DeeperXSS: An Exploration of the DeepXSS Approach

Matthew Bush, Dawson Brown
Ryerson University, Toronto, Canada
matthew.bush@ryerson.ca,dawson.brown@ryerson.ca

ABSTRACT

Cross-site scripting (XSS) attacks pose a major threat to current Web applications. XSS very regularly ranks among the top ten most common attacks and vulnerabilities on the Web. As such, the successful detection and prevention of such attacks remain an active and important area of research. In this paper, we set out to replicate, critique, and expand on the DeepXSS approach proposed by Fang et al. DeepXSS gives a lackluster presentation of an LSTM classifier that boasts very high precision and recall rates. In order to better understand the application of deep learning techniques to XSS classification problems, we dissect their approach, try to recreate their impressive results and contribute some modifications, extensions, and alternative architectures. With this, we hope to justify and critique some of their choices and provide some context as to what works and what doesn't in deep learning XSS classifiers.

KEYWORDS

Cross-site Scripting, Deep Learning, LSTM, Word2Vec.

1 INTRODUCTION

Cross-site scripting (XSS) attacks persist as a major issue for Web applications despite their root causes being well understood. XSS is a vulnerability by which attackers can inject malicious code into an HTML page which is then received and executed by a victim's browser. With ever more sensitive data being transacted on the Web, the personal and industrial consequences of XSS are ever increasing [2]. XSS can be split into three broad categories: persistent, reflected, and DOM [7]. Persistent XSS is when an attacker manages to save malicious code on a server to be served later to victims; reflected occurs when a server echos back a portion of a request–if an attacker can get unsanitized malicious code to be echoed to a victim's machine (by having the victim click a malicious link) the code will be executed in the browser; And lastly DOM is when the attack is executed by modifying the DOM of the victim's browser [7]. The Open Web Application Security Project (OWASP) consistently ranks XSS in the top ten vulnerabilities on the Web, including in their most recent 2021 survey (however XSS was lumped into the generic 'injection' category in 2021 instead of having its own category) [11].

For this reason, there exists a large body of work aimed at automating the detection and prevention of XSS attacks; this includes approaches using Machine Learning techniques to detect both attacks and vulnerabilities. DeepXSS is an LSTM classifier that is meant to detect XSS payloads in reflected XSS attacks; it was designed by Fang *et. al* and is purported to have very high precision and recall rates [4]. We want to more clearly outline and comment on the strengths and weaknesses of their architecture as well as replicate their purported results.[4].

The primary motivation for this work is to verify Fang *et. al*'s DeepXSS method of detecting cross-site scripting (XSS) attack payloads [4]. Given the significance of XSS attacks on the Web, there is a need to scrutinize detection and prevention techniques to better understand what works and what doesn't. We found that DeepXSS lacked detail and failed to address key questions related to their work and so required a deeper investigation. Given DeepXSS's promising results, it would be very useful to address the lack of detail, and attempt to replicate the methods used by the authors. In verifying these results we could further our understanding of why DeepXSS was so effective (or why it was not as effective as it seemed) and apply those lessons to future research. Our main contributions are as follows:

1. A custom built URL decoder
2. An extension of DeepXSS's tokenization technique
3. A successful recreation of DeepXSS's Word2Vec application
4. A successful recreation of DeepXSS's LSTM classifier results
5. Seven additional LSTM models that vary in their input, embedding and/or output to compare and contrast some of the choices made in DeepXSS
6. A public Github repository with all our code and data [1]

The rest of this paper is organized as follows: first we give a brief overview of DeepXSS highlighting its limitations in terms of reproducibility, then we outline our DeeperXSS approach where we recreate (with some modifications) the DeepXSS architecture, then we evaluate our models using 10-fold cross validation, we then go over some related work that factored into our decision making when trying to recreate this work, lastly we give a brief discussion and conclusion.

2 BACKGROUND: DEEPXSS

Fang *et al.* claim to make 3 contributions in DeepXSS:

1. a decoder to undo URL obfuscation
2. a Word2Vec model to embed token strings
3. an LSTM classifier to detect XSS

The decoder is a function that will recursively try to decode 'all' possible URL encoding techniques [4]. Precisely what is meant by 'all' encodings is unspecified nor is any source code provided. They however claim to be able to restore arbitrarily obfuscated data [4].

After decoding, they 'generalize' the resulting URLs. They map all domain names to simply 'website', all numbers to '1' and all string parameters to "param_string". They also remove blank and control characters [4]. They give little justification for this and this

Supervised by Prof. Manar Alalfi.

Program Analysis for cybersecurity(CP8320),
2021.

[1] https://github.com/dawson-brown/DeeperXSS

Classification	Example
Start Label	<script>, <body>, <img
End Label	< /script>, < /body>
Windows Event	onerror=, onload=, onblur=, oncut=
Function Name	alert(, String.fromCharCode(
Script URL	javascript:, vbscript:
Other	>,), #

Table 1: DeepXSS Tokens.

step does not seem to affect feature extraction (tokenization) as parameters and numbers aren't tokenized (see table 1).

The tokenization is the first step in feature extraction. They turn each URL into a sequence of tokens according to custom regexes. They do not provide those regexes; they merely provide a table summarizing the types of tokens being extracted (table 1) [4]. The tokens chosen appear to target only the types of strings that would be present in XSS payloads and many benign URLs contain no tokens. It appears that the classifier doesn't come to learn what benign URLs look like as the vast majority contain no features.

Once the URLs have been tokenized, they use the continuous bag of words (CBOW) model to perform word embedding on the token strings [4]. They do this to extract the semantics of the tokens and provide more features to the LSTM classifier [4]. They do not specify the numbder of dimensions their word vectors have nor the number of epochs used while training the CBOW model; CBOW is, of course, sensitive to both [9].

CBOW is useful for extracting semantic information from words. A CBOW model is a neural net language model trained by iterating through all words (or tokens in our case) in a given corpus and attempting to predict each word based on those around it. After training the model, the embedding layer is able to produce multidimensional vectors that represent semantic information about input words [9]. Sequences of these word embeddings are the input for the LSTM in DeepXSS [4].

Once the word vectors are trained, the LSTM can be trained by feeding it sequences of word vectors representing URLs [4]. LSTM-based models are extremely useful for processing sequential data. A particular advantage of the LSTM approach is that its memory cell allows it to learn long-term patterns and helps to reduce the vanishing and exploding gradients problems [8].

3 OUR APPROACH: DEEPERXSS

In this section we outline our approach for recreating the DeepXSS architecture, including difficulties we had in reproducibility, limitations with DeepXSS that needed to be addressed, some creative liberties we took, and a few alternative machine learning architectures that proved interesting.

3.1 Data Preprocessing

3.1.1 Decoding. We built a custom recursive URL decoder. This decoder performs a depth first search of 5 different decodings. At each level, the decoder tries to decode the URL with all decodings. For each decoding that is successful, the decoder will recursively try to further decode the string that resulted from the decoding; see algorithm 1 for a pseudocode implementation. The first string encountered that none of the decoders can decode is returned as the decoded string. The supported encodings are: URL unicode encoding (this includes characters of the form %uxxxx and \uxxxx),URL encoding (that is characters of the form %xx), HTML character references (that is characters of the form &xx;), hex encoding, and base64 encoding.

Algorithm 1 works as follows: on line 4 all the decoder functions are called and passed the URL to be decoded. These functions all return a Tuple of the form (Boolean, String) where the Boolean value indicates if the decoding was successful while the String is the decoded string (the String remains unchanged if the decoding fails). Next, on line 8, the result tuples of all the attempted decoders are looped over. If it's found that a decoder was successful (line 10) then recursively call the decoder on the resulting string (line 11). If the recursive decoding succeeds then return the result of the recursive call (13). If the decoder gets to line 17, than that means one of two things happened: either none of the decoders succeeded, in which case the string must be fully decoded and so we go to line 20; if on the other hand some of the decoders succeeded, then if the algorithm gets to line 17 it must have been the case that none of the recursive calls succeeded (meaning line 13 was never reached) which means the decoder was unable to decode the string and we return on line 18.

Algorithm 1 Recursive Decoder

```
1:  function DECODE(url)
2:      decoders ← [url(), unicode(), html(), hex(), base64()]
3:      dec_results ← []
4:      for decoder in decoders do
5:          dec_results.append(decoder(url))
6:      end for
7:      some_decoded ← False
8:      for decode_success, decode_str in dec_results do
9:          some_decoded ← decode_success or some_decoded
10:         if decode_success then
11:             (next_decode_success, next_decode_str) = DECODE(decode_str)
12:             if next_decode_success then
13:                 return (True, next_decode_str)
14:             end if
15:         end if
16:     end for
17:     if some_decoded then
18:         return (False, url)
19:     else
20:         return (True, url)
21:     end if
22: end function
```

To clarify with an example, consider the URL:

`http://example.com/706174682F746F2F66696C653F783D26616D703B6C743B7363726970742616D703B67743B253230616C65727428253230312532302925323026616D703B6C743B2F7363726970742616D703B67743B`.

Passing this through the decoder, the Hex decoder will succeed and produce the string:

`http://example.com/path/to/file?x=&amp;lt;script&amp;gt;%20alert(%201%20)%20&amp;lt;/script&amp;gt;`

then the URL decoder will succeed giving the string:

`http://example.com/path/to/file?x=&amp;lt;script&amp;gt;alert(1)&amp;lt;/script&amp;gt;`

then the HTML decoder will succeed and give:

`http://example.com/path/to/file?x=&lt;script&gt;alert(1)&lt;/script&gt;`

and finally the HTML decoder will succeed again and give:

`http://example.com/path/to/file?x=<script>alert(1)</script>`.

No decoders will succeed here making this the final form.

3.1.2 Tokenization. As seen in table 1, DeepXSS defined six different token types. We expanded on this set of tokens and ended up with a total of 14 token types. The reason we expanded on this token set is because many URLs contained zero tokens. For example `http://www.wittebeer.be/?oid=911&pid=8056` or `http://www.facebook.com/Euphnet?sk=wall`, both of which are in the DMOZ directory, do not contain any of their token types. Using more token types common to normal URLs and increasing the feature richness as a way to have benign samples not be empty was something we encountered in the literature [10][16]. But put simply, we didn't want our classifier to come to learn that benign URLs are largely empty; we wanted it to learn the structure of both benign and malicious URLs. As such, we expanded their table to include 8 more tokens as shown in table 2.

An integer argument token is any integer value that follows a function name token and is enclosed in brackets. Similarly, a string argument token is taken to be any argument that isn't an integer; hence *String.fromCharCode(65)* being an example of a string argument. We figured that distinguishing futher between arguments would be superfluous, and by inspection these were the most common arguments (including functions which returned a string or integer). The integer constant token was introduced because many URLs contained non-descript integers outside of function calls and URL arguments. The two assignment tokens (left-hand side and right-hand side) were introduced so that URL arguments would not be lost (notice that the two benign examples from Dmoz above include arguments that are skipped by DeepXSS). Similarly, path tokens were intoduced so that the path wasn't entirely skipped. And finally we introduced a generic identifier token. Assignment LHS, Path, Script URL, Function Name, and Windows Events all begin with an identifier but all have additional characters to further distinguish them (for example Windows Events always start with 'on' and end with '=', paths end with '/' or '?', etc). Any identifier that is found that cannot be further categorized is included as a generic Identifier token.

Classification	Example
Integer Argument	(543), (1), (2004)
Integer Constant	1, 2, 5432, 54
String Argument	("Hello"), (String.fromCharCode(65))
Assignment LHS	x=, variable=
Assignment RHS	=x, =654, =value
Path	path/ t56543-trer-yt43/
Identifier	iden, value, hello, goodbye

Table 2: Extended DeepXSS Tokens.

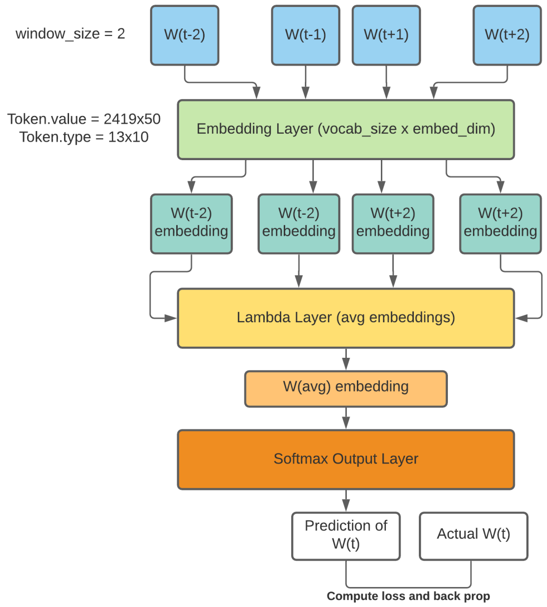
3.1.3 Generalization. In keeping with DeepXSS, we generalized many parts of the URL. In the original paper all the domains became simply 'website'. Having every URL begin with 'http://website' communicates no information and so we simply omitted the domain. We mapped all integer arguments to '(1)' and all string arguments to '("str_arg")'. All integer constants were mapped to '1'. The left-hand side of assignments were all generalized to 'assign<num>=' where '<num>' starts from 0 and increments for each encountered left-hand side. This was done because most identifiers in URLs proved to be unique, and so not useful in classification (especially when using CBOW) with the exception of Function Names, script URLs, and Windows Events which are reserved words and so can't be unique across many URLs. The right-hand sides are generalized to 'val<num>' where '<num>' again increments with each right-hand side encountered. Similarly, all path tokens are generalized to 'path<num>/' and finally all identifier tokens were mapped to 'ident<num>'.

To clarify the tokenization and generalization process with an example, consider `http://website/search.php?uid=ws848d8024088c327.36898031&src=&term=<script>alert(123)</script>&args=qs=06o`. The following sequence of (token type, token value) pairs would be extracted going left to right:

Listing 1: Example tokenization

```
(Path, `path0/') for search.php?
(Assignment LHS, `assign0=') for uid=
(Assignment RHS, `val0') for =ws848d8024088c327
(Integer Constant, 1) for 36898031
(Assignment LHS, `assign1=') for src=
(Assignment LHS, `assign2=') for term=
(Start Label, `<script>') for <script>
(Function Name, `alert') for alert(
(Integer Argument, `(1)') for (123)
(End Label, `</script>') for <script>
(Assignment LHS, `assign3=') for args=
(Assignment LHS, `assign4=') for qs=
(Integer Constant, `06') for 06
(Identifier, `o') for o
```

Figure 1: CBOW Architecture



Note that if two different token types overlap, the one that starts earlier is preferred; if they start in the same place then the one that extends further is preferred. This is why 'args=qs=' is taken to be two left-hand sides–the 'qs' could be a right-hand side, but with the equals on the right its taken to be a left-hand side since that is longer and both start at the 'q'. As per the last two tokens, identifiers cannot start with numbers (in Javascript) and so the 060 is taken to be an integer constant followed by an identifier.

3.2 Word2Vec

As one of the primary stated contributions of the original DeepXSS [4] paper, properly implementing CBOW for Word2Vec is an important part of our recreation. Unfortunately, the authors do not give good indication as to how CBOW was applied, merely stating that CBOW was used to construct word representations. We therefore took liberties with our implementation and based many decisions on the work from [1]. One major point of confusion was whether CBOW representations were constructed for token type labels, or the token itself. To test the difference, we implemented both.

In order to generate the CBOW embeddings we first needed to build a vocabulary of tokens with each token represented by a unique integer. Zeros represent padding tokens which are used later in the process. Once the vocabulary is constructed, each list of URL tokens is replaced with corresponding integer representations. Every vectorized URL is then padded with zeros until they are all a uniform length. The padding tokens ensure that empty space does not influence the outcomes of the model. Figure 3.2 shows the architecture for the model we use to generate our CBOW embeddings. For every token in our vectorized URL, the CBOW model attempts to predict the current token using the tokens around it. Using a window size of two means that our CBOW model will attempt to predict the middle token using two adjacent tokens on either side of it. As the model does this for every token in the dataset, the weights of the embedding layer are trained. The embedding layer is responsible for taking each token and turning it into a multidimensional representation. In the case of token type labels, each token will have a ten-dimensional representation and token values receive a 50-dimensional representation.

Listing 2: Related token values

```
<script>: >,ident1,</title>,path4/,<h1>
alert: ident11,ident13,ident12,<img>,path4/
</script>: <h1>,<marquee>,ident2,ident4,ident5
(1): ident4,<"<<script>,javascript,</style>,</title>
path0/: assign0=,ident1,ident0,</title>,<iframe>
```

Extracting the weights from the embedding layer and calculating the Euclidean distance between tokens allows us to see an example of similar words according to our CBOW embeddings. Listing 2 shows examples of the five most closely related token values and Listing 3 shows examples of the five most closely related token types.

Listing 3: Related token types

```
path: int_const,value,int_arg,func_name,assign
int_const: path,value,assign,func_name,int_arg
assign: value,other,int_cons,int_arg,str_arg
value: assign,other,int_const,int_arg,func_name
ident: other,start_label,value,assign,int_arg
```

Training each new CBOW model on the entire dataset could take as many as 24 hours meaning we needed to make changes to the tokenization process to produce better generalized data. This sped up CBOW generation but likely resulted in some lost semantic information.

3.3 LSTM Classifier

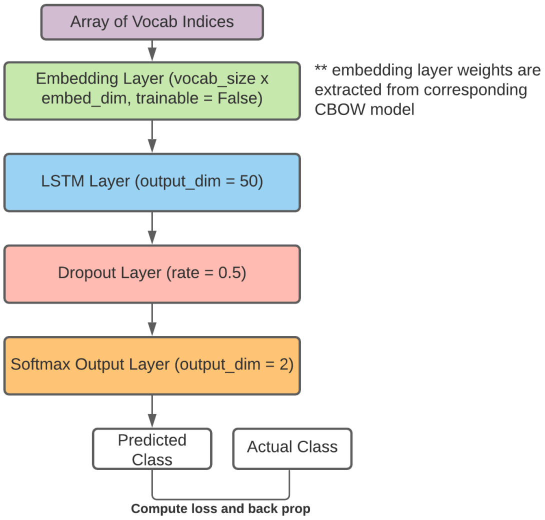
The final stated contribution of DeepXSS [4] and arguably the most important is the LSTM classifier for XSS detection. The authors provide a brief description of this classifier stating that it has three layers: an LSTM layer, a dropout layer to reduce overfitting, and a softmax output layer that makes predictions. We had two primary difficulties in reconstructing the model from this description.

The first difficulty was with using the embedded token vectors in the LSTM. We often had large numbers of embedded token vectors for every URL. This resulted in large, nested arrays of data that were difficult to work with. We chose to transplant a non-trainable embedding layer from our CBOW model into the LSTM model as shown in 3.3. This allows us to simply feed the model sequences of integers where the embeddings are constructed within the model rather than during preprocessing.

The other difficulty we had with constructing and training the LSTM model was with hyperparameters and model details. We found that the LSTM was often very difficult to train, seemingly making no progress after multiple epochs. Experimenting with dropout rates, the output size of the LSTM, and several other factors was extremely time consuming. Eventually we were able to settle on the model in 3.3 as our primary architecture for both token type and token value.

We also constructed other models to test the importance of various design choices from the original DeepXSS paper [4]. The first

Figure 2: LSTM Architecture


Array of Vocab Indices
Embedding Layer (vocab_size x embed_dim, trainable = False)
** embedding layer weights are extracted from corresponding CBOW model
LSTM Layer (output_dim = 50)
Dropout Layer (rate = 0.5)
Softmax Output Layer (output_dim = 2)
Predicted Class
Actual Class
Compute loss and back prop

Model	Output	Embedding	Input
Value-Softmax	softmax	CBOW	Token values
Value-Sigmoid	sigmoid	CBOW	Token values
Value-Sequential	softmax	None	Token values
Value-Random	softmax	Random	Token values
Type-Softmax	softmax	CBOW	Token types
Type-Sigmoid	sigmoid	CBOW	Token types
Type-Sequential	softmax	None	Token types
Type-Random	softmax	Random	Token types

Table 3: Summary of Models

model is our primary design that is shown in 3.3. We also created a model with a sigmoid output layer, given that a softmax output layer is somewhat unnecessary for a dual classification model. Another model is intended to test the value of the CBOW embeddings by simply training the model on sequences of integers that map to all of the tokens. This model has no embedding layer. The final model has an untrained embedding layer to test if the CBOW embeddings needed to be trained separately or if word embeddings could be learned through normal training. All of these models exist for both token type and token value resulting in eight models total. The models are listed in 3.

Prediction	XSS	Not XSS
Predicted XSS	T_p	F_p
Predicted Not XSS	F_n	T_n

Table 4: Confusion Matrix

Model	Precision	Recall	F1	Accuracy
DeepXSS	0.995	0.979	0.987	n/a
DeeperXSS:softmax	0.989	0.973	0.981	0.981
DeeperXSS:sigmoid	0.988	0.976	0.982	0.983
DeeperXSS:no embed	0.991	0.956	0.973	0.975
DeeperXSS:random embed	0.099	0.097	0.098	0.56

Table 5: Token Value Model Comparison.

4 EVALUATION

Our implementation mimics the original paper's use of 10-fold cross validation wherein the model is divided into ten equal subsets. The model is trained ten times for ten epochs. Each subset is used as a validation dataset exactly once, with the remaining data serving as training data. The final results are an average of the ten models. This process was repeated for all eight of the models we tested. The confusion matrix we show in 4 is identical to the one used in DeepXSS [4]. T_p (True Positive) indicates an actual XSS sample predicted to be XSS. T_n (True Positive) indicates a benign sample predicted to be benign. F_p are benign samples classified as malicious and F_n is a malicious sample classified as benign. These definitions are used to inform our key metrics, precision, recall, F1, and accuracy. The calculations for these are shown in the formulae below.

$$\text{Precision} = \frac{T_p}{T_p + F_p}$$

$$\text{Recall} = \frac{T_p}{T_p + F_n}$$

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Accuracy} = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$$

Tables 5 and 6 outline the comparison between our various models using both token values and token types respectively. Our results are ultimately very similar to DeepXSS [4] for the primary models with the softmax output layer. This indicates that our model was an acceptable recreation of the original. All other models also perform well with the very notable exception of the model with the randomly initialized embedding layer.

5 RELATED WORK

Mokbal *et al.* created a multilayer perceptron (MLP) model for detecting XSS both in dynamic webpages and URLs [10]. Their approach, called MLPXSS, has three main pillars: data scraping,

Model	Precision	Recall	F1	Accuracy
DeepXSS	0.995	0.979	0.987	n/a
DeeperXSS:softmax	0.989	0.978	0.981	0.982
DeeperXSS:sigmoid	0.987	0.977	0.982	0.983
DeeperXSS:no embed	0.983	0.955	0.969	0.97
DeeperXSS:random embed	0.089	0.091	0.08	0.555

Table 6: Token Type Model Comparison.

feature extraction, and an artificial neural network (ANN). Their feature extraction level has three modules to extract HTML-based features, Javascript-based features, and URL-based features. The HTML module tokenizes tags, attributes and events–focusing on things that trigger Javascript execution (like *href* or *onclick*). The Javascript module parses and tokenizes Javascript code that is pulled from a webpage creating an abstract syntax tree. There are various ways to include Javascript in a page like script tags, *onclick* and *onsubmit* calls, *href*, etc... Lastly they tokenize potentially malicious parts of URLs, like HTML properties, tags, some keywords (*login*, *signup*), *document* references, and various special characters like '<', '>' and '/'. The MLP is trained on token streams with a sigmoid output layer. Their perceptron had precision, f-measure, and accuracy all in excess of 99% [10].

Zhang *et al.* propose a dual Gaussian mixture model (GMM) approach that trains two seperate GMMs models (one for benign and one for malicious) and then combines their outputs to make a prediction. Additionally, they train the models on both the URL and the server response to the URL in an attempt to get richer features [16]. To preprocess the URLs, they decode, tokenize, and train a Word2Vec model to retrieve a vector representation of each token. Their tokenization approach is very similar to that of DeepXSS and MPLXSS, they however inclue the domain and path for the benign GMM. They are however generalized to simply 'domain' and 'path' [16]. Their reasoning for this is that containing just a domain and path is characteristic of a benign URL, whereas an XSS URL is characterized by its maliciously constructed parameters and not the presence of a domain and path [16]. Their models can be trained on requests, responses, or both. They reason that in many cases, benign requests contain no XSS tokens, which isn't very interesting, however responses contain useful features for both XSS and none-XSS tokenization. Their multi-stage dual GMM using both responses and requests greatly improved classification [16].

Goswami *et al.* propose an attribute clustering technique to perform unsupervised grouping of malicious and benign scripts. They're feature extraction is wholly different from DeepXSS and other deep learning classifiers. They propose 15 features that characterize malicious and benign scripts creating a 16-dimensional vector for each script (including class) [6]. These features are meta-features like length of the script, number of strings and the average string length, number of methods, number of unicode and hex characters, among others. These features are then min-max normalized before clustering [6]. Their algorithm was able to achieve an accuracy in excess of 98% [6].

The authors in [1] focus on classifying broadly defined malicious URLs sent through email or over social networks. In this case a malicious URL is any URL that could result in harm to the user visiting it. They propose a hybrid deep-learning approach called *URLdeepDetect* to extract semantic features from URLs to classify them as either benign or malicious. The preprocessing stages tokenizes various parts of the URL before applying word-level embedding through Word2Vec. The embedded tokens are then fed to an LSTM where samples are classified based on LSTM outputs or k-means clustering. This paper claims 98.3% accuracy for LSTM classification of malicious URLs and 99.7% accuracy with k-means clustering. The authors claim that the success of their approach is in part due to the Word2Vec token embedding and maintaining URL sequence as it provides the model with more semantic information for classification.

The work done in [14] explores XSS detection using three machine learning algorithms: Naïve Bayes, Support Vector Machine, and J48 Decision Tree. This paper attempts to detect reflected XSS, persistent XSS, and DOM based XSS meaning it requires web page scripts as well as URLs. The malicious URLs and scripts were collected from the XXSed [15] project and the benign samples were collected from the Dmoz open directory project [3]. This work, however, only uses five features for scripts (e.g., number of characters, request for cookie, etc.) and seven features for URLs (e.g., number of characters, presence of script tags, etc.) rather than extracting tokens to represent the entire URL or script. Each model was then subjected to 10-fold cross validation and the results were compared. J48 performed the best based on features from both URL and JavaScript achieving a 99% true positive rate and 99% precision. The authors also found that discretized attributes provided the best classifier results for J48.

Detecting XSS attacks on social networking sites is the primary focus of [13]. The authors propose an approach consisting of feature identification, web page collection, feature extraction, building a training dataset, and using a machine learning algorithm to classify web pages as XSS or non-XSS. This approach extracts features from URLs, HTML tags, and the host social networking site. The features of these web pages are very coarse-grained, consisting of things like the maximum size of URLs, and counts of harmful keywords. The authors then train ten classifiers with a RandomForest classifier achieving 97.7% precision and 97.1% recall. They suggest future work to enhance the feature set and apply more machine learning algorithms such as deep learning.

6 DISCUSSION

This approach is entirely dependent on the scope and correctness of the decoder. The filter evasion techniques used for XSS can be extremely complicated and can include multiple and mixed encodings with near arbitrary white space [12]. To demonstrate this sensitivity, we applied the standard blacklist/whitelist sanitizer **bleach** to both the decoded and non-decoded malicious URLs to test the relative efficacies [5]. On non-decoded data, the sanitizer only cleaned 62% of malicious samples, while on the decoded data it cleaned 91% of samples, clearly showing the importance of decoding. Furthermore, new evasion techniques are continually being discovered as well. The decoder has to be sophisticated enough to handle the evasion techniques of novel XSS payloads to be useful in practice. That said, a good decoder will be hard to evade and probably only a negligible number of XSS payloads will get past it, on the other hand, given the severity of the consequences of some XSS attacks, no number is negligible.

As with all machine learning, this approach is very sensitive to the preprocessing of the data and the feature extraction. In our case, this is the tokenization step. There have been many different tokenization procedures proposed; often they exclusively looked for tokens that would indicate a malicious URL and largely ignored the benign segments of a URL [4][10]. This approach means that many benign URLs contain no tokens [16]. This sort of approach requires the designer of the tokens to judge what kinds of strings characterize malicious URLs and only tokenize those which can be hard to do well. To address this, perhaps both the URL and the server's response to it can be tokenized which gives a much richer set of features and would help fill out the empty URLs [16]. This has the drawback of having to interact with the server for each URL which adds overhead–ideally the URL provides sufficient information. In our approach, we had a more general tokenization approach that aimed at tokenizing URLs and not just malicious URLs. The idea being that the machine learning algorithm can be left to figure out what sort of strings and patterns characterize malicious URLs without much input from us. This means benign URLs aren't empty and the server response is not required. That said, there are likely benefits to including special strings that give a strong indication of an XSS payload.

With regards to the models themselves, our results indicate that we have achieved a successful recreation of the original DeepXSS paper validating claims of highly effective XSS classification [4]. This is a reasonable conclusion given the empirical effectiveness of deep learning methods for classification problems. DeepXSS does have slightly higher metrics than DeeperXSS and the discrepancies can likely be explained by a few key limitations. We were unable to use the exact dataset used for DeepXSS, we lacked specific details about the Word2Vec implementation and the exact nature of the tokens it was applied to, and we lacked details as to how the embedded tokens were used as inputs to the model. With more time we likely could have improved our results with more training epochs, tuning hyperparameters like the dropout rate, and more rigorous data preparation to remove unsuitable samples.

Interestingly, the results trained with token type and token value are almost identical. The models without token embeddings also perform comparably well to the original. This calls into question the significance of the Word2Vec step that is cited as a primary contribution of the original work. At best the word embeddings provide a marginal boost to the model's metrics. One explanation could be that the model does not heavily rely on semantic information surrounding the tokens but rather focuses on features like the presence of specific tokens, the length of the URL, and the sequence particular tokens occur in. This said, a properly trained CBOW embedding layer massively outperformed a randomly initialized embedding layer trained alongside the classifier. Training the CBOW model was necessary in order to produce usable word embeddings. Future work should focus on isolating the most important features for classification.

7 CONCLUSION

Detecting and preventing XSS attacks remains an important domain of research. As such it is all the more important that research into this area remains reproducible and usable for practical purposes. In this paper we recreate, validate the DeepXSS approach for XSS detection. We also make comparisons to additional models to validate various claims. Ultimately we found that while DeepXSS is an extremely effective XSS classifier, claims surrounding the importance of Word2Vec for extracting semantic information are exaggerated. Our results indicate that deep learning should continue to be pursued as a means of classifying XSS. Future work should focus on extending this approach to web pages and embedded scripts to detect stored XSS attacks. Additionally, expanding the recursive decoder to handle more encodings, and isolating the most important features for detecting XSS is an important extension of this work. The final goal should be integrating these classifiers with working systems so that they can help with real time detection and prevention of XSS attacks.

REFERENCES

[1] Sara Afzal, Muhammad Asim, Abdul Rehman Javed, Mirza Omer Beg, and Thar Baker. 2021. URLdeepDetect: A Deep Learning Approach for Detecting Malicious URLs Using Semantic Vector Models. *Journal of Network and Systems Management* 29, 3 (2021), 1–27.

[2] Oxana Andreeva, Sergey Gordeychik, Gleb Gritsai, Olga Kochetova, Evgeniya Potseluevskaya, Sergey I Sidorov, and Alexander A Timorin. 2016. Industrial control systems vulnerabilities statistics. *Kaspersky Lab, Report* (2016).

[3] DMOZ. [n. d.]. The Open Directory Project. http://www.dmoz.org/

[4] Yong Fang, Yang Li, Liang Liu, and Cheng Huang. 2018. DeepXSS: Cross site scripting detection based on deep learning. In *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*. 47–51.

[5] Mozilla Foundation. [n. d.]. Bleach. https://pypi.org/project/bleach/

[6] Swaswati Goswami, Nazrul Hoque, Dhruba K Bhattacharyya, and Jugal Kalita. 2017. An Unsupervised Method for Detection of XSS Attack. *Int. J. Netw. Secur.* 19, 5 (2017), 761–775.

[7] Shashank Gupta and Brij Bhooshan Gupta. 2017. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management* 8, 1 (2017), 512–530.

[8] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[10] Fawaz Mahiuob Mohammed Mokbal, Wang Dan, Azhar Imran, Lin Jiuchuan, Faheem Akhtar, and Wang Xiaoxi. 2019. MLPXSS: an integrated XSS-based attack detection scheme in web applications using multilayer perceptron technique. *IEEE Access* 7 (2019), 100567–100580.

[11] OWASP. [n. d.]. OWASO Top Ten. https://owasp.org/www-project-top-ten/

[12] OWASP. 2021. XSS Filter Evasion Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

[13] S. Rathore, P.K. Sharma, and J.H. Park. 2017. XSSClassifier: An efficient XSS attack detection approach based on machine learning classifier on SNSs. *Journal of Information Processing Systems* 13, 4 (2017), 1014–1028. https://doi.org/10.3745/JIPS.03.0079 cited By 42.

[14] BA Vishnu and KP Jevitha. 2014. Prediction of cross-site scripting attack using machine learning algorithms. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*. 1–5.

[15] XSSED. [n. d.]. XSSed Cross Site Scripting (XSS) attacks information and archive. http://www.xssed.com

[16] Jingchi Zhang, Yu-Tsern Jou, and Xiangyang Li. 2019. Cross-site scripting (XSS) detection integrating evidences in multiple stages. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*.