ] sa ← NULL; // Declaration of array of strings whose contents will be tokens of code snippet.
    int k ← 0;
    for ( SϵSet ) {
        // Application of the algorithm for each JavaScript Code in the set.
        String [ ][ MAX]B ← NULL;
        for ( stϵS ) {
            // Application of Division of JS code into lines.
            int m ← 0;
            String st1 ← StringTokenizer(st, ηη); // Taking one single line of the code at a time and storing in st1.
            int i ← 0;
            while (st1.hasMoreTokens( )) do
                // Conversion of lines into array of words for further computation.
                sa[ i] ← st1.nextToken(); // Dividing the single line of code into words and storing it in the array.
                i ← i + 1;
            end while
            int j ← 0;
            Ignore the part till one of the elements in sa is ε < script > ε // Avoid Unnecessary tags for computation since the
                initial code contains noise elements which do not add any value.
            while (sa[ j]! = η < /script > η) do
                // Conversion of strings into required format.
                if (sa[ j].substring(0, 2) == ε&#ε) then
                    A ← Decode(sa[j]); // Decoding of the Encoded Strings using Algorithm 2
                end if
                else if (sa[ j].substring(0, 7) == εhttp : //ε) then
                    A ← εhttp : //websiteε; // Replaced with shorter string to reduce data.
                end if
                else
                    A ← sa[ j];
                end if
                Label(A);// Semantic labelling of the string using encoding information as mentioned in
                    Algorithm 3.
                B[ l][ m] ← Convert(A); // Conversion of the labelled string into required input using Algorithm 4.
                m + +;
            end while
            Dataset[ k] ← B[ l]; // Entry of the converted data into data set.
            Increment k and l and continue this until all the data is converted
        }
    }
    Append_Zero(Dataset); // Appending zeroes to each row of data set for consistent length.
    Sparse_Dimension_Reduction(Dataset); // Reduction technique to reduce dimensionality.
    return Dataset ;
end
```

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 (Chaudhary & Gupta, 2017). The algorithm for the same is as mentioned in algorithm 6.

4. Implementation

4.1. 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. These data samples are taken from sources like XSS Payload Dataset (Payloadbox, 2020), Cross-site scripting dataset (Shah, 2020), XSS Filter Evasion Cheat Sheet (OWASP, 2017) and various other XSS cheat sheets like (Cozamanis, 2019), Balaji (2019), *Cross-site scripting (XSS) cheat sheet* (2021),
HTML5 security cheatsheet (2021).

4.2. 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

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; // Num is a temporary string used to append numbers and later on convert to integer.
    int[ ]Number ← 0; // Declaration of integer array to store converted numbers
    // This method is used as Encoded strings are formed by appending '&' and '#' with ASCII values of each character and ';'
    for ( chϵS ) {
        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; // converting the string to integer and adding it to the array.
            Num ← NULL;
        end if
    }
    for ( nϵNumber ) {
        // Each number is taken and converted to its corresponding character using ASCIIand 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 ← η0η;
    end if
    else if (EI[1].contains(S)) then
        A ← η1η;
    end if
    else if (EI[2].contains(S)) then
        A ← η2η; // Concatenation of a label based on the context.
    end if
    A.append(S); // Appending the existing string with the label.
    S ← A;
end
```

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 ( chϵS ) {
        // 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
```

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 take two csv files containing malicious and benign codes. Post merging, we shuffle them to make the training and test data set more random. We then extract the code and apply the respective decoders if needed. We concatenate the binary representations of each character in each line of the code, and then stack these representations. Finally, we reduce the dimensions of the dataset to avoid the curse of dimensionality.

For dividing the dataset into testing and training sub datasets, and for the evaluation of the model, we used K-fold cross validation

Algorithm 5: CNN Classifier

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

begin
// Following are the layers added to the network of convolution neural network
$convnet \leftarrow input\_data(shape = [None, SIZE, SIZE, 1], name =' input')$;
$convnet \leftarrow conv\_1d(convnet, No.of Features, FeatureVectorSize, activation =' relu')$;
$convnet \leftarrow max\_pool\_1d(convnet, FeatureVectorSize)$;
$convent \leftarrow conv\_1d(convnet, No.of Features, FeatureVectorSize, activation =' relu')$;
$convnet \leftarrow max\_pool\_1d(convnet, FeatureVectorSize)$;
$convnet \leftarrow dropout(dropout fraction)$;
$convnet \leftarrow flatten()$;
$convnet \leftarrow dense(number, activation =' relu')$;
$convnet \leftarrow dense(number, activation =' sigmoid')$;
$model \leftarrow tflearn.DNN(convnet, tensorboard\_dir =' log')$; // 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

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 (Duan, Li, & Liao, 2017). 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 our CNN implementation, the initial deep layer is more extensive than the input layer. For each subsequent layer, the number of neurons is gradually reduced, with the output layer having one neuron. 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 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 did not 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

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 ( linesϵJS ) {
        // Taking each line of the code into consideration
        for ( Sϵsl ) {
            // Considering each word in the line taken
            int l ← label_integer(S) // Taking the labeled 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
```

Table 2
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 3
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

models are as mentioned in the Table 4. 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. When we chose the number of hidden layers to be less than 8, the results of the evaluative measures are low as the neural network is not deep enough. When we chose the hidden layers to be more than 10, the neural network gets so deep that the computation time for the training and testing is huge. That is the reason behind we choosing the hidden layers to be 8 and 10.

4.3. 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 misclassification 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 Ghaffarian and Shahriari (2017), where these four cases can represent the detection models that use artificial intelligence techniques. The four cases are as follows, Fig. 6(a) represents low recall-low precision which leads to a high FP and FN, Fig. 6(b) represents high recall low precision resulting in high Fp rate and low FN rate, Fig. 6(c) represents low recall-high precision leading to high FN and low FP, or Fig. 6 (d) represents high recall-high precision which is the best case scenario, representing low FP and FN rates as given by our classifier.

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 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:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$
$$Precision = \frac{TP}{TP+FP}$$
$$Recall = \frac{TP}{TP+FN}$$
$$FPR = \frac{FP}{FP+TN}$$
$$F-Score = \frac{2*Precision*Recall}{Precision+TPR}$$
$$MisclassificationRate = \frac{FP+FN}{TP+TN+FP+FN}$$
$$AreaUndertheCurve(AUC) = \frac{1}{2}\times\left(\frac{TP}{TP+FN}+\frac{TN}{TN+FP}\right)$$

After the implementation of different 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 (see Table 2). 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. For model 4, the accuracy, precision and recall turn out to be 99.15%, 99.01%, 99.41%(see Table 3). The accuracy, precision and the recall for the other models are mentioned using a bar graph against their respective models in Fig. 7.

Table 4
Evaluative measures for models with different hidden layers and number of neurons.

Model no.	Number of hidden layers	Number of neurons[a]	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

[a]c: Conv1D + MaxPool1D, dp: Dropout, fl: Flatten, d: Dense.

Fig. 6. The four cases of model performance.

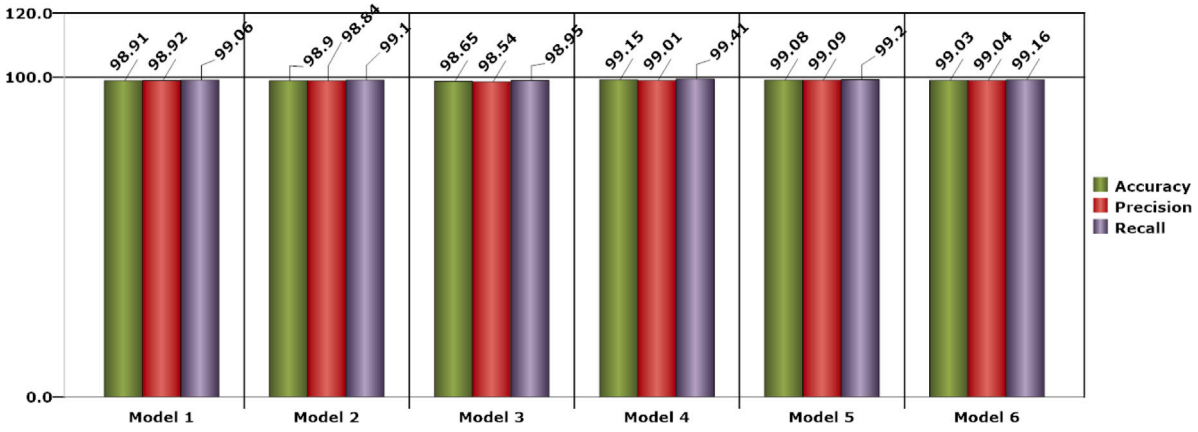

Fig. 7. Evaluation of different models.

Table 5
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)

Table 6
Comparison 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 (Abaimov & Bianchi, 2019)	95.7	99.0	91.2
Selvam, M. K. Selvam (2018)	98.54	98.65	98.40
Wang, Y., Cai, W. D., Wei, P. C. Wang et al. (2016)	94.82	94.9	94.8
DeepXSS - Fang, Y., Li, Y., Liu, L., Huang, C. Fang, Li, Liu, and Huang (2018)	98.5	99.5	97.9
Adaboost - Wang, R., Jia, X., Li, Q., Zhang, S. Wang, Jia, Li, and Zhang (2014)	94.1	93.9	93.9

4.4. 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 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 Abaimov and Bianchi (2019) and Wang et al. (2016), 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 6 below.

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.

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.

4.5. 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, BLSTM (Song, Chen, Cui, & Fu, 2020). Naïve Bayes classifiers are simple probabilistic classifiers that run on the application of Bayes' theorem with naïve independent assumptions considered between the features (Zhou, Li, Xiao, Zhou, & Li, 2016). 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

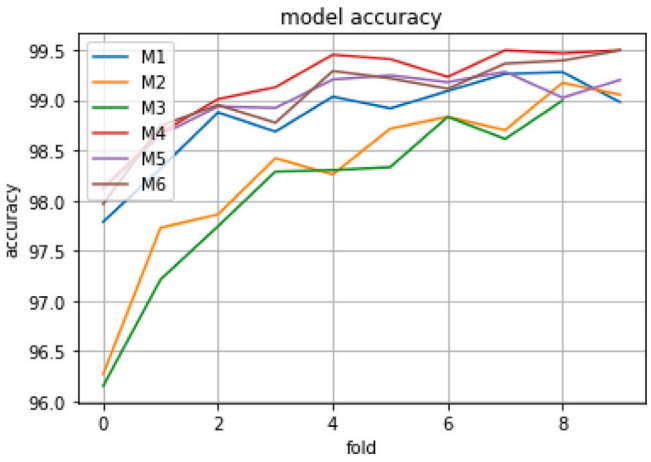

Fig. 8. Accuracy vs fold value.

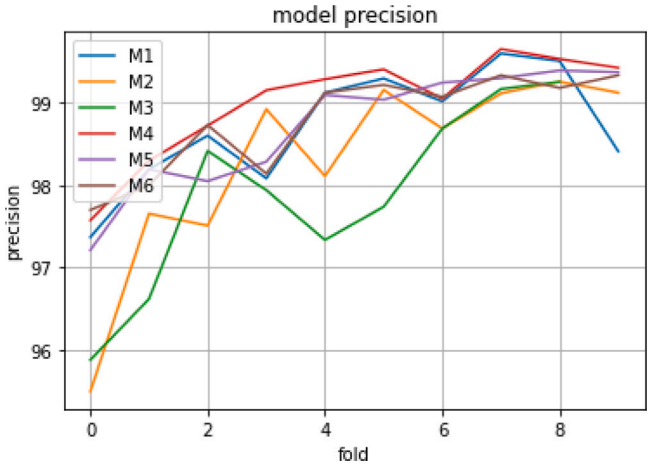

Fig. 9. Precision vs fold value.

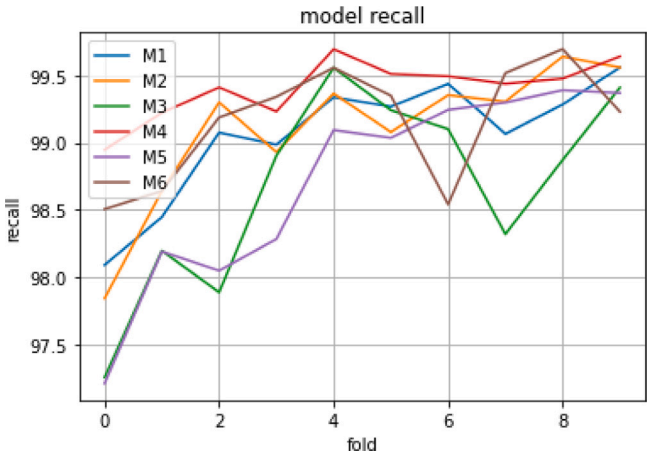

Fig. 10. Recall vs fold value.

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

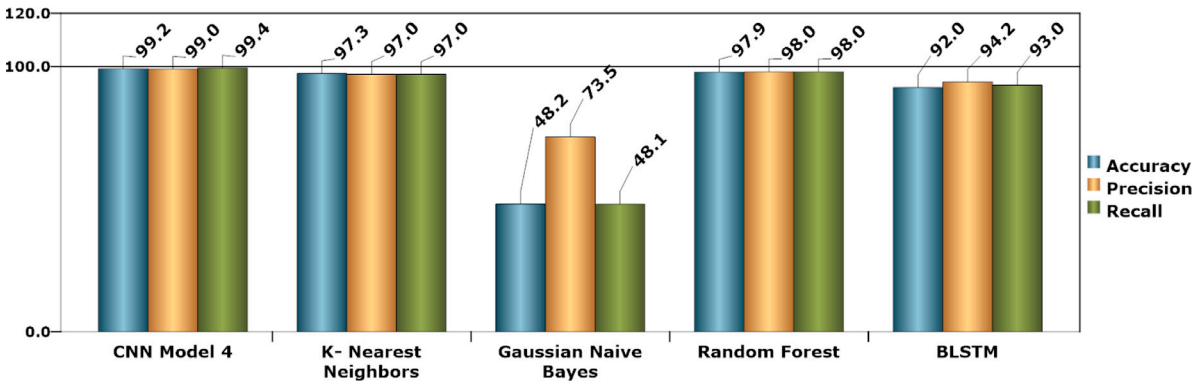

Fig. 11. Comparison with other models.

Table 7
Dataset subdivisions.

DATA			
Name	Benign	Malicious	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

various sub samples of data-set for prediction. The tuning parameters for the decision tree are the tree depth and number of trees (Chen et al., 2016). (SVM, a supervised learning model is associated with learning algorithms for analyzing data and recognition of patterns). We have also trained and tested another deep learning classifier, BLSTM for the purpose of comparison. 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 graph, CNN provides slightly higher accuracy than other models. All the CNN models performed better than KNN, Random Forest, Gaussian Naïve Bayes models and BLSTM out of which Model 4 has better performance among all the CNN models.

4.6. 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 desired 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 7 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 feeding it into our deep learning model to keep the process memory efficient and batch-wise operable. The convolutional

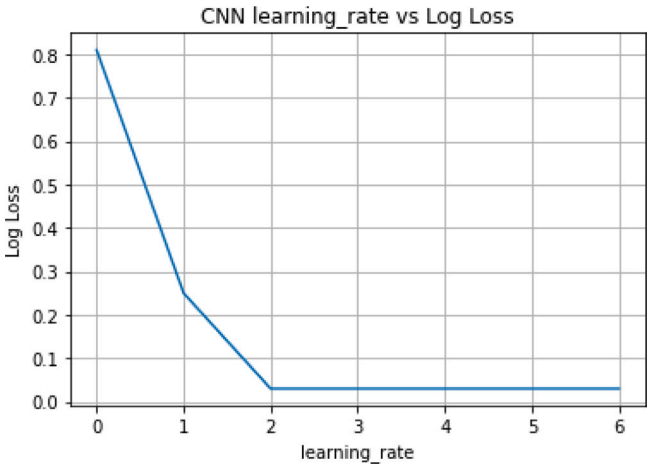

Fig. 12. Learning rate vs log loss.

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 8. The proposed model achieved an accuracy rate of 99.42%, recall rate of about 99.35% and precision value of about 99.81%. Figs. 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 Figs. 13 and 14.

Fig. 15 shows the result of the confusion matrix with a complete statistical view. This figure is the pictorial representation of Table 5, along with the statistics for each category — TP, FP, TN, FN. 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

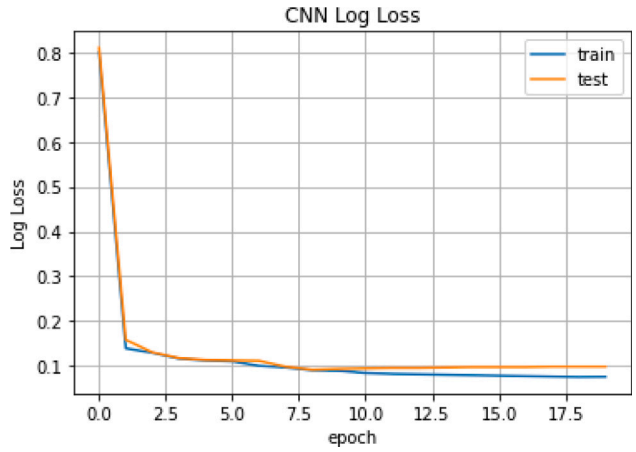

Fig. 13. Cross entropy vs epochs.

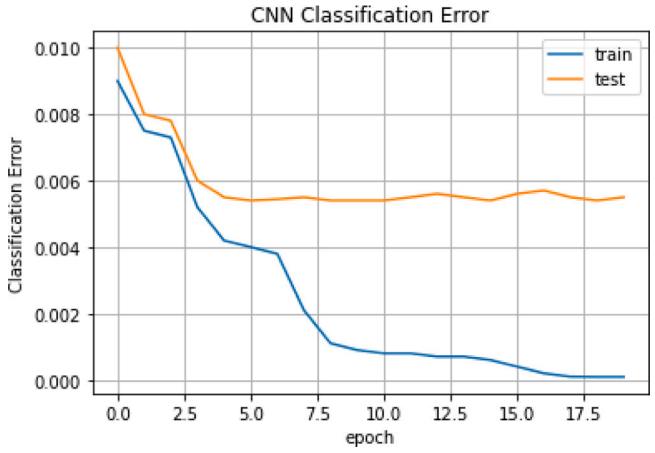

Fig. 14. Classification error vs epochs.

Table 8
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 (Song et al., 2020)	92.01	94.23	93.03
CNN-LSTM (Kadhim & Gaata, 2021)	99.30	99.90	99.10
Proposed: ConvXSS	**99.42**	**99.81**	**99.35**

and high recall simultaneously. For the purpose of analysis of latency, we have noted down the computation time of the above procedure. The time taken for 20 epochs for above procedure is 455.12 s for total samples of 1,05,470. So, each code snippet takes 0.216 ms on average to get classified by ConvXSS. Considering the average number of lines of code be 10000, the latency will be around 2.16 s .

4.7. 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 9. Table 9 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.

Table 9
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

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.

4.8. Limitations of this model

This paper discusses the XSS attacks in sustainable smart cities and proposes a CNN Based Model to detect the malicious code and also the method of sanitization to sanitize the code given. Though the results of this paper have been good it has its fair share of limitations. They have been listed under:

(1) In the Preprocessing of Data, during the phase of Semantic Labeling, we use Encoding Information for the purpose of labeling of data. This Encoding Information is limited and could be further elaborated based on the different languages used for the development of applications.
(2) In the same way, for the purpose of sanitization, more number of sanitizers could be added as the number of variety of attacks have been increased.
(3) Even though we claimed that this method provides faster results, we could still improve the computation power by devising more efficient algorithms

5. Conclusion

One of the important factors in sustainable 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

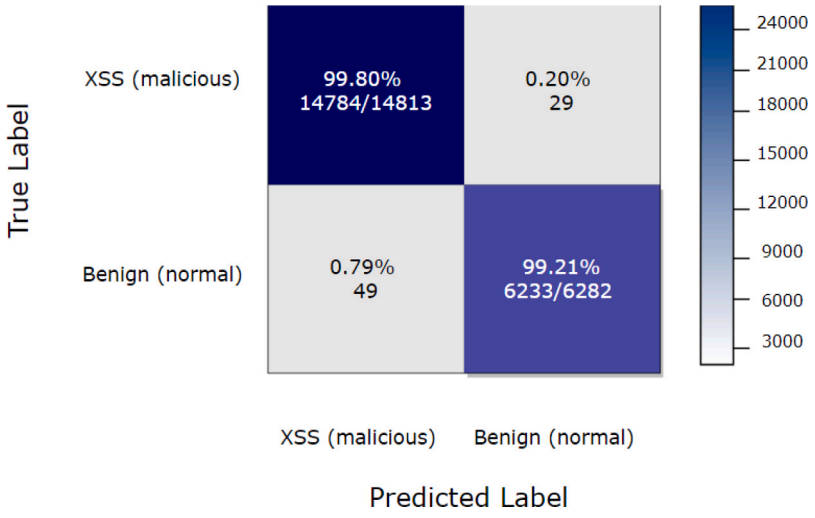

Fig. 15. Confusion matrix on the test dataset.

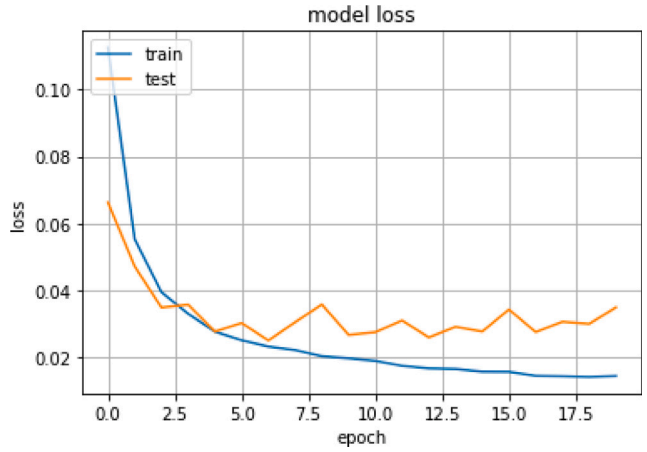

Fig. 16. Loss versus number of epochs.

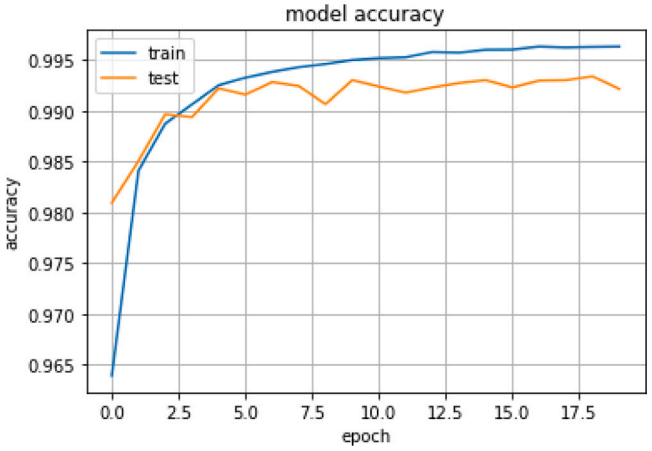

Fig. 17. Accuracy versus number of epochs.

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 initially discussed briefly about the need of rise of sustainable smart cities and the definition of security and privacy in the context of sustainable smart cities. Then, we studied one of the important attacks of sustainable smart cities, XSS attacks and the potential risk 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 and discussed few examples of possible attacks on sustainable smart cities. 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 its 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 (Song et al., 2020) and hybrids of CNN-LSTMs (Kadhim & Gaata, 2021) as mentioned in Table 8. After the evaluation of the model, we have also 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 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) This method could be embedded in the operating system of the smart phone and other smart applications such that whenever a person opens or downloads a compromised website/app, the OS can immediately block or delete or sanitize the site or app securing the appliance.
(4) 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 sustainable 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.

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.

Code availability

Code will be available based on the formal request to the corresponding author.

Acknowledgment

The work is funded by external research grant from Data Security Council of India.

Funding

This work is economically supported by Data Security Council of India.

References

Darktrace, https://www.darktrace.com/en/resources/.
Abaimov, S., & Bianchi, G. (2019). CODDLE: Code-injection Detection with deep learning. *IEEE Access*, *7*, 128617–128627.
Alazab, M., Venkatraman, S., Watters, P., & Alazab, M. (2013). Information security governance: the art of detecting hidden malware. In *IT security governance innovations: theory and research* (pp. 293–315). IGI Global.
Alosefer, Y., & Rana, O. (2010). Honeyware: A web-based low interaction client honeypot. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops* (pp. 410–417).
Balaji, N. (2019). Top 500 most important XSS cheat sheet for web application pentesting. *GBHackers On Security*, https://gbhackers.com/top-500-important-xss-cheat-sheet/.
Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006). Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems*, *19*, 153–160.
Braun, T., Fung, B. C., Iqbal, F., & Shah, B. (2018). Security and privacy challenges in smart cities. *Sustainable Cities and Society*, *39*, 499–507.
Chang, Q., Ma, X., Chen, M., Gao, X., & Dehghani, M. (2021). A deep learning based secured energy management framework within a smart island. *Sustainable Cities and Society*, *70*, 102938.
Chaudhary, P., & Gupta, B. (2017). A novel framework to alleviate dissemination of XSS worms in online social network (OSN) using view segregation. *Neural Network World*, *27*(1), 5.
Chen, J., Li, K., Bilal, K., Li, K., Philip, S. Y., et al. (2018). A bi-layered parallel training architecture for large-scale convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, *30*(5), 965–976.
Chen, C., Li, K., Ouyang, A., & Li, K. (2018). Flinkcl: An Opencl-based in-memory computing architecture on heterogeneous cpu-gpu clusters for big data. *IEEE Transactions on Computers*, *67*(12), 1765–1779.
Chen, J., Li, K., Tang, Z., Bilal, K., Yu, S., Weng, C., et al. (2016). A parallel random forest algorithm for big data in a spark cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, *28*(4), 919–933.
Chen, C., Li, K., Teo, S. G., Zou, X., Li, K., & Zeng, Z. (2020). Citywide traffic flow prediction based on multiple gated spatio-temporal convolutional neural networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, *14*(4), 1–23.
Chen, D., Wawrzynski, P., & Lv, Z. (2020). Cyber security in smart cities: A review of deep learning-based applications and case studies. *Sustainable Cities and Society*, 102655.
Cova, M., Kruegel, C., & Vigna, G. (2010). Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *WWW '10, Proceedings of the 19th International Conference on World Wide Web* (pp. 281–290). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/1772690.1772720.
Cozamanis, A. (2019). XSS vectors cheat sheet, Gist, https://gist.github.com/kurobeats/9a613c9ab68914312cbb415134795b45.
Cross-site scripting (XSS) cheat sheet - 2021 edition: Web security academy, Cross-Site Scripting (XSS) Cheat Sheet - 2021 Edition | Web Security Academy, https://portswigger.net/web-security/cross-site-scripting/cheat-sheet.
Duan, M., Li, K., & Li, K. (2017). An ensemble CNN2ELM for age estimation. *IEEE Transactions on Information Forensics and Security*, *13*(3), 758–772.
Duan, M., Li, K., Liao, X., & Li, K. (2017). A parallel multiclassification algorithm for big data using an extreme learning machine. *IEEE Transactions on Neural Networks and Learning Systems*, *29*(6), 2337–2351.
Egele, M., Kirda, E., & Kruegel, C. (2009). Mitigating drive-by download attacks: Challenges and open problems. In *iNetSec open research problems in network security, Zurich, Switzerland*.
Elmaghraby, A. S., & Losavio, M. M. (2014). Cyber security challenges in smart cities: Safety, security and privacy. *Journal of Advanced Research*, *5*(4), 491–497.
Elnour, M., Meskin, N., Khan, K., & Jain, R. (2021). Application of data-driven attack detection framework for secure operation in smart buildings. *Sustainable Cities and Society*, *69*, 102816.
Fang, Y., Li, Y., Liu, L., & Huang, C. (2018). DeepXSS: Cross site scripting detection based on deep learning. In *Proceedings of the 2018 international conference on computing and artificial intelligence* (pp. 47–51).
Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, *50*(4), 1–36.
Gilbert, A. C., Zhang, Y., Lee, K., Zhang, Y., & Lee, H. (2017). Towards understanding the invertibility of convolutional neural networks. arXiv preprint arXiv:1705.08664.
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)* (pp. 162–167). IEEE.
(2021). HTML5 security cheatsheetwhat your browser does when you look away...HTML5 security cheatsheet, http://html5sec.org/.
Jin, X., Hu, X., Ying, K., Du, W., Yin, H., & Peri, G. N. (2014). 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* (pp. 66–77).
Kadhim, R., & Gaata, M. (2021). A hybrid of CNN and LSTM methods for securing web application against cross-site scripting attack. *Indonesian Journal of Electrical Engineering and Computer Science*, *21*, 1022. http://dx.doi.org/10.11591/ijeecs.v21.i2.pp1022-1029.
Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., & Paxson, V. (2014). Hulk: Eliciting malicious behavior in browser extensions. In *23rd {USENIX} security symposium ({USENIX} Security 14)* (pp. 641–654).
Khatoun, R., & Zeadally, S. (2017). Cybersecurity and privacy solutions in smart cities. *IEEE Communications Magazine*, *55*(3), 51–59.
Kim, H.-G., Kim, D., Cho, S.-J., Park, M., & Park, M. (2012). Efficient detection of malicious web pages using high-interaction client honeypots. *Journal of Information Science & Engineering*, *28*(5).
Laufs, J., Borrion, H., & Bradford, B. (2020). Security and the smart city: A systematic review. *Sustainable Cities and Society*, *55*, 102023.
Le Roux, N., & Bengio, Y. (2010). Deep belief networks are compact universal approximators. *Neural Computation*, *22*(8), 2192–2207.
LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.
Lever, K. E., & Kifayat, K. (2020). Identifying and mitigating security risks for secure and robust NGI networks. *Sustainable Cities and Society*, *59*, 102098.
Li, K., Tang, X., Veeravalli, B., & Li, K. (2013). Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems. *IEEE Transactions on Computers*, *64*(1), 191–204.
Likarish, P., Jung, E., & Jo, I. (2009). Obfuscated malicious javascript detection using classification techniques. In *2009 4th international conference on malicious and unwanted software (MALWARE)* (pp. 47–54). IEEE.
Liu, Q., Li, W., Chen, Z., & Hua, B. (2021). Deep metric learning for image retrieval in smart city development. *Sustainable Cities and Society*, 103067.
Liu, J., Li, K., Zhu, D., Han, J., & Li, K. (2016). Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, *16*(2), 1–25.
Madu, C. N., Kuei, C.-h., & Lee, P. (2017). Urban sustainability management: A deep learning perspective. *Sustainable Cities and Society*, *30*, 1–17.

Malviya, V. K., Saurav, S., & Gupta, A. (2013). On security issues in web applications through cross site scripting (XSS). *vol. 1*, In *2013 20th asia-pacific software engineering conference (APSEC)* (pp. 583–588). IEEE.

Mantha, B., de Soto, B. G., & Karri, R. (2021). Cyber security threat modeling in the AEC industry: An example for the commissioning of the built environment. *Sustainable Cities and Society*, *66*, 102682.

Mereani, F. A., & Howe, J. M. (2018). Detecting cross-site scripting attacks using machine learning. In *International conference on advanced machine learning technologies and applications* (pp. 200–210). Springer.

Mishra, A. K., Puthal, D., & Tripathy, A. K. (2021). GraphCrypto: Next Generation data security approach towards sustainable smart city building. *Sustainable Cities and Society*, 103056.

Mohammadpourfard, M., Khalili, A., Genc, I., & Konstantinou, C. (2021). Cyber-resilient smart cities: Detection of malicious attacks in smart grids. *Sustainable Cities and Society*, 103116.

Nagarajan, S. M., Deverajan, G. G., Chatterjee, P., Alnumay, W., & Ghosh, U. (2021). Effective task scheduling algorithm with deep learning for internet of health things (IoHT) in sustainable smart cities. *Sustainable Cities and Society*, *71*, 102945.

OWASP, X. (2017). Filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

Payloadbox (2020). Payloadbox/XSS-payload-list: Cross Site scripting (XSS) vulnerability payload list, GitHub. https://github.com/ismailtasdelen/xss-payload-list.

Rahman, M. A., Asyhari, A. T., Leong, L., Satrya, G., Tao, M. H., & Zolkipli, M. (2020). Scalable machine learning-based intrusion detection system for IoT-enabled smart cities. *Sustainable Cities and Society*, *61*, 102324.

Rahman, M. A., Hossain, M. S., Showail, A. J., Alrajeh, N. A., & Alhamid, M. F. (2021). A secure, private, and explainable IoHT framework to support sustainable health monitoring in a smart city. *Sustainable Cities and Society*, 103083.

Ratanaworabhan, P., Livshits, B., & Zorn, B. (2009). NOZZLE: A defense against heap-spraying code injection attacks. In *USENIX security symposium*.

Rathore, S., Sharma, P. K., & Park, J. H. (2017). XSSClassifier: An Efficient XSS attack detection approach based on machine learning classifier on SNSs. *Journal of Information Processing Systems*, *13*(4).

Said, O., & Tolba, A. (2021). Accurate performance prediction of IoT communication systems for smart cities: An efficient deep learning based solution. *Sustainable Cities and Society*, *69*, 102830.

Schwenk, G., Bikadorov, A., Krueger, T., & Rieck, K. (2012). Autonomous learning for detection of JavaScript attacks: Vision or reality?. In *AISec '12*, *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence* (pp. 93–104). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/2381896.2381911.

Selvam, M. K. (2018). *Classification of Malicious Web Code Using Deep Learning*. (Ph.D. dissertation), Dublin, National College of Ireland.

Shah, S. S. H. (2020). Cross site scripting XSS dataset for deep learning, Kaggle. https://www.kaggle.com/syedsaqlainhussain/cross-site-scripting-xss-dataset-for-deep-learning.

Silva, B. N., Khan, M., & Han, K. (2018). Towards sustainable smart cities: A review of trends, architectures, components, and open challenges in smart cities. *Sustainable Cities and Society*, *38*, 697–713.

Song, X., Chen, C., Cui, B., & Fu, J. (2020). Malicious JavaScript detection based on bidirectional LSTM model. *Applied Sciences*, *10*(10), http://dx.doi.org/10.3390/app10103440, https://www.mdpi.com/2076-3417/10/10/3440.

van der Stock, A., Glas, B., Smithline, N., & Gigler, T. (2017). OWASP Top 10-2017 the ten most critical web application security risks. *Creative Commons*.

Tang, T., Fonzone, A., Liu, R., & Choudhury, C. (2021). Multi-stage deep learning approaches to predict boarding behaviour of bus passengers. *Sustainable Cities and Society*, 103111.

Wang, Y., Cai, W.-d., & Wei, P.-c. (2016). A deep learning approach for detecting malicious JavaScript code. *Security and Communication Networks*, *9*(11), 1520–1534.

Wang, R., Jia, X., Li, Q., & Zhang, S. (2014). Machine learning based cross-site scripting detection in online social network. In *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).

Xiao, X., Yan, R., Ye, R., Li, Q., Peng, S., & Jiang, Y. (2015). Detection and prevention of code injection attacks on HTML5-based apps. In *2015 third international conference on advanced cloud and big data* (pp. 254–261). IEEE.

Yan, R., Xiao, X., Hu, G., Peng, S., & Jiang, Y. (2018). New deep learning method to detect code injection attacks on hybrid applications. *Journal of Systems and Software*, *137*, 67–77.

Zhou, X., Li, K., Xiao, G., Zhou, Y., & Li, K. (2016). Top k favorite probabilistic products queries. *IEEE Transactions on Knowledge and Data Engineering*, *28*(10), 2808–2821.