Detecting Injection Attacks using Long Short Term Memory

Ioana-Crinela Potinteu, Robert Varga
Technical University of Cluj-Napoca
Computer Science Department
Email: crinela.potinteu@student.utcluj.ro, robert.varga@cs.utcluj.ro

***Abstract*—Nowadays, the rise of cybercrime requires the companies to empower their business with innovative defensive mechanisms. This article focuses on the security of APIs and it proposes two methods to defend them against different types of injection attacks. The analyzed APIs belong to a large scale company and they respond to more than 10 000 requests per second, coming from applications that belong to totally different business domains. At the foundation of these solutions is the usage of the powerful Long Short Term Memory network and two different tokenization strategies: a word-level and a character-level. The solutions outperform the current model used by the company and they are able to detect an injection attack tentative sent by a client with 96% accuracy, in comparison with the model employed, which only achieves 92% accuracy.**

I. Introduction

Companies are paying more and more attention to Cyber Security and one reason is the fact that their reputation will be irreparably damaged by any data leak. Laws that address hacking were promulgated in many countries around the world. For example, in UK the punishments include prison sentences from 1 to 20 years [1], but the attackers do not seem to be scared. It is the job of software developers to take into consideration aspects related to security from the design phase of an application.

Even if many security best practices are respected, the hackers are searching for more innovative and unexpected attacks and this forces the developers to also think about creative and effective defense mechanisms [2]. This work proposes new ways to protect APIs (**A**pplication **P**rogramming **I**nterfaces). The APIs that were analyzed are designed according to REST principles. REST is an architectural style for distributed systems that is very popular for designing web services [3]. The proposed solutions do not use any particularities of this architectural style, so they should be suitable for any normal URL.

An API running on the server will respond to the requests of different clients. The clients can be mobile, web or desktop applications, running on different operating systems or browsers. The defending solutions presented in this article are designed for the case when the communication between the client and the server is done though HTTP (**H**yper **T**ext **T**ransfer **P**rotocol). Web services are built around the concept of *resource* [4]. Any piece of information that can be uniquely identified is a resource, for example an image. The identifier is called **URL** (**U**niform **R**esource **L**ocator).

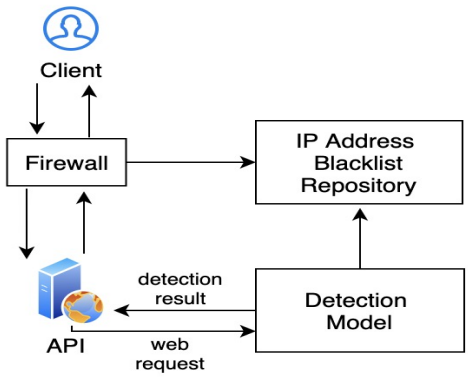

Fig. 1. Possible use case for malicious web request detection

The clients of a web API send an HTTP request message to retrieve a resource, this message is mandatory to contain the URL of that particular resource. The API responds with a status code that indicates that the resource was successfully accessed or it provides information about the error that occurred. If it is the case the resource is sent to the client.

In this context we analyze and propose two different methods that are able to automatically detect malicious web request sent to an HTTP API. Malicious web requests are those that contain in the URL parameters or code sequences that have bad intentions. Among the bad intentions we mention obtaining confidential data or modifying a table from a database without the required permissions. This type of attack against a web API is called injection attack.

A possible use case of the detection models that will be presented can be seen in Figure 1. A client application sends to the API a request to retrieve or to save resources. Firstly, the request passes through a firewall. The firewall analyses all the data packages that form the request and based on a set of rules it can block the traffic from suspicious sources.

In order to enhance the security of a web API, a rule aimed to verify if the IP address of the source belongs to a blacklist, can be added to the firewall, initially the blacklist is empty. Every time the Detection Model receives a request that will be classified as malicious, it will add the source IP to the blacklist and it will return the result to the API. The API processes only the requests that are classified as being normal, for the malicious requests it returns an error message.

978-1-7281-9080-8/20/$31.00

The rest of the paper is organized as follows. Section II surveys the related work about convolutional neural networks used for anomaly detection. Section III presents two approaches to build training, validation and test datasets, Section IV details the proposed architecture, Section V shows the results and Section VI summarizes out methods and enumerates possible further developments.

II. Related work

This section presents anomaly detection methods that use log files written by programs in execution. Logs are similar to URLs, they are unstructured data, their format and semantics vary from one system to another. The work [5] presents an architecture of a neural network that detects anomalies from logs. The authors treat logs as natural language and apply classification methods specific to this domain. A script was used to build training and validation datasets. The script executed the following tasks: create, delete, start, stop, pause and resume virtual machines using OpenStack in CloudLab.

All the log entries were parsed and for each a key was extracted, called *log key*. The log key of a log entry e is the constant string k from the print statement that generated it. For example, the log key k for the log entry e = *"Took 10 seconds to build instance."* is k = *Took * seconds to build instance.*, which is the constant string from the statement: *print("Took %f seconds to build instance.", t)*. It can be noticed that the parameters are abstracted as asterisks.

A tuple is obtained in the form of *(key, parameter value vector)*. The log keys sequence is used to train a log key anomaly detection model and the sequence of parameter value vectors is used to train a model that decides if the parameter values are normal for this log entry. Both detection models use a **L**ong **S**hort **T**erm **M**emory (LSTM) network. Its architecture was introduced in [6], it is a subclass of *Recurrent Neural Networks*, which are networks designed to classify sequences of data. LSTM can be imagined as a chain of multiple copies of the same network and each passing information to the next one.

The input of the LSTM network used in [5] is a list of keys or parameter value vectors that correspond to a window of recent consecutive log entries. One LSTM network is trained to maximize the probability of having key k_i as the next key. The other LSTM network is used to minimize the error between a parameter value vector that was predicted using the other vectors from the window and the one that was extracted for that particular key. This solution achieves very good values for F1-score when the code running on the monitored system is not changed. If the code is updated and new sequences of log entries appear, the value of F1-score is only 28.0.

The article [7] presents an architecture to detect anomalies in system logs. For training and validation the authors used the *HDFS log* dataset, that was proposed in [8]. The logs were parsed in the same way as in the previous article, but only the keys were kept. They have designed a neural network with the following architecture: an *Embedding* layer, followed by 3 *Convolutional 1D* layers applied in parallel to the output of the Embedding layer, the output of these three layers is concatenated and sent to the next layers: *Dropout, Max-pooling* and at the end a *Fully-connected* layer with *softmax* activation.

The Embedding layer maps a key to a dense vector of float values. This dense vectors are learned during the training of the neural network. Here it can be noticed that problems also appear when the code of the monitored system is updated and new log keys are produced. All the new log keys that were not seen during training are mapped to the same dense vector, assigned to unknown log keys. The authors presented the result of F1-score only when the code was not changed and the value is very high: 98.5 ± 0014.

The paper [9] classifies files containing log entries as belonging to programs that run normally, or programs that encountered different anomalies during execution. To obtain the dataset for training and validation the authors use log files from systems that were monitored and they experienced only normal behaviors and also they have injected faults to obtain anomalous behaviors.

A sequence of preprocessing operations were applied to all the labeled log files. Firstly, all the non alphanumeric characters were removed. Secondly, the algorithm *word2vec* presented in [10] was used to map each word from a log entry to a dense vector of float values. This vector represents the position of the word in an Euclidean space. After this step all log entries become lists of vectors. By computing the average of all the vectors that form a log entry it is obtained a single vector that represents the position in the Euclidean space for the entry. Finally, the position of the log file is obtained by computing the average of the vectors computed for each entry.

The vector computed for each file, together with the associated label were used to train a Convolutional Neural Network and other two classifiers: *Naive Bayes* and *Random Forest*. The metric used for comparison was *AUC* (*Area Under the Curve*). For the Convolutional Neural Network and Random Forest the value of AUC was 95% and for Naive Bayes the value was 77%.

III. Dataset construction

The datasets used to train, validate and test the model were built using two files received from a team of Cyber Security experts. One file contains normal URLs that are expected to be received by an API. The other file contains URLs that can be used by an attacker to perform an injection attack. All the requests were made to multiple APIs that we intend to protect using the resulted detection models. The total number of labeled URLs was 4 705 156. There were 3 466 443 (73.67%) normal URLs and 1 238 713 (26.33%) malicious URLs. A percentage of 80% from these URLs was used to train the neural network that will be detailed in the next section, while the rest was used as a validation dataset. The test dataset was received in a separate file from the Cyber Security experts, it contains a total of 10 000 URLs, together with the corresponding labels, there are 1000 malicious URLs and the rest of them are normal.

Two different strategies to build a dataset were explored. The first one converts a URL into a format similar to natural language and each URL will become a list of words. The second strategy will not apply any preprocessing operation, the raw URL will just be separated into the list of characters that it contains.

A. Convert a URL to natural language

In this paper we propose an original method to convert a URL into a format similar to natural language. The method was inspired by the algorithm to extract a key from a log entry, proposed in [5]. The operations required by this transformation are simple for a normal URL, but for a malicious one they depend on the type of injection attack that they contain. The following list contains all the injection attacks that we aim to detect together with the operations that will be performed to the URL:

- **SQL Injection**: this type of attack tries to modify the query that will be executed by the database engine to retrieve the requested information. For example, in a web application that sells products from different categories, the URL to see the category *navigation* can look like this:

  `/products?category=navigation`

  If the application does not have any mechanism to protect against this type of attacks, then the attacker can modify the URL to obtain more information than allowed:

  `/products?category=navigation'+OR+1=1`

  Special characters, like ' in this case, appear frequently in malicious URLs. In a traditional natural language processing application they are removed [11], but in this case their presence helps the network to detect this attack. To defend against this type of attack the special characters can be left unchanged or they can be replaced by tokens. They were replaced by tokens as the idea of this strategy is to convert a URL to a format similar to natural language, the next presented strategy leaves them unchanged. The URL becomes:

  `/products?category=navigation tick plus OR plus equality`
- **Command Injection**: aims to execute different operating system commands on the server where the API is running. Typically the APIs that need to call shell commands to retrieve some data are vulnerable to this type of attack. Similar to the case of SQL injection attacks, the typical characters used are: *|, ;, //* to insert and separate the commands into the shell. These special characters should be replaced with tokens, for example: *pipe, semicolon* and *slashslash*
- **Cross Site Scripting (XSS)**: is a vulnerability that allows an attacker to manipulate a vulnerable web site and return malicious JavaScript content to regular users. For example, a forum that receives messages from the users and displays them to the other participants, can use simple HTML elements:

  `<p>Message text</p>`

  An attacker can send the following message:

  `<p><script>alert(1)</script></p>`

  This is a harmless attack, it just displays a pop up with the message *1*, but these type of attacks can be dangerous, in worst cases the attacker can control the browser remotely. As it can be noticed, each special character *<, >, (,), [,],* {, } must be replaced with a token.
- **Path Traversal**: is used to access files that are saved on the server where the API is running. These files can contain sensitive information. For example, if the path of an image that should be retrieved by the API is sent into the URL, the attacker can change it to look like this:

  `/resources/images/../../../etc/passwd`

  The characters *../* are used to access the parent directory and they will be replaced with the token *dotdot*. This request tries to read the file *passwd*, for Linux systems it contains data about the users of the system.
- **Double encoding**: in URL some characters have a special role, for example & separates the query parameters. Also, a URL can contain only letters from the US-ASCII set. To send characters outside of this set they have to be encoded using a technique called *URL Encoding*. This technique converts the special characters to a universal format accepted by all the browsers and web servers. By default, the browsers and web servers decode the URL only once. Also, they may have some filters to protect against *Path Traversal* and *Cross site scripting* attacks. These filters detect the sequence *../* , the characters *<, >, (,), [,],* {, } or their codification. To bypass these filters the attackers will double encode the special characters. The backend platform, that knows how to handle encoded data, performs the second decoding process and this platform might not have implemented the corresponding security checks. To convert a URL to natural language the first step will be to double decode it.

Previously we presented operations that will be applied to URLs depending on the different types of injection attacks that we aim to detect. In the following a list of general operations that can be performed is proposed:

1) **ID abstractization**. All the IDs that uniquely identify a resource should be abstracted with the term *identifier*. The neural network just needs to learn that some URLs require an ID and it should not learn all the IDs.
2) **Timestamp abstractization**. All the timestamps should be abstracted with the term *timestamp*, regardless of their format.
3) **Abstract Click Identifiers**. For example, *Google Click Identifier* or *Facebook Click Identifier* should be replaced with the tokens *fbclib, glid*.
4) **Abstract the names of the files**. Client applications send many requests for different files. The neural network should learn how the request for an image is made, regardless of its name or extension. To perform this operation in the URL are identified the names of the files and they are replaced with their type, for example:

TABLE I
DETAILS OF THE DATASETS

Dataset	Total	Malicious	Normal
Raw Train	3 764 124	991 093 (26%)	2 773 031 (74%)
Raw Validation	941 032	247 773 (26%)	693 259 (74%)
Raw Test	10 000	1 000 (10%)	9 000 (90%)
NLP Train	1 161 548	243 040 (21%)	918 509 (79%)
NLP Validation	290 388	58 077 (21%)	232 310 (79%)
NLP Test	10 000	1 000 (10%)	9 000 (90%)

```
/birdseye/usgs/drg_24_n4870-3m.jpg
/birdseye/usgs/image
```

5) **Abstract the properties of the images**. If in the URL is sent the dimension for the requested image, it should be abstracted by the term *dimension*.
6) **Eliminate the numbers**. If in the URL numerical parameters are sent, for example *startPageIndex, pageSize*, they should be eliminated.
7) **Replace the characters /, _, =, &, :, ?, - with space**. These characters are used to separate the information from the URL, after this operation the URL will become a list of words.
8) **Convert upper case to lower case**. This operation is very used to preprocess natural language.

B. Datasets summary

We have built 6 datasets from the files with labeled URLs, they can be seen in Table I. All the datasets whose name starts with *Raw* contain URLs that were not preprocessed. The ones whose name starts with *NLP* were converted to a format similar to natural language by using all the preprocessing operations that were presented above. Note that after the preprocessing operations, many duplicate URLs were generated and they were deleted.

C. Imbalanced data

Looking at Table I it can be noticed that the datasets are highly imbalanced. The number of normal URLs greatly outnumbers the number of malicious ones. The solution proposed in [10] was used to solve the problem. Different class weights were assigned to indicate that an error on a malicious URL is 7 times as expensive as an error on a normal URL.

D. Convert URLs to numerical form

Machine learning algorithms are able to work only with numerical values. The next step in the dataset construction is to convert all the URLs from each dataset to a sequence of numbers.

1) Word level codification: For the NLP datasets each word was mapped to an ID. The total number of unique words found into the dataset NLP Train was 316 171. The APIs are used in production for a wide range of client applications. One of them is a large electronics shopping website. The high number of unique words can be explained by the diversity of the web requests that the APIs serve and also by the fact that some of the client applications are translated in almost 80 languages. We have chosen the first 99 999 words, ordered by their frequency in the dataset, to represent the vocabulary of known words by the neural network. Up to this number the words had a frequency greater or equal to 2, after this number the words occurred only once into the dataset. Each word from the vocabulary was mapped to an integer (ID) by using a very simple algorithm.

The number of occurrences in the training set was counted for each word and the mapping is done using these values. The most frequent word was mapped to the lowest ID, starting from the value 2, ID 1 is reserved to replace words that are not part of the vocabulary. The second most frequent word was mapped to value 3 and this procedure continued until all the words from the vocabulary were mapped to IDs. All 99 999 words that form the vocabulary were mapped to IDs between 2 and 100 000. After building the vocabulary and determining the ID for each word and also setting an ID to represent all the words that do not belong to the vocabulary, all three NLP datasets were converted to a numerical form, that can be sent as input to a neural network.

2) Character level codification: For the Raw datasets each character was mapped to an ID. All the unique characters found into the dataset Raw Train represent the vocabulary. There were found 67 distinct characters and they were mapped to an ID based on their number of occurrences into the dataset. The most frequent character was mapped to 2, the second most most frequent was mapped to 3 and so on. All 67 characters were mapped to an integer number between 2 and 68, the ID 1 is used to map all the characters that do not belong to the vocabulary. All three Raw datasets were converted to a numerical form using the vocabulary and the mapping strategy.

IV. PROPOSED NETWORK ARCHITECTURE

In [5] LSTM is used to detect anomalies and the log keys represent the classes. In [7] and [9] a custom neural network is built and the log keys are the input of the first layer. In [7] the first layer is of type Embedding. Figure 2 depicts an architecture that combines the ideas from the analyzed articles and this is the proposed architecture to detect malicious web requests. The architecture is the same as the one presented in [12] to classify a movie review as being positive or negative. The only difference consists in the choice of the hyperparameters, they must be suitable for a significantly larger dataset. The datasets NLP Train and Raw Train were used to train two neural networks having the presented architecture, only some values of the hyperparameters were different. We explore two distinct ways of using LSTM to detect malicious web requests.

A. Embedding layer

This is the first layer of the neural network, it is used to convert each word/character ID into an *embedding*. Embeddings are defined in [13] as being trainable dense vectors that represent the position of a word in an Euclidean space, the dimension of this space is a hyperparameter. This layer takes as input a sequence of IDs that encode a URL (the IDs can

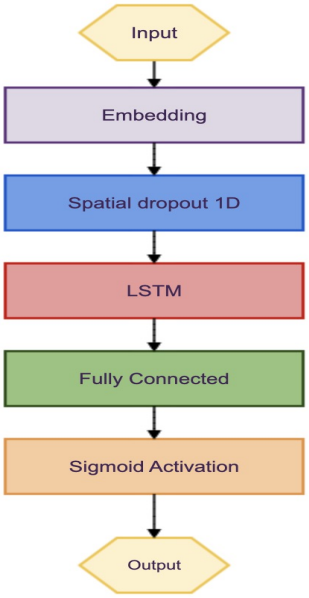

Fig. 2. Proposed network architecture

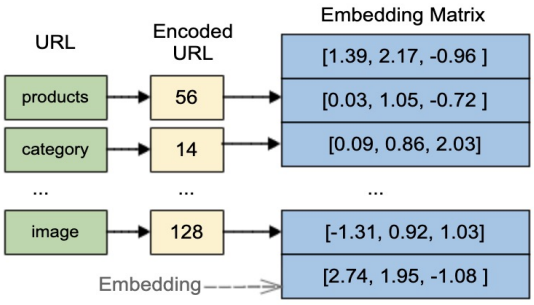

Fig. 3. Embedding layer overview

represent words or characters), then the layer will output a matrix containing the vectors associated to each ID. All the trainable vectors are saved in an embedding matrix.

The mapping between the IDs and the vectors is done very easily, the ID is used as index in the embedding matrix. Figure 3 illustrates the usage of this layer. The URL is encoded at word level and the dimension of the Euclidean space is 3. This layer was created to capture semantic similarities between words. The words that have similar meanings or that are used in related contexts should have close representations and the same reasoning applies to characters. The ID based codification does not provide information to decide if two words/characters are used in similar contexts.

B. Spatial dropout 1D layer

Dropout was introduced in [14] and is one of the most used regularization techniques in deep learning. This technique drops a neuron with probability p during training time. Spatial dropout 1D is based on this idea, but it drops with probability p every column from the embedding matrix. To drop a column means to set all its elements to 0. This layer is recommended to

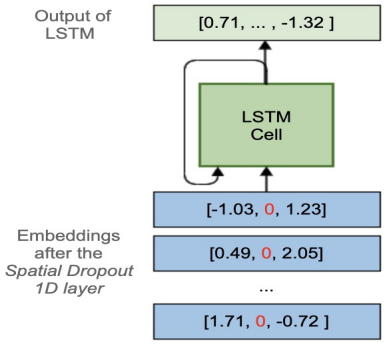

Fig. 4. LSTM module overview

be used after the Embedding layer, to decrease the dependency between the elements of dense vectors.

C. LSTM

Inside the proposed architecture the LSTM network can be seen as a layer that takes the output from the Spatial Dropout 1D layer, it performs the computations required for each embedding and then it sends the final output to the next layer. An LSTM network is able to remember information over arbitrary periods, its memory is called *cell state*. The network contains the following components: the cell state and 3 gates that are used to remove or add information to this state. The gates use the sigmoid function, that returns values between 0 and 1, to control how much from the output of these gates is used for the next computations. The three gates are: *forget* - decides what information should be eliminated from the cell state; *input* - selects what information will be added to the cell state; *output* - computes the final output of the module for each time step. The total number of time steps represents the length of the input sequence.

D. Fully Connected layer with sigmoid activation

This type of layer is often placed at the end of the deep learning architectures to obtain the prediction result. The input of this layer is the result of LSTM module, which is a vector of real values, the features that were learned from a URL. Using these features the Fully Connected layer with sigmoid activation returns the probability of a URL to be malicious or not.

E. The choice of hyperparameters

For some of the hyperparameters we have used the values recommended in [12], for others we have analyzed the datasets and for the rest we tried multiple values and choose based on the accuracy and the time needed to train and to make a prediction. The optimization algorithm was *Adam*, with the following parameters: *learning_rate = 1e-3, beta1 = 0.9, beta2 = 0.999* and *epsilon = 1e-7*. According to the recommendations from [12], the chosen loss function was *binary_crossentropy* and we take into consideration the weights for each class. The loss function for the i-th instance from the batch has the following formula:

TABLE II
INPUT AND OUTPUT SHAPES OF THE LAYERS

Layer	NLP Model	Raw Model
Embedding	In: (2000, 250) Out: (2000, 250, 128)	In: (2000, 1000) Out: (2000, 1000, 64)
SpatialDropout1D	In: (2000, 250, 128) Out: (2000, 250, 128)	In: (2000, 1000, 64) Out: (2000, 1000, 64)
LSTM	In: (2000, 250, 128) Out: (2000, 64)	In: (2000, 1000, 64) Out: (2000, 64)
FullyConnected	In: (2000, 64) Out: (2000, 1)	In: (2000, 64) Out: (2000, 1)

$$L_i = w(y_i) * (y_i * log(p(y_i)) + (1 - y_i) * log(1 - p(y_i))) \quad (1)$$

where w is the weight for class 0 and 1, respectively; y_i is the label of the instance i (0 for normal and 1 for malicious) and $p(y_i)$ is the probability of being malicious predicted by the model in the case of instance i.

The TensorFlow library was used for implementation. TensorFlow builds static graphs for the neural networks, as a consequence the dimension of the input data needs to be known when the graph is built. The URLs from all the datasets have different lengths, TensorFlow has the needed functions in order to bring them all to the same length. To choose a good value for the length we have built a histogram to see for each training the distribution of the lengths. The maximum allowed length for a URL was determined for both tokenization methods. The URLs that are shorter than this length will be padded at the beginning with the value 0. The URLs that are longer than this length will be truncated by eliminating values from the beginning.

The final input and output shapes for each layer are shown in Table II. The model trained on the dataset NLP Train (containing URLs encoded at word level) is called *NLP Model* and the model trained on the dataset Raw Train (containing URLs encoded at character level) is called *Raw Model*. The batch size for both architectures is 2000. The maximum length of a word-level encoded URL is 250 and for character-level encoded URL is 1000. Each word is mapped to an embedding vector having 128 elements and each character is mapped to an 64-elements vector. The output vector of the LSTM network has 64 elements for each URL, regardless of the codification method.

V. EXPERIMENTAL RESULTS

A neural network having the proposed architecture was implemented for each URL codification method. The development was done using Keras with TensorFlow as the backend. The two approaches to detect malicious web requests are evaluated on the test dataset received from the Cyber Security experts.

A. Evaluation measures

The models are evaluated by the standard metrics listed below: *TP (True Positive)* represents the number of real malicious web requests that were correctly detected as attacks by the model; *TN (True Negative)* is the number of normal web requests that are correctly identified as not malicious; *FP (False Positive)* shows the number of normal web requests that are incorrectly identified as malicious; *FN* represents the number of malicious requests that are classified as normal. Based on these four metrics we calculate: $BinaryAccuracy = \frac{TP+TN}{TP+TN+FP+FN}$; $Precision = \frac{TP}{TP+FP}$; $Recall = \frac{TP}{TP+FN}$ and $F1 - measure = \frac{2*Precision*Recall}{Precision+Recall}$.

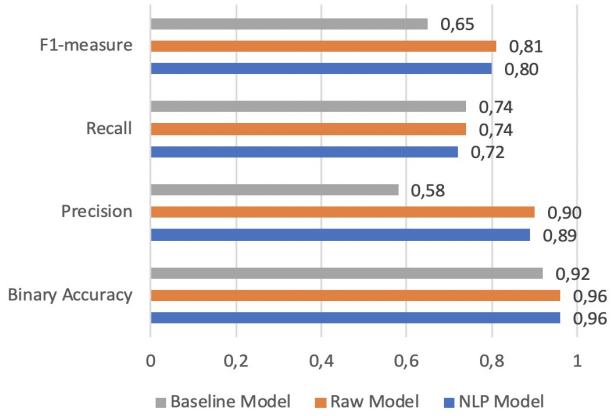

Fig. 5. Evaluation results on the test set

Figure 5 compares binary accuracy, precision, recall and F1-measure for the two models and the Baseline one. The Baseline Model was used by the company to block IP addresses that send malicious web requests. Its architecture was not published by the company. The proposed models obtained similar values for the enumerated metrics, the Raw model slightly outperforms the NLP model. The recall metric has the lowest values, this measures how often a malicious request was detected as such.

A 90% precision means that for every 10 detected attacks, one of them is a false positive, a legitimate request that was blocked. This clearly affects the usability of the provided web services, users can send by mistake URLs containing special characters that make them to be classified as malicious. At the cost of blocking some legitimate users, many truly malicious requests will be blocked. The cost of a successful injection attack is much greater and it can cause a lot of harm to the company.

B. Timing performance

The trained models are used in real-time to detect anomalies, in consequence it is important to have a small prediction time. The API processes a web request only if the model classifies it as not being malicious. The Table III shows the wall time needed by the two models to perform certain tasks. All tasks were executed on a computer having the hardware specification: Intel Core i9-7920X CPU, 128GB Physical System Memory and 4 NVIDIA GeForce RTX 2080 Ti GPU.

TABLE III
WALL TIME FOR DIFFERENT TASKS PERFORMED BY THE MODELS

Task	NLP Model	Raw Model
Train 1 batch (2000 examples)	0.94 s	4.00 s
Train 1 Epoch	539 s	6 715 s
Make 1 prediction	0.20 ms	0.62 ms
Preprocess 1 URL	0.16 ms	0.00 ms
Encode 1 URL	0.02 ms	0.02 ms
Total time to make 1 prediction	0.38 ms	0.64 ms

The first two tasks are related to the offline training. Training a model on a batch of character-level encoded URLs is more than 4 times slower than training a model on a batch of word-level encoded URLs. This is due to the fact that word-level codification is significantly shorter than the other. Also, the time to complete a training epoch is greater for the Raw Model because the dataset Raw Train is significantly larger than NLP Train.

The last row of Table III shows the total wall time to make a prediction about a URL, including preprocessing and encoding. The NLP model is 1.7 times faster than the Raw Model, even if the Raw Model does not require any preprocessing operations.

VI. CONCLUSION

This paper presents two solutions to defend high performing APIs against injection attacks. The APIs are used by a large scale company that operates globally. Cyber Security experts provided large-volume files containing labelled URLs. The large volume was due to the high variety of web requests that the APIs handle. The main difference between the proposed methods consists in the tokenization strategy that is used. The first method converts the URL to a format similar to natural language, resulting a list of words. The conversion algorithm is original and it was designed to be very general in order to be used for APIs that are handling very diverse requests, also it puts emphasis on capturing information specific to injection attacks. Each word is mapped to an integer ID and this mapping is used to encode each URL to a numerical format. The second method does not apply any preprocessing operation. In this case each character is mapped to an integer ID to convert the URL into a numerical format.

The datasets obtained by these tokenization strategies were used to train two neural networks that incorporate the powerful LSTM network. Both resulted models achieve 96% accuracy, this demonstrates that LSTM can be successfully used to detect web requests that contain injection attacks. Also, the models obtained a higher accuracy than the baseline model.

For the future work the imbalance of the data will be exploited for a better value of precision and recall, for instance by lowering the 0.5 threshold over which a request is classified as malicious, trying different values for the class weights or using focal loss from RetinaNet. Furthermore, other types of Recurrent Neural Networks will be incorporated into the architecture to test their efficiency.

REFERENCES

[1] UK Crown Prosecution Service, *Cybercrime - prosecution guidance*, https://www.cps.gov.uk/legal-guidance/cybercrime-prosecution-guidance, 2020-05-30.

[2] J. J. Jaccard and S. Nepal, *A survey of emerging threats in cybersecurity*, Journal of Computer and System Sciences, 2014, 80.5: 973-993.

[3] F. Tobias and P. Braun, *Model-driven testing of restful apis*, Proceedings of the 24th International Conference on World Wide Web. 2015.

[4] P. Sanjay, *Pro RESTful APIs*, Apress, 2017.

[5] D. Min, F. Li, G. Zheng and V. Srikumar, *Deeplog: Anomaly detection and diagnosis from system logs through deep learning*, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017.

[6] J. Schmidhuber and S. Hochreiter, *Long short-term memory. Neural Computation*, 9(8), 1735-1780, 1997.

[7] S. Lu, X. Wei, Y. Li and L. Wang, *Detecting anomaly in big data system logs using convolutional neural network*, 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 2018.

[8] W. Xu, L. Huang, A. Fox, D. Patterson and M. I. Jordan *Detecting large-scale system problems by mining console logs*, In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009.

[9] C. Bertero, M. Roy, C. Sauvanaud and G. Trédanet, *Experience report: Log mining using natural language processing and application to anomaly detection*, 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2017.

[10] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado and J. Dean, *Distributed representations of words and phrases and their compositionality*, In Advances in neural information processing systems, 2013.

[11] V. Sowmya, H. Surana, A. Gupta and B. Majumder, *Practical Natural Language Processing*, O'Reilly Media Inc, 2020.

[12] D. Sarkar, R. Bali and T. Sharma, *Practical machine learning with python - A Problem-Solvers Guide To Building Real-World Intelligent Systems*, Berkely: Apress, 2018.

[13] A. Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media, 2019.

[14] G. E. Hinton, N. Srivastava, A. Krizhevsky et al. *Improving neural networks by preventing co-adaptation of feature detectors*, arXiv preprint arXiv:1207.0580, 2012.