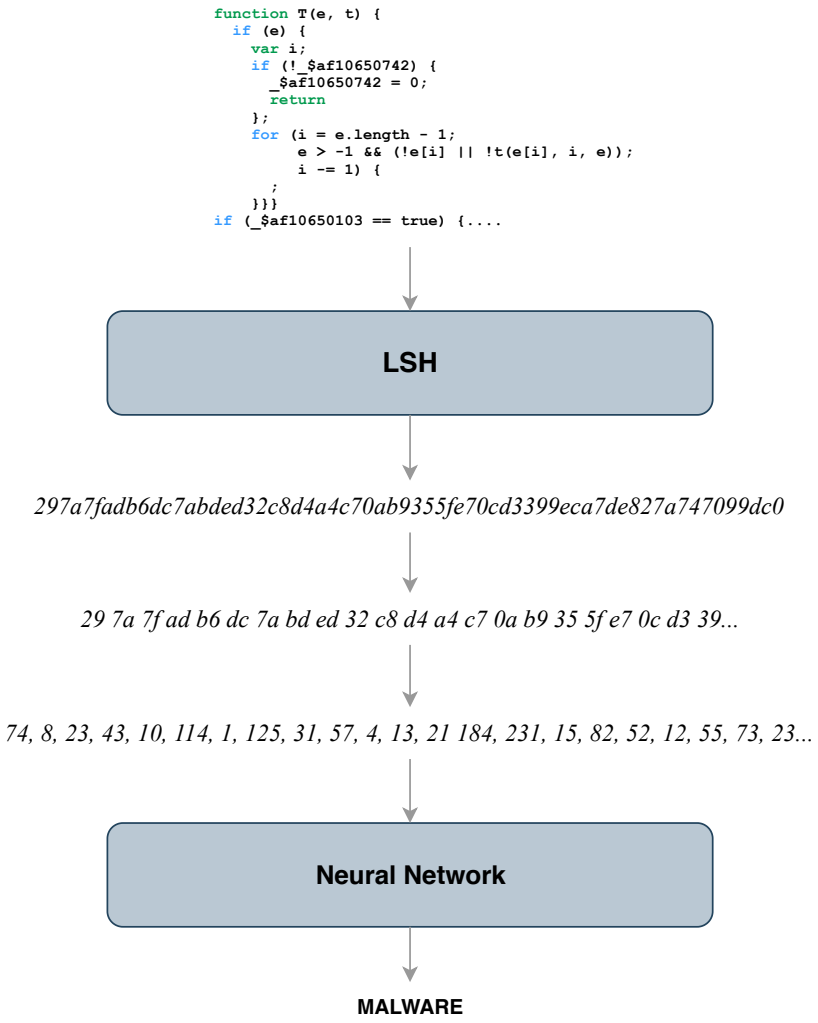
CHALMERS
UNIVERSITY OF TECHNOLOGY



Malware Classification using Locality Sensitive Hashing and Neural Networks

Master of Science Thesis in Software Engineering

Ludwig Friborg & Stefan Carl Peiser

Department of Software Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY & UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019:42

Malware Classification using Locality Sensitive Hashing and Neural Networks

LUDWIG FRIBORG & STEFAN CARL PEISER

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Software Engineering
Thesis report group 42
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2019

LUDWIG FRIBORG
STEFAN CARL PEISER

Supervisor: RICCARDO SCANDARIATO, DEPARTMENT OF SOFTWARE ENGINEERING
Company Supervisor: ARNA MAGNUSARDOTTIR, CYREN
Examiner: JENNIFER HORKOFF, DEPARTMENT OF SOFTWARE ENGINEERING

Master's Thesis 2019:42
Department of Software Engineering
Thesis group 42
Chalmers University of Technology and Gothenburg University
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

In this thesis, we explore the idea of using locality sensitive hashes as input features to a feedforward neural network to perform static analysis to detect JavaScript malware. An experiment is conducted using a dataset containing 1.5M evenly distributed benign and malicious samples provided by the anti-malware company Cyren, which is the industry collaborator for this thesis. Four different locality sensitive hashing algorithms are tested and evaluated: Nilsimsa, ssdeep, TLSH, and SDHASH. The results show a high prediction accuracy of 98.05% and low false positive and negative rates of 0.94% and 2.69% for the best performing models. These results show that LSH based neural networks are a competitive option against other state-of-the-art JavaScript malware classification solutions.

Keywords: locality sensitive hashing, static analysis, malware detection, artificial neural networks, machine learning, feature extraction

Acknowledgements

We would like to thank Cyren for all the help and support, especially Arna Magnúsardóttir for sharing her domain expertise. The knowledge and resources they have given us in form of access to data and computing power were essential to this project. We would also like to thank Riccardo Scandariato our supervisor and Jennifer Horkoff our examiner for providing valuable feedback.

Ludwig Friborg, Stefan Carl Peiser, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1

Introduction

Software Engineering encompasses a lot of different research fields, from finding the best way to write requirements to static code analysis. This thesis will focus on exploring a new method to perform static analysis to classify script files, or more explicitly JavaScript files where a JavaScript file's locality sensitive hash will be used as a feature to predict whether it is malicious or benign.

The method proposed in this thesis entails combining locality sensitive hashing with machine learning. Locality sensitive hashing (LSH) is a family of dimensionality reducing algorithms which are widely used in different fields like computer vision, recommendation systems, and more. In this thesis, LSH algorithms will be applied to static code to produce a new representation of the original script file, which will be more applicable for statistical learning.

Locality sensitive hashing has potentially many real-world applications where one of them is in the field of malware identification. There are two main approaches used to identify malware: static analysis and dynamic analysis. Static code analysis is a form of analysis where analysis is done on the malware (source code or compiled executable) directly without executing it, and the other is dynamic analysis where the malware is executed, and its behaviour monitored and judged. Static analysis is preferred as it minimises risks of malware spreading as well as potentially being a quicker way to classify files compared to dynamic analysis. Static analysis might be preferred when analysing large datasets of scripts since some might not be easy to execute correctly. Dynamic analysis in comparison can be harder to apply when analysing large datasets or unknown files since it requires sandbox environments or emulation, certainly in the circumstance of malware detection.

This thesis is done in collaboration with Cyren, an internet-based security company, focusing on anti-spam, anti-malware, and more. They are known for providing the anti-malware scanner that Gmail uses for attachments along with having their services integrated into the Microsoft Office 365 network to help combat malicious email.

Currently, Cyren's approach to JavaScript classification is dynamic analysis. Thus there is a demand to find a general way to identify JavaScript malware statically. As will be shown in the related work's sections, others have attempted to classify JavaScript malware using static analysis, but more commonly, dynamic analysis is used or in some cases both (hybrid analysis).

This thesis will focus on using LSH together with neural networks to classify JavaScript malware using data provided by Cyren, leading into the following research questions that are the focus of this thesis:

- Do neural network models with locality sensitive hashes as input features yield high accuracy along with low false positive and false negative rates when classifying JavaScript malware?
- How well do different LSH methods compare to each other as input into neural networks?
- How well do LSH based neural network models compare to other state-of-the-art techniques for detecting malicious JavaScript files?

2

Background

This chapter will go through the theoretical background this thesis encompasses. Beginning with a summary of JavaScript malware, then locality sensitive hashing methods, and lastly artificial neural networks, which is the machine learning method this thesis utilizes.

2.1 JavaScript malware

Malicious software or *malware* is a term that describes software with intent to harm/intrude/spy on systems or people. It is not uncommon for malware to also be self-replicating, automatically distributing the attacks among multiple targets (hence malware commonly being known as *Computer viruses*).

There exist numerous attack vectors for introducing malware to victims, e.g. e-mail attachments, shellcode etc. This thesis focuses on one attack vector, and that is *JavaScript* files. Almost all web pages today utilizes JavaScript in some form, whether to display fancy animations or to send data to web servers. These new services have thus become common attack vectors, as browsers often run these script files automatically when loading a website.

There exist numerous types of malware and a defined terminology of the most common JavaScript malware can be found in table 2.1. Since JavaScript is an interpreted language, the code does not go through a compiler that optimizes the code to byte code, in a way normalizing it. Instead, it is always represented precisely as it was written, this becomes a problem when malicious actors add obfuscation to the code to make it harder to analyse and detect. Obfuscation is a problem covered in the next section.

Table 2.1: Basic terminology for JavaScript malware

Category	Description
Adware	Malware that serves the victim unwanted advertisements.
Cryptominer	Malware that mines cryptocurrencies like Monero, Bitcoin and others. Can sometimes be used legitimately, so it is in some cases classified as a potentially unwanted application. The most popular JS based crypto miner is CoinHive.
Downloader	Overall family of malware that executes drive-by-download attacks. Where it tries to download malware on to the victim's computer. The malware type can be anything, e.g. Ransomware, Trojans, Adware and more.
Faceliker	Malware that manipulates "Likes" on Facebook.
FakejQuery	Malware that tries to disguise themselves as the popular jQuery library. Is often a downloader type malware.
IFrame	Malware that injects code into HTML sites using the `<iframe>` tag.
Ransomware	A family of malware downloaders that tries to install ransomware. Ransomeware encrypts and locks the victim's computer and then demands a ransom to open it again. Common ransomware is LOCKY, Crypted, and Ramnit.
REDIR	Malware that tries to redirect the victim on to a malicious website.
SEOHide	JavaScript trojan that spies on its victim.

2.2 Obfuscation

Obfuscation in JavaScript is a large problem in malware identification as it is not necessarily an indicator of maliciousness. Companies often use JavaScript obfuscation to protect their IP, hiding client-side logic or just want to keep file sizes at a minimum. Thus obfuscation techniques are now often used on benign JavaScript files. An example of this can be seen in figures 2.1 and 2.2.

```
function T(e, t) {
  if (e) {
    var i;
    if (!_$af10650742) {
      _$af10650742 = 0;
      return
    };
    for (i = e.length - 1;
         e > -1 && (!e[i] || !t(e[i], i, e));
         i -= 1) {
      ;
    }}}
if (_$af10650103 == true) {
  if (_$af10650364 === true) {
    _$af10650364(null);
    _$af10650364 = null;
    return
  };
  return
};
if (_$af10650742 == 1) {
  _$af10650742();
  _$af10650742 = false
};

function t(e, t) {
  return fa.call(e, t)
}

function n(e, i) {
  return t(e, i) && e[i]
}
```

Figure 2.1: Obfuscated code snippet from malicious code

```
function p() {
    if (!u) {
        var n = s(g);
        u = !0;
        for (var e = d.length; e; ) {
            for (l = d,
            d = []; ++c < e; )
                l && l[c].run();
            c = -1,
            e = d.length
        }
        l = null,
        u = !1,


    function(n) {
        if (i === clearTimeout)
            return clearTimeout(n);
        if ((i === a || !i) && clearTimeout)
            return i = clearTimeout,
            clearTimeout(n);
        try {
            i(n)
        } catch (e) {
            try {
                return i.call(null, n)
            } catch (e) {
                return i.call(this, n)
            }
        }
    }(n)
}
```

Figure 2.2: Obfuscated code snippet from CNN.com

These kinds of obfuscation techniques have become the standard when deploying code as can be seen by the widely used tools *webpack*, *uglifyJS*, and more. Other obfuscation techniques entail encrypting the code using various encryption methods and trying to hide function calls by encapsulating them in an `eval` method and more.

2.3 JavaScript classification

There are two major ways of doing JavaScript malware classification, through *static analysis* and through *dynamic analysis* [14, p. 862].

2.3.1 Static analysis

The main theory behind static analysis is to not execute any of the code given within the script. There are numerous ways to do static analysis, for instance, searching for special keywords such as specific function calls or destinations. Another method could be *bag of words* together with statistical learning. A clear benefit of using static analysis is the fact that the analysis does not run the potentially malicious code, protecting the system running the analysis. Static analysis might also be faster since it is easier to batch the work, and in dynamic analysis, one might need to wait for certain events to occur.

The main issue with static analysis methods for malicious JavaScript classification is that they are often rendered unusable on well-obfuscated code, where the compromising keywords might be hidden within the code.

2.3.2 Dynamic analysis

What characterizes dynamic analysis is the fact that the code is executed during analysis. A common way of doing this, to protect the analysts' system as well as gaining more control over the process is to run the targeted script inside of a sandbox or virtual environment. For instance, using a sandbox we run the script while monitoring the functions it tries to run, then if a specified security policy/rule is triggered, e.g. the script should not do an unprompted file download, the analysis will classify the file as malicious and terminate it.

The other way of doing a dynamic analysis that does not involve using sandbox environments is to use emulators. Emulators only try to emulate the results from system calls that a running browser (or operating system) would send for different function calls. Emulation has the benefits of being less resource intensive and more customized than sandboxes, which are virtual machines running entire operating systems. The drawback is that more development time is needed to write functions that emulate the various system/browser calls that a JavaScript file makes. Both of these dynamic methods requires domain expertise as it is necessary to craft security policies that only malware trigger, or in-depth knowledge of the running environment when creating an emulator.

2.4 Locality sensitive hashing

Locality Sensitive Hashing (LSH) is a relatively new family of dimensionality reducing algorithms. This family of algorithms focuses on producing condensed representations of the given input data which can later be used for comparison. This means in practice that using LSH methods on almost identical files will output almost identical hashes. Comparing this to cryptographic hashing techniques like SHA256 where hashing two almost identical files will yield two drastically different hashes that are not comparable to each other.

There are a few methods for producing these representations or "hashes" and the methods used in this project are described in the following subsections.

2.4.1 Nilsisma

One of the older locality sensitive hashing methods is Nilsimsa. The method has been demonstrated as useful in the area of spam detection, which has been presented in the paper *An Open Digest-based Technique for Spam Detection* by *Damiani et al*[4]. The algorithm's original publication is as of 2019 unavailable. However, there is a description of the algorithms inner workings made by *Damiani et al.*

The algorithm works by traversing the byte strings of the input data and pairing neighbouring characters into triples. These triples are then hashed with the *tran53* algorithm which produces a key determining the locality that the triple belonged to. The *tran53* algorithm returns a value in the range of 0 and 255. All of the localities found are then counted, and a vector is produced containing the amount of each specific locality found. By mapping the vector to an array with positive bits where the value is larger than the median, a 32-byte code can be produced.

The final hash produced is a simple 32-byte code which in base64 form consists of 64 characters.

2.4.2 TLSH

TLSH (Trend Micro Locality Sensitive Hash) is an LSH algorithm by *Oliver et al.* from Trend Micro, first described in the paper *TLSH - A Locality Sensitive Hash*[18]. This LSH method was designed for malware detection and clustering. The hash is constructed by digesting a byte string using a sliding window of size five and then mapping the corresponding values to buckets using the *Pearson hash*, a fast non-cryptographic hashing function. Next, the digest body which gets represented as a hexadecimal string is constructed from the bucket array by splitting it into different quartiles q_1, q_2, and q_3 depending on the bucket counts. The first three bytes of

the hash is the digest header. The final hash is thus represented as a 70 character hexadecimal string.

In essence, this LSH method is similar to the Nilsimsa hashing method, where the significant differences are in how the final hash is constructed from the bucket/feature vector. In TLSH features are mapped after being divided into quartiles depending on bucket occurrences, and in Nilsimsa they are mapped based on whether they are larger than the median of the feature vector.

2.4.3 ssdeep

Ssdeep is an LSH algorithm (or as the author calls it, a context triggered piecewise hashing algorithm) by *Jesse Kornblum*, first described in the paper *Identifying almost identical files using context triggered piecewise hashing* [17]. Ssdeep is a continuation from the spamsum algorithm by *Andrew Tridgell* which was designed for anti-spam purposes. Ssdeep has become the de facto LSH algorithm in use for malware detection and is the only locality sensitive hash supported by the industry-leading malware analysis repository VirusTotal[12].

The ssdeep hash is constructed using two hashing methods, one which is *piecewise hashing* that is just an arbitrary hashing algorithm but is used on small sections of byte string instead of the entire file. So, for example, a hash is made of the first 256 bytes, and then another for the next 256, and so on. The hashing algorithm ssdeep uses for hashing is the *Fowler-Noll-Vo* hash function by *Fowler et al.*[5], it was primarily chosen due to it being a non-cryptographic hash function with a focus on speed. The other hashing method is the *rolling hash*, which is a hashing algorithm that produces a pseudo-random value based on the current context of the input. The rolling hash is used to determine when a suitable range of bytes of piecewise hashing has occurred. When this *trigger* has gone off the accumulated piecewise hash is appended to a final hash. The final hash has the following form:

$$block\_size : hash_1 : hash_2$$

Block size is a calculated value that determines how large each block of code should be before appending the accumulated piecewise hash to the final hash.

As can be seen, there are two different hashes. This is due to there being *two* triggers for piecewise hashing. The first trigger is when the rolling hash produces a value that is equal, modulo the block size, to the block size minus one. The second trigger happens when the rolling hash produces a value that is equal, modulo twice to the block size or twice to the block size minus one. However, as the triggers are based on the block size of the file, the hashes do not have a fixed length, instead implementing a maximum length of the entire hash, which is 148 base64 encoded characters.

2.4.4 SDHASH

The algorithm SDHASH originally proposed by *Roussev* in the paper *Data fingerprinting with similarity digests*[20] is a newer LSH-method.

SDHASH uses a somewhat different system to represent the given input compared to the other methods in focus. It works by using a sliding window which traverses the input while traversing the program calculates a normalized Shannon entropy of each window to determine whether the contents of the window has any statistical significance. If it is deemed statistically significant at this stage, it assigns the window a feature number. After finding all of the features in the file, the program filters out all features that are deemed weak. Once all features have been filtered, the features are added to a bloom filter to avoid having duplicate features in the final results. The bloom filter is then used to construct the hexadecimal hash string.

In practice, the most significant difference between SDHASH and the other LSH algorithms is that SDHASH has no limit on the size of its hashes and that they vary drastically in size depending on the input as large files might output a hash that consists more than 10 thousand characters, which means that they are harder to use in neural network models which require fixed input sizes that are not too large.

2.5 Artificial Neural Networks

Artificial Neural Networks (*ANNs*)[7, p. 253] is a family of machine learning methods inspired by how neural networks work in brains. Models based on neural networks have revolutionized the industry in the last decade in most aspects of data analysis. From image recognition to personal assistants, neural networks have become an integral part of many complicated systems that are used by the population.

Typically, ANNs are multiple interconnected layers of nodes (also often known as *neurons*) where the first layer is an input layer, and the last layer is the output layer which produces the final result for the given input. In between the input and output layers, there are *hidden layers*. In particular *deep learning* refers to ANNs that have two or more hidden layers. An example of an ANN can be seen in figure 2.3

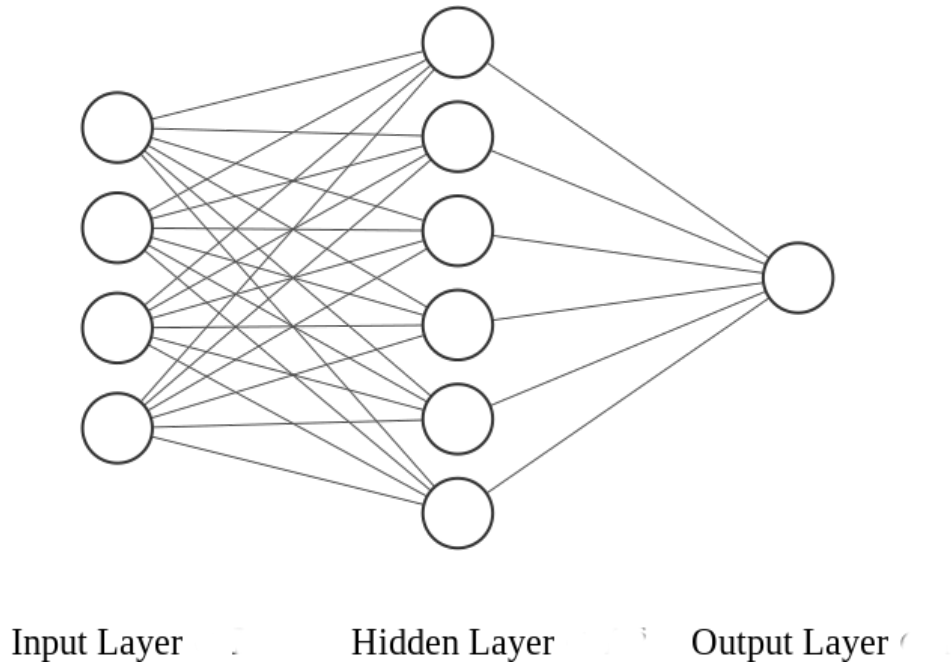


Figure 2.3: Example diagram of an ANN with one hidden layer

There exist many types of artificial neural networks, the book *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by *Aurélien Géron*[7] describes some of the more notable types:

- Feedforward Neural Networks (FNN)[7, p. 263]: These are the most standard neural networks like the one pictured in figure 2.3. They are used in classification and regression problems.
- Convolutional Neural Networks (CNN)[7, p. 353]: Most known for solving image recognition problems or other problems that use high dimensional data.
- Recurrent Neural Networks (RNN)[7, p. 379]: Models that excel at Sequence and Time-based data. A subcategory of RNNs is long short-term memory (LSTM) models, which are the driving force behind technologies like Google Assistant, Google Translate, and more. RNNs are usually applied in regression tasks but can be used to classify as well.

3

Related Work

As of now there are no other published works combining artificial neural networks with locality sensitive hashing methods as a form of feature extraction.

The field of malicious JavaScript classification has been active for a long time. However, when determining what kind of work can be deemed related to this thesis, a couple of different things is taken into consideration. The method preferably has to evaluate static JavaScript code, and it should show consistent results. The metrics used in these publications rarely focus on precision and recall. Instead the focus is mostly on false positives and to a lesser extent, false negatives. According to Cyren this is due to false positives being deemed the worst kind of classification errors made in the anti-malware industry.

The most promising competitor comes from a project made by *Curtsinger et al.* (Microsoft Research), which is a mostly static JavaScript classifier. The static part of the project is called ZOZZLE [3] and utilises a form of automated feature selection and a Bayesian classifier to detect malware and the dynamic part is called NOZZLE which deobfuscates JavaScript code before ZOZZLE can classify it. This project seems to be the closest classifier to our classifier as it is purely static with the only restriction being that the code has to be unobfuscated (a restriction our model does not have). One noteworthy thing with the paper is that the dataset used by them for training and testing is rather small and given that they are not focusing on one family of malware, as with a larger more diverse dataset their results might be different. Another noteworthy thing is the age of the paper, it was published in 2010, and a lot has changed in the industry since then. It is just in the last few years that widespread use of obfuscation on benign files has become the norm.

In the table 5.3 we can see performance indicators given by ZOZZLE. Two performance indicators are provided, one where they hand-picked the features based on domain expertise, and then an automatically picked feature set which focuses on minimising the false positive rate.

JStill by *Xu et al.* [21] is a project that focuses on statically detecting obfuscated malicious JavaScript code. The tool tries to identify obfuscation and then by looking at some key aspects, determine whether the script is malicious or not. A generous dataset has consisting of 50k clean files selected from Alexas top sites [1], and 30k

obfuscated malware was evaluated. A potential problem with this might be that the method is limited only to detect obfuscated code.

CUJO by *Rieck et al.* [19] is an automatic detection system for drive-by-downloads (defined as *Downloader* in Table 2.1). The system combines static and dynamic analysis to generate features which then can be used to detect malware with the help of support vector machines. Some of the problems we notice with the seemingly great results where the quantity of malware's is meagre, which might indicate that the results presented could be somewhat optimistic. As CUJO only focuses on *Downloader* malware, it might not be a good counter to this thesis' classifier, which is a general classifier.

JSAND/Wepawet by *Cova et al.* [2] is a hybrid classifier that detects malicious JavaScript code belonging to the *Downloader* family. Their feature extraction results in 10 different features where 3 of them are achieved through static analysis and the rest from dynamic analysis (through browser emulation). They are thus working on both obfuscated and deobfuscated code. Then the features are put into four models belonging to the libAnomaly library and their results aggregated. The results they got is impressive. However, their datasets might be a bit skewed as their known malicious dataset was tiny at 823 samples compared to 11.2 thousand benign samples. However, they then claimed to have a 0.01% false positive rate on another benign dataset consisting of 115 thousand benign websites. This service was once hosted openly as *Wepawet* but has since been acquired by Lastline defense and become proprietary/private.

Wang et al. published a paper in 2016 called "A deep learning approach for detecting malicious JavaScript code". In this paper, a couple of different machine learning techniques were evaluated on a large test dataset and their ability to detect malicious code recorded. The models using an RBF SVM or an ADTree method proved to yield the most promising results and is used for comparisons. The dataset used for training here consists of 12k benign files fetched from Alexas top websites [1] and 14k malicious files fetched by a web crawler named Heritrix [8].

4

Research Methodology

This chapter describes the actual experiment, what will be measured, and what research questions this thesis aims to answer.

The research approach in this thesis is a quantitative analysis where the data is gathered from an experiment and the results evaluated by comparing them to results from related works.

4.1 Research questions

The research questions are based on the problem formulated by Cyren and what might benefit the fields of malware research and program analysis.

RQ 1: Do neural network models with locality sensitive hashes as input features yield high accuracy along with low false positive and false negative rates when classifying JavaScript malware?

RQ 2: How well do different LSH methods compare with each other as input into neural networks?

RQ 3: How well do LSH based neural network models compare to other techniques for detecting malicious JavaScript files?

4.2 System structure

This section goes into how we combine the LSH methods and neural network models as described in the background section. To the best of our knowledge of academic literature, we are the first to combine ANNs and LSH methods in this way.

We take a locality sensitive hash and treat it as input to our neural network. To do this we split the hash into character n-grams which then gets tokenized. The tokenization process assigns each different permutation of the n-grams a corresponding integer token. This leads to the hash being represented as a sequence of integer tokens. The input layer of the feedforward network is of a fixed size, and the hashes that do not always have fixed length sequences get 0-padded so that the input size stays fixed.

The reason for splitting the hash into n-grams is to find correlations between sub hashes. Using probabilistic n-gram models to solve natural language problems is common and have also been used in source code analysis models as can be seen in the paper *Naturalness of Software*[9] by *Hindle et al.*

Figure 4.1 shows an example of how one locality sensitive hash traverses throughout our system.

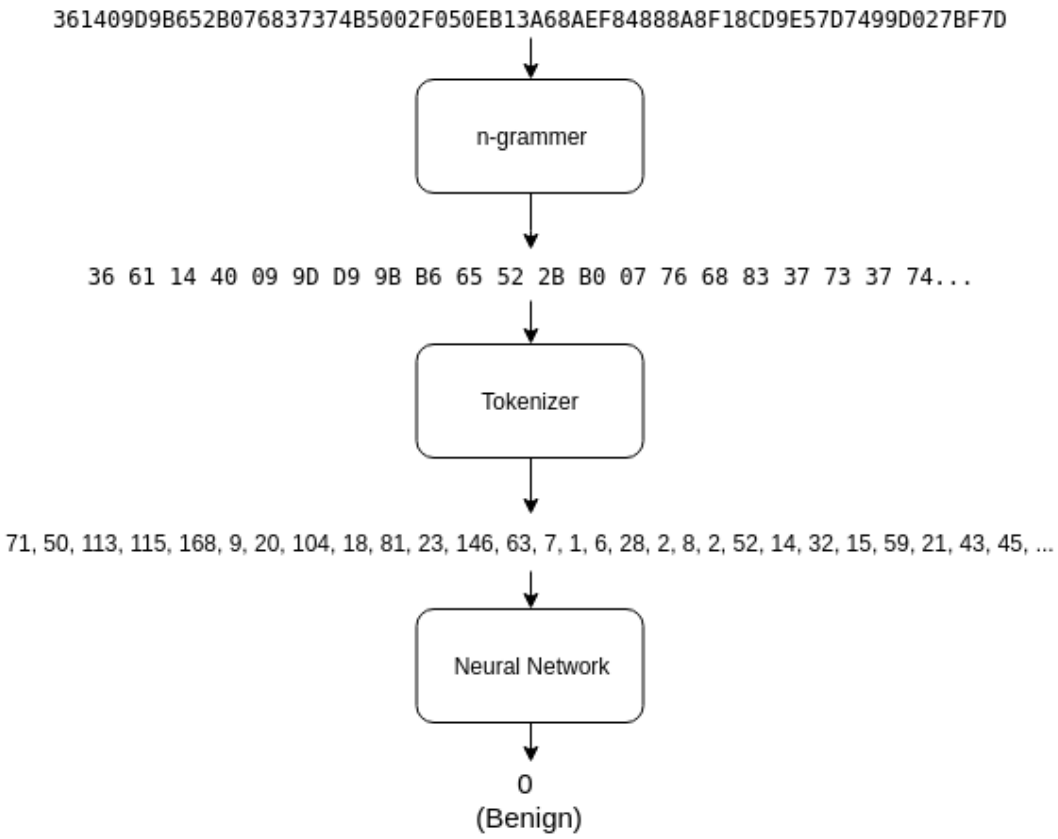


Figure 4.1: Example on how to classify a TLSH hash

4.3 Neural network design and implementation

A supervised learning approach with a normal deep feedforward neural network is used to classify each locality sensitive hash, as the dimensionality of the hashes is pretty low and they have accompanying labels indicating maliciousness. The input layer takes the tokens generated from each hash, and the output layer will return one single value, presented on a scale between 0 and 1, determining the likelihood of the input is malicious. The final network structure can be found in figure 4.2 or in the table 4.1.

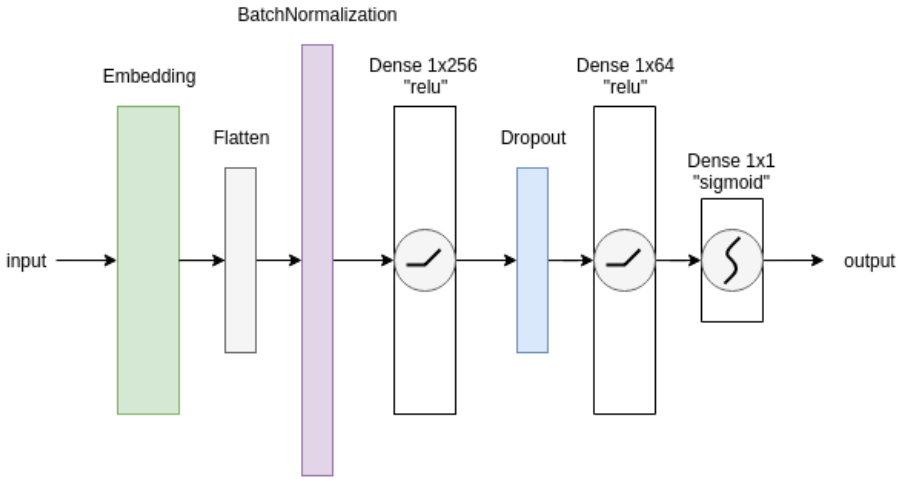


Figure 4.2: Abstract view over our network

Table 4.1: Neural network composition (where L is length of input vector)

Layer name	**Output dimensions**	
Embeddings	32xL	
Flatten	1xL	
BatchNormalization	1xL	
Dense	1x256	activation: relu
Dropout	1x256	magnitude: 0.125
Dense	1x64	activation: relu
Dense	1x1	activation: sigmoid

We chose to have an *embedding layer*[6] which is a layer that turns positive integers into dense non-zero vectors e.g. $[0] \rightarrow [[-0.2, 0.6]]$. The reasoning for this is that the input vectors can, in some cases, become mostly zero, especially in the case of ssdeep that has a varying hash length, leading to a lot of 0-padding. Neural networks can train on mostly zero vectors (also known as sparse vectors), but it is computationally inefficient.

When each layer of neurons propagates their input, an activation function is used to determine the layer's output. The *ReLU* function is used as the activation function, which is one of the most frequently used activation methods due it its simplicity, which can be seen in its definition in figure 2.5.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

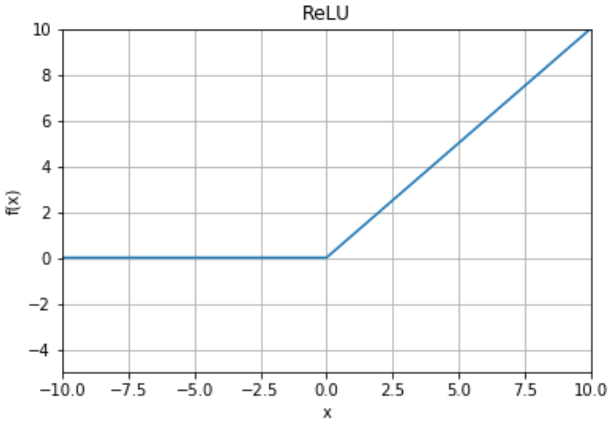


Figure 4.3: ReLU function and its plot

ReLU's properties make it so that during backpropagation the derivative of ReLU becomes

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Meaning that training networks utilizing ReLU activations are both computationally easier and faster than other more complicated activation functions.

However, since all negative numbers become zero immediately, there is a possibility that during training, the model will encounter the *vanishing gradient problem*, which might stop the network's training prematurely.

To combat this we utilize a *Batch Normalization* layer directly after our embedding layer[13]. Which normalizes each batch to combat both the vanishing gradient problem.

A *dropout layer*[10] is also used, this layer kills some of the previous layer's neurons during each training phase to prevent the model from overfitting (memorizing) the data. However, due to the batch normalization layer, we have a pretty low dropout rate of 12.5%, meaning that there is a 12.5% chance that a neuron is killed during a training step.

The last layer is a dense layer consisting of a single neuron with the sigmoid activation function. The sigmoid function is defined as the following:

$$f(x) = \frac{1}{1 + e^{-x}}$$

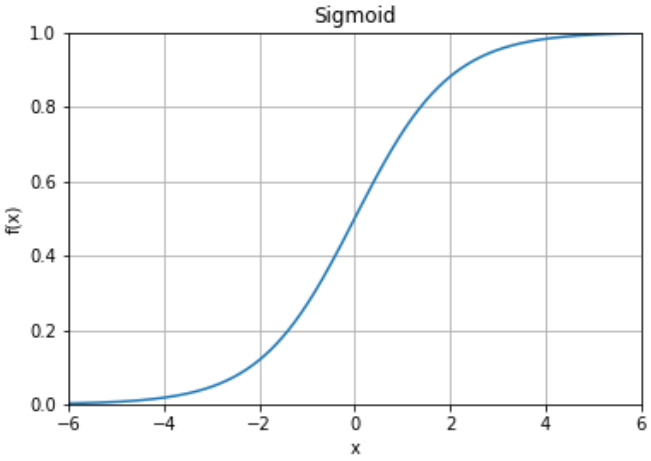


Figure 4.4: Sigmoid function and its plot

This means that the output of the sigmoid function is capped between 0 and 1. Thus the output is the probability of a sample belonging to the class label 1, or in this case, the probability of a sample is malicious. Using the sigmoid function is common for binary classification problems.

4.3.1 Network training process

There are numerous things to keep in mind during the neural network training process itself that influence the overall model performance, such as how to optimize the backpropagation, what batch size to use, and the number of epochs to train on.

4.3.1.1 Optimizers and loss function

In terms of fitting the network to gain an accurate understanding of the given data *adam optimization*[16] was used. Optimizing the network calls upon the procedure of determining how much the weights should be tuned given the resulting output-loss. Adam is an all-around good optimization technique which usually provides good performance[16].

Binary cross-entropy was used for calculating the output-loss given by each prediction, the function is expressed below where the variable p is the estimated probability of the target being 1, and y is the actual binary target.

$$loss(y, p) = -(y \cdot \log(p) + (1 - y) \cdot \log(1 - p))$$

4.3.1.2 Batch sizing

During neural network training, the dataset is often split into batches and sent through the network instead of sending the entire dataset through the network at once.

There has been much talk about what batch size leads to the best model. The smaller the batch size is, the longer it takes to train, meaning that there is a trade-off in model performance and model training speed. The consensus has been that larger batch sizes lead to less general models which were then further proven by *Keskar et al.* in their paper *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*[15] where larger batch sizes often leads to models converging at sharp minima early on.

Thus we have opted for a batch size of 512, which is a relatively small batch size but large enough that the training time on the largest datasets during the cross-validation experiment will be acceptable without converging too soon at a sharp minima.

4.3.1.3 Number of epochs

Another factor in training is the number of epochs to train the model on, where one epoch is when all the training samples have gone through a forward and backpropagation pass once.

Generally, the more epochs a model is trained on, the higher the risk is that the model overfits (memorizes) the data rather than generalizing the data. To combat overfitting, a validation based early stopping approach is used. By splitting 15% of the training set into a validation set, it is possible to monitor the changes in validation accuracy and validation loss for each epoch during training. Monitoring these metrics has the benefit of seeing that if the training loss is decreasing while the validation loss is not decreasing, then the model is starting to overfit and thus no need to further train the model. As the history of validation loss and accuracy is kept for each epoch. The weights of the epoch that yielded the lowest validation loss are restored for the model once early stopping has been applied.

4.3.2 Other ANN architectures

As was stated in section 2.5 there exist numerous types of ANN types and for each type there exist different architectures.

Many different architectures were considered, especially recurrent neural network (RNN) architectures as they tend to work better with sequential data, which the tokenized sequence of a locality sensitive hash is. A long short-term memory network [11], a popular RNN architecture, was tested early on but did not yield any better results than the deep feedforward network seen in figure 4.2 but instead took a lot longer time to train.

A one-dimensional convolutional neural network (CNN) was also considered but never tested, due to the high performance of the feedforward network and a lack of time.

We also considered and tested deeper feedforward networks but did not see any improvements compared to the final network depth.

4.4 SDHASH count vectorization

As was mentioned in section 2.4.4, the hashes generated from SDHASH has no maximum length, which is a drawback as our neural network model assumes a fixed input size. As a workaround, the input layer was changed from taking in sequences of tokens representing the hash to utilizing the *bag of words* method which identifies all tokens and counts them. This was done using the *Scikit-Learn* `CountVectorizer` pre-processor. A diagram illustrating this process can be seen in figure 4.5.

This means that the input is changed from a variable length sequence with no maximum size into a fixed array of size $vocabularySize^n$ where n is the type of n-gram used. Bigrams were used in this experiment and SDHASH is represented as hexadecimal strings, meaning the input array has size $16^2 = 256$, which is comparable to the input sizes of the other hashes.

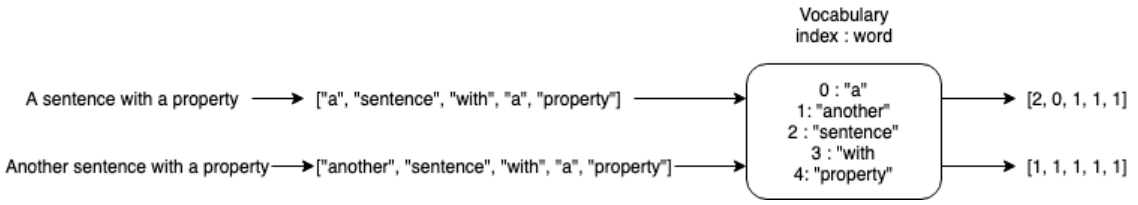


Figure 4.5: Count Vectorization example

Count vectorizing the input data results in loss of information as the ordering of tokens gets lost in the process, which might negatively affect the neural network model performance. Even though the occurrences are not explicitly counted in the non count vectorized models described in the subsections above, instead the occurrences become a property of the input implicitly.

4.5 Data selection

The data used in this thesis was provided by Cyren; they gave access to their pipeline of incoming files. The sources of these files are various, e.g. from web scrapers, customers specifically sending in files for analysis, e-mail attachments, incoming files from VirusTotal and more. Each of these files then goes through Cyren's malware scanners and the system applies a label to the sample indicating whether it is clean or malicious. These labels provided by Cyren will be the ground truth of the dataset and is generated from dynamic analysis.

During a period of four months (January-April 2019), this pipeline was monitored, and every time a JavaScript file went through the system, the file was fetched and hashed using ssdeep, Nilsimsa, TLSH, and SDHASH, along with the sample's detection name, and classification label. For replicability purposes a SHA256 fingerprint was taken of each sample to serve as a unique identifier for each sample, SHA256 was chosen as it is a common identifier used in the anti-malware industry.

As the neural network model uses binary cross-entropy as the error function, it is important to try to have a dataset that is as evenly distributed as possible in its class labels to prevent the model from getting a bias. If the dataset is uneven, there is a chance that the model prioritizes the majority class over the other as it results in a lower overall error during training, which results in the model having a bias towards the majority label.

Thus to ensure similar distribution of clean and malicious files the distribution was closely monitored, if the distribution became skewed towards one label, it was possible to correct the distribution by focusing the pipeline listener to the minority label until the distribution became roughly equal again. This approach yielded a dataset size of 1.527.177 samples, where the distribution between malicious and benign files is 53.6% malicious and 46.4% benign.

4.6 Experiment setup

The experimental setup consists of the following: At first, data was retrieved from Cyren as described in section 4.5. These hashes are stored within a CSV document together with corresponding labels. Via sampling, we derive seven subsets of the following sizes: 5k, 10k, 50k, 100k, 500k, 1M, 1.5M. These splits are to ensure that all of the experiments will be conducted on the same subsets of data.

For each subset and each LSH method, we run a 5-fold cross-validation experiment, where the performance indicators described in section 4.7 are recorded and then averaged over the five folds. Along with the performance metrics the false positives and false negatives were stored, for false positives, the LSH method, the SHA256 fingerprint and the subset it belonged to were recorded, for false negatives, the same properties were recorded along with the sample's detection name, as false negative samples are malware.

4.7 Metrics

In order to compare the different prediction models, we rely on three performance indicators: accuracy (ACC), false positive rate (FPR), and false negative rate (FNR).

The false positive rate is of particular importance since it will demonstrate the methods of actual usability in a malware identification environment, where it is important not to classify clean files as malware.

To calculate the false positive rate, we use the following formula based on the count of false positives (FP) and the count of true negatives (TN):

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

To calculate false negatives rate, we use the following formula based on the count of false negatives (FN) and count of true positives (TP):

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

Along with this, we also calculate the accuracy (ACC) this is calculated by dividing the number of the correctly predicted values with the total:

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

Classification metrics are not the only things considered. Practical metrics like measuring how large of a dataset is needed to train an acceptable classifier is also taken into consideration.

5

Results

This chapter will present the results of our cross-validation experiments. The results gained from this experiment can be found in table 5.1 in addition to figure 5.1,

5.1 Detection rates

Table 5.1 shows the results from our cross-validation experiment according to the experiment setup mentioned in 4.6.

Observing the results stated in figure 5.1 and table 5.1 it is possible to see that from the smallest dataset of 5000 is that the neural network model is highly capable of finding patterns in the hashes as it yields more than 90% mean accuracy for Nilsimsa, ssdeep, and TLSH. SDHASH has a higher dataset requirement producing stable comparable models around the 50k dataset size. The trend continues through all the dataset sizes, and it shows that false positive rates and false negative rates decrease as the dataset increases, although with diminishing returns as can be seen in the difference between the 1M and 1.5M datasets.

Interestingly the SDHASH model which used a count-vectorized style of network input seems also to produce good results, though falling short against the other LSH methods.

The results show that the neural network model is more prone to making false negative predictions rather than false positives, which is a positive trait in the world of malware detection.

It is also possible to detect a pattern of Nilsimsa having a slight advantage in terms of mean accuracy compared to the other methods. The difference between the different LSH-methods' false positive and false negative rates decreases as the dataset grows.

Table 5.1: Results from 5-fold cross-validation experiment

LSH TYPE	ACC (%)	FPR (%)	FNR (%)	Num Samples
TLSH	93.12	3.53	9.73	5k
Nilsimsa	93.72	3.64	8.56	
ssdeep	91.52	3.44	12.75	
SDHASH	69.52	60.18	3.68	
TLSH	93.15	3.48	9.77	10k
Nilsimsa	94.03	3.36	8.25	
ssdeep	91.37	2.15	14.28	
SDHASH	78.77	36.12	7.69	
TLSH	95.42	2.01	6.1	50k
Nilsimsa	95.95	1.45	6.30	
ssdeep	94.62	1.67	8.59	
SDHASH	93.12	3.88	9.51	
TLSH	96.18	1.42	5.88	100k
Nilsimsa	96.49	1.14	5.55	
ssdeep	95.65	0.60	7.56	
SDHASH	93.31	4.28	8.78	
TLSH	96.80	1.26	4.87	250k
Nilsimsa	97.09	1.09	4.46	
ssdeep	96.59	0.63	5.79	
SDHASH	94.11	2.65	8.69	
TLSH	97.14	1.19	4.30	500k
Nilsimsa	97.41	1.04	3.93	
ssdeep	97.21	1.00	4.34	
SDHASH	94.52	2.07	8.44	
TLSH	97.52	1.03	3.74	1M
Nilsimsa	97.78	1.09	3.17	
ssdeep	97.71	1.22	3.21	
SDHASH	95.17	2.16	7.14	
TLSH	97.79	1.01	3.25	1.5M
Nilsimsa	98.05	1.09	2.69	
ssdeep	97.97	0.94	2.98	
SDHASH	95.06	1.83	7.63	

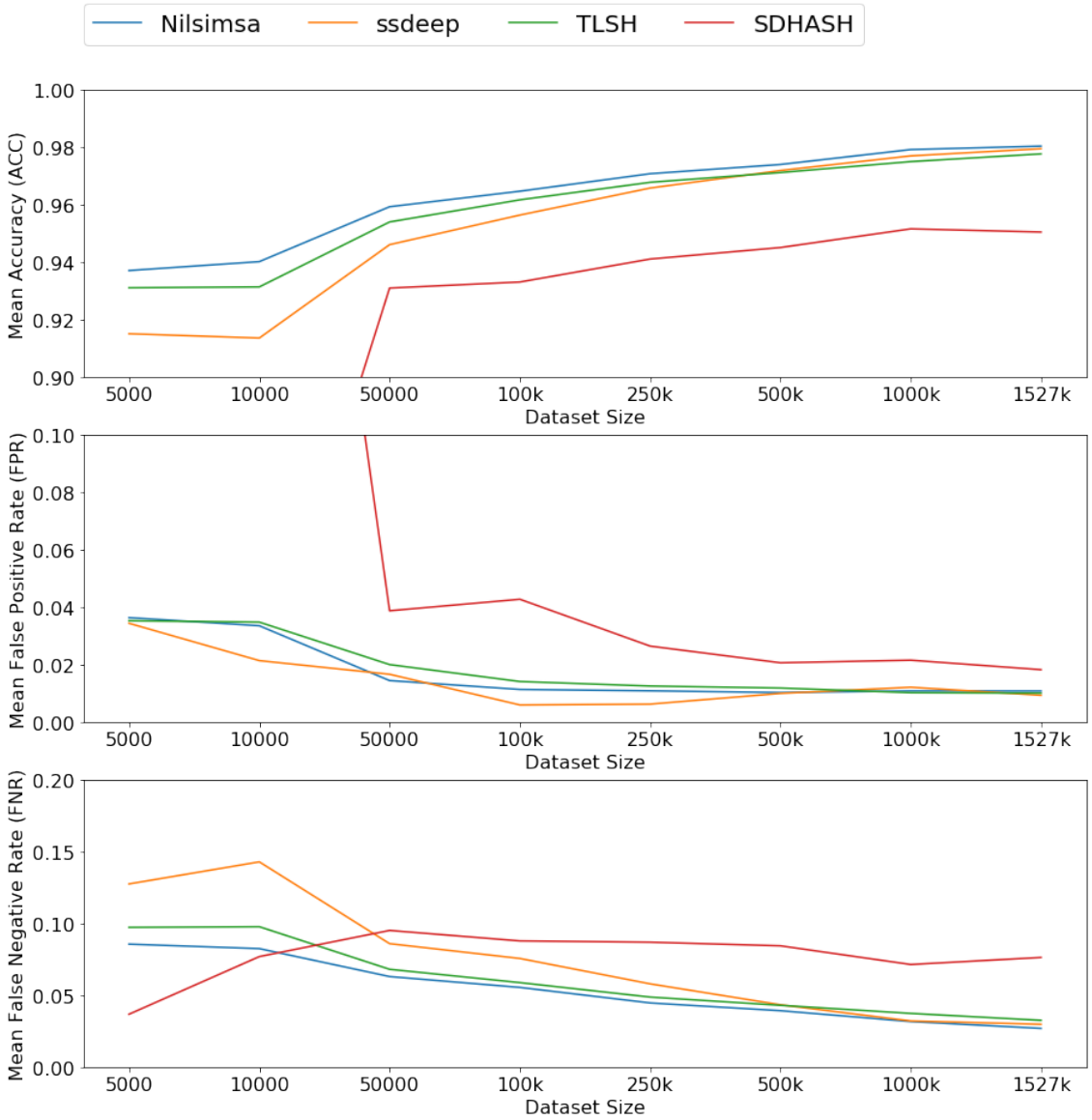


Figure 5.1: Results in relation to dataset size

5.2 Causes of misclassifications

During the experiment, all false negatives and false positives were stored. For false positives, only the sample's SHA256 was stored, but for false negatives, the SHA256 and detection names were stored. This section will focus on the models trained on the entire 1.5M sample dataset, as these are the best performing models and also the dataset that contains all malware and clean files, giving a better view of the shortcomings of LSH even under the best circumstances.

5.2.1 False negatives

The results in table 5.2 were retrieved after aggregating the top 10 most common false negative categories.

Table 5.2: Top 10 most common false negative categories, ordered by percentage of occurences

TLSH	%	**Nilsimsa**	%	**ssdeep**	%	**SDHASH**	%
Unknown	60.09	Unknown	59.55	Unknown	60.13	Unknown	60.54
Redirect	29.15	Redirect	29.86	Redir	29.45	Redir	29.19
Trojan	6.36	Trojan	6.10	Trojan	6.14	Trojan	5.95
CoinHive	1.86	CoinHive	1.95	CoinHive	1.89	CoinHive	1.85
SEOHide	1.22	SEOHide	1.08	SEOHide	1.07	SEOHide	1.12
IFrame	0.46	IFrame	0.49	IFrame	0.47	IFrame	0.43
Faceliker	0.23	Faceliker	0.27	Faceliker	0.22	Faceliker	0.31
Crypted	0.22	FakejQuery	0.29	FakejQuery	0.21	Crypted	0.23
FakejQuery	0.20	Crypted	0.24	Ramnit	0.21	FakejQuery	0.22
Ramnit	0.20	Ramnit	0.19	Crypted	0.20	Ramnit	0.15

The detection names here come from Cyren's labelling system, the most occurring false negative category is also the most difficult category to generalise, as according to Cyren they are *no names* or *unknown*, these are files that got flagged for malicious behaviour but there was not enough information along with the identification to sort the files into one of the more known malware families.

We can see a general weakness in the LSH approach here as all LSH methods lead to almost the same false negatives, where the top 7 misclassified categories for all four LSH methods are the same.

When inspecting these files, it can be seen that they have two common elements, either they are very similar to clean looking code, like the cases for Redirectors and FakejQuery, or the actual malicious part of the code is very small, making it easy to inject into otherwise clean code, like CoinHive.

The consequences of these factors are that during LSH hashing it might lead to malicious information getting lost, e.g. if there is a single line of malicious code in an otherwise clean file it might result in that the hash looks more like a clean file rather than a malicious file, which is often the case with CoinHive or other cryptocurrency mining malware. An example of how one can use CoinHive can be seen in figure 5.2.

```
var miner = new CoinHive.Anonymous(<wallet_address>); miner.start()
```

Figure 5.2: CoinHive example

In the case of redirectors, we have malware that is not necessarily doing anything malicious, as redirecting users on websites is a very common thing, but it is the destination that is malicious. This is a similar problem with downloader malware which in figure 5.2 are Trojan, Ramnit, FakejQuery, and Crypted, the act of downloading is not malicious, but the file it downloads is malicious which can be hard to detect. So often in these cases, the destination/download URLs are the malicious indicators which might get lost during locality sensitive hashing.

So when creating a locality sensitive hash, the hash might end up looking like other downloader/redirection programs that are benign.

5.2.2 False positives

False positives are a much harder thing to find a pattern in, and since false positives should be clean files, they do not carry a detection/category name to use as a basis for generalization. Another issue is that clean files were not explicitly stored during the data collection process as they are more likely to contain personally identifiable information; this fact makes looking up false positives for further diagnosis harder.

Thus we had to do the following the false positive analysis. We randomly chose 50 SHA256 signatures from each LSH method's false positive list, totaling in 200 samples. As the samples were not stored locally, we used VirusTotal as other anti-malware vendors might have classified the files differently than Cyren. Out of the 200 SHA256 fingerprints, 50 of them were found on VirusTotal. By inspecting said files, the following can be said on why some false positives happen:

- Due to malware like FakejQuery, there is a chance that other similar benign code gets detected, e.g., code that is a fork of jQuery or a jQuery plugin.
- Shorter files give hashes that carry less information, leading to higher false positives. This is the reason for that all four LSH methods has a recommended minimum file size before hashing.

- Some obfuscation techniques are less common than others, for example, encoding JavaScript statements as a string which the program then interprets using the `eval` method is highly suspicious but is not always an indicator of maliciousness. It is sometimes used to hide sensitive data that should not be able to get scraped by web crawlers.

5.3 Comparison with existing approaches

When comparing our LSH based models to the competitors, the major differences are: our classifiers do not rely on any dynamic analysis, which most competitors use in conjunction with static analysis except for Zozzle, Cujo static, RBF SVM and ADTree, which are the only fully static competitors. In addition to this, Zozzle only works on unobfuscated code, if the classifier encounters obfuscated JavaScript it has to run a companion program called Nozzle which deobfuscates the JavaScript file before Zozzle can classify it. This is not the case for our LSH based models as it does not need any extra pre-processing if the file is obfuscated.

In other cases where the classifiers are based on dynamic analysis, they have some performance limitations where each script has to be run, and then the classifiers need to wait for certain events to occur before being able to classify the input file.

Making classifiers with small datasets will lead to less generic models. Since there exists a vast diversity of possible malware and clean files, a small dataset might give a skewed image of the performances of the methods, due to not being able to verify whether it works on new never-seen-before malware. This makes it harder for us to for trust the results stated in table 5.3 of the Zozzle model or the JSAND/Wepawet model. In our case, we have 1.5M samples with a 54/46 distribution, the only competitor that had a similar dataset size was Cujo with roughly 201k samples, but the distribution was 0.3/99.7. This can be further seen in 5.3 as the best performing classifiers there all have very skewed malware/clean ratios and the classifiers that do have even distributions tend to perform much worse.

To summarize our models perform better than most competing classifiers and comparable with the very best classifier without the drawbacks they have. Another thing is due to the sheer size of our dataset, we can reliably enforce the validity of our models and that it should manage a similar performance when used in real life circumstances.

Table 5.3: Performance indicators gathered from related works and our best models

Classifier	ACC (%)	FPR (%)	FNR (%)	Malware : Clean
Zozzle hand picked	98.2	1.50	1.20	900:8000
Zozzle automatically picked	99.2	0.30	9.20	900:8000
JStill	97.3	17.5	0.53	30k:50k
JSAND/Wepawet	98.3	0.00	0.20	823:11.2k
RBF SVM	86.8	4.92	8.33	14k:12k
ADTree	82.7	2.42	14.92	14k:12k
CUJO static	90.1	0.10	9.80	609:200k
CUJO dynamic	85.9	0.10	14.00	609:200k
CUJO combined	93.8	0.20	6.00	609:200k
Ours - TLSH	97.79	1.01	3.25	818k:709k
Ours - Nilsimsa	98.05	1.09	2.69	818k:709k
Ours - ssdeep	97.97	0.94	2.98	818k:709k
Ours - SDHASH	95.06	1.83	7.63	818k:709k

5.4 Answering the research questions

This thesis has three research questions which can now be answered:

5.4.1 Research question 1

The first question was *Do neural network models with locality sensitive hashes as input features yield high accuracy along with low false positive and false negative rates when classifying JavaScript malware?*

The results in table 5.1 shows that all LSH methods except for SDHASH show high accuracy and low FPR and FNR. Starting at the smallest dataset of 5000 samples, whereas SDHASH needs 50k samples to find a learnable pattern.

For TLSH, ssdeep, and Nilsimsa the FPR for all of them is always below 5% starting immediately at 5000 samples and decreases from there on to a FPR of 0.94%. Similarly can be said of their FNR capabilities whereas they start at 13% and decreases to a lowest FNR of 2.69%.

SDHASH perfoms worse than the other LSH methods, with 60.18% FPR when trained on 5000 samples, but achieves 1.83% when trained on 1.5M samples. However, coincidental with gaining an above 90% ACC achieves a maximum FNR of 9.51% at 50k samples which then improves along with an increase of the dataset size yielding a minimum FNR of 7.63% at 1M samples.

So we are confident in saying that LSH based neural network models do yield high accuracy along with low false positive and false negative rates when classifying JavaScript malware.

5.4.2 Research question 2

The second research question was *How well do different LSH methods compare with each other as input into neural networks?*

As was discussed in the subsection above, TLSH, Nilsimsa and ssdeep all perform very similarly. With only SDHASH performing considerably worse than the rest.

The results in table 5.1 show that in general Nilsimsa makes the least tradeoffs between FPR and FNR, ssdeep has its strengths in a low FPR at the cost of a higher FNR, TLSH falls somewhere in the middle of ssdeep and Nilsimsa.

So to conclude is that TLSH, Nilsimsa, and ssdeep all perform similarly with SDHASH being the clear outlier. One can choose an appropriate LSH method depending on the situation: ssdeep if the focus is more on having a low FPR, Nilsimsa if the focus is on low FPR and low FNR or TLSH which performs similarly to Nilsimsa, but is faster at hashing than Nilsimsa.

Regarding SDHASH, it is not only that the classification performance is not as good, but with the fact that it has no maximum length of the generated hash, which means that it is both significantly slower to process the SDHASH hashes and it takes much much more memory to do so as well. Whereas the 1.5M sample dataset for TLSH, ssdeep and Nilsimsa all together was roughly 439MB, and the same dataset with SDHASH hashes was 6.6GB.

5.4.3 Research question 3

The third research question was *How well do LSH based neural network models compare to other techniques for detecting malicious JavaScript files?*

Many clear benefits in regards to the proposed method being static are discussed in previous sections. Given the circumstances of having a vast amount of computing power and time, a dynamic analysis might still be preferable if the absolute goal is to have as low of a FPR as possible as can be seen with the JSAND classifier.

With this in mind, we can see that that the best TLSH, Nilsimsa, and ssdeep models all perform better than all of the competitors in terms of mean accuracy, except for the Zozzle ones. The false positive rates of our models are also comparable to the competitors where the only lower FPRs came from the automatically picked Zozzle classifier and the CUJO classifiers, though in their cases they traded off low FPR with a high FNR which our models do not do to the same extents. The three models all had lower FNR than all competing models except for the hand-picked Zozzle model at 1.2% FNR and JSAND being a dynamic analysis tool at 0.2%.

Comparing the method to other static analysis methods, our method performs well being able to produce comparable results while being fully static without some of the drawbacks of the other competing models, such as working on obfuscated code.

5.5 Threats to validity

Given the results retrieved from this experiment being based on data from a single origin, the generalisation this method could be questioned. The JavaScript files processed by Cyren might not be comparable to the data used in other related works. The classification models proposed in this thesis might thus not be generalisable to all JavaScript documents but rather only those an anti-malware company might process.

Since the labels used are a product of Cyren's internal systems, there is guaranteed to be some false positives or negatives within the provided dataset. These mislabeled samples, given a low rate, might add some variance to the final results. Due to the sheer size of the dataset validating it using manual inspection is not feasible. This could entail that some false positives and negatives presented in the results are actually correct classifications. This could also mean that if the mislabeling of the data were systematic, larger groups of false positives or negatives would go undetected. Thus in a way, it can be said that our LSH based models might have a similar bias as Cyren's internal systems.

6

Conclusion

6.1 Effectivness of LSH and ML

The results presented in this report strongly suggests that it is possible to find a clear pattern with malicious script files when compressed into locality sensitive hashes. All of the different hashing techniques provides a good base for the neural network to learn and make accurate predictions.

There are many clear benefits with these results in terms of the method being static and relatively lightweight to run. A personal computer can batch predict thousands of hashes in just seconds of a measure, making it powerful regarding the real-time analysis. Since it is static analysis, the system does not have to execute any of the actual script code, making this method more secure in contrast to dynamic analysis.

Having access to a labelled dataset of the size used in this project is pretty rare and might result in it being hard to replicate to this scale. As presented in the results give we learned that this method does not require a staggering amount of labelled data to be predictive, around 5000 data points for three out of the four hashing methods. The model performance does increase with more data, however, with diminishing results.

Another benefit of using locality sensitive hashing with artificial neural networks is the short training time required to produce a model. Regardless of the LSH method used, training a model on a consumer grade graphics card (in our case an Nvidia RTX2080) usually takes less than 10 minutes.

6.2 Practical application

A core benefit of using a model similar to the methods proposed in this thesis is the speed and performance gains. Because of this method might be beneficial to use when classifying large amounts of data. The method might also be used in a first impression system being able to classify new samples in real time.

Since the best models all have low FPR they are also suitable to *stacking* which is widely done in the malware industry, which is stacking classifiers on top of each other, meaning that if the model is not prone to false positives but prone to false negatives, it can serve as a filter before other classifiers that are prone to false positives but has better false negatives.

Also, seeing that all LSH methods have similar performances, it is also possible to create an *ensemble classifier* that combines different models. In this case, one could have three classifiers, each one trained on a different LSH method. Then the predictions are averaged together and a final classification based on said average, which is called *soft voting*.

Our dataset comprised of malware that has been trending the last four months, meaning that if the trend suddenly changes, there is a risk that there is a large amount of new malware that the classifiers have never seen before. This problem can be mitigated by regularly retraining the models on datasets representing the current malware trend.

6.3 Further research

As has been shown, LSH algorithms used in malware identification perform well as input into neural networks. However, as was discussed in section 4.2, it is necessary to do some pre-processing on the hashes to make them viable as input in a neural network model. Creating a locality sensitive hashing algorithm that outputs a feature vector directly rather than a character string would perhaps yield even better results as it might be possible to include more information, if not then at least lower the amount of pre-processing needed to create the input as the LSH method would output something that could be used directly as input into a neural network model.

We have also shown that the performance indicators of our models all improve with more data (although with diminishing returns), meaning that there might still be room for improvement for the models which could be shown with further research, or perhaps that there is a ceiling on how much our model is able to learn on.

6.4 Closing remarks

In this thesis, we have shown that using locality sensitive hashes as input into neural networks is a strong and viable option in the world of JavaScript malware identification. We evaluated four LSH methods, Nilsimsa, TLSH, ssdeep, and SDHASH and showed that three of them perform comparably to each other with the exception of SDHASH being an outlier. We also explored the scalability of this method showing that the proposed method scales with the dataset size, though with diminishing returns.

In the end, the results of this thesis spawn a number of areas to explore in the field of using locality sensitive hashing with machine learning.

Bibliography

[1] Amazon. *Alexa's top sites*. 2019. URL: https://www.alexa.com/topsites (visited on 03/28/2019).

[2] Marco Cova, Christopher Kruegel, and Giovanni Vigna. "Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code". In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. Raleigh, North Carolina, USA: ACM, 2010, pp. 281–290. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772720. URL: http://doi.acm.org/10.1145/1772690.1772720.

[3] Charles Curtsinger et al. "Zozzle: Low-overhead Mostly Static JavaScript Malware Detection". In: (Nov. 2010). URL: https://www.microsoft.com/en-us/research/publication/zozzle-low-overhead-mostly-static-javascript-malware-detection/.

[4] Ernesto Damiani et al. "An Open Digest-based Technique for Spam Detection". In: vol. 2004. Jan. 2004, pp. 559–564.

[5] *FNV Hash*. 2013. URL: http://www.isthe.com/chongo/tech/comp/fnv/index.html (visited on 05/30/2019).

[6] Yarin Gal and Zoubin Ghahramani. "A theoretically grounded application of dropout in recurrent neural networks". In: *Advances in neural information processing systems*. 2016, pp. 1019–1027.

[7] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491962291, 9781491962299.

[8] *Heritrix*. 2019. URL: http://crawler.archive.org/index.html (visited on 03/28/2019).

[9] Abram Hindle et al. "On the Naturalness of Software". In: *Commun. ACM* 59.5 (Apr. 2016), pp. 122–131. ISSN: 0001-0782. DOI: 10.1145/2902362. URL: http://doi.acm.org/10.1145/2902362.

[10] Geoffrey E. Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580.

[11] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://doi.org/10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[12] https://www.virustotal.com. *VirusTotal*. 2018. URL: https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works (visited on 12/29/2018).

[13] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[14] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Static analysis for detecting taint-style vulnerabilities in web applications." In: *Journal of Computer Security* 18.5 (2010), pp. 861–907. ISSN: 0926227X. URL: http://proxy.lib.chalmers.se/login?url=http://search.ebscohost.com.proxy.lib.chalmers.se/login.aspx?direct=true&db=buh&AN=52929755&site=eds-live&scope=site.

[15] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *CoRR* abs/1609.04836 (2016). arXiv: 1609.04836. URL: http://arxiv.org/abs/1609.04836.

[16] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[17] Jesse Kornblum. *VirusTotal*. 2018. URL: https://ssdeep-project.github.io/ssdeep/index.html (visited on 01/01/2019).

[18] Trend Micro. *TLSH*. 2018. URL: https://github.com/trendmicro/tlsh (visited on 01/01/2019).

[19] Konrad Rieck, Tammo Krueger, and Andreas Dewald. "Cujo: Efficient Detection and Prevention of Drive-by-download Attacks". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. Austin, Texas, USA: ACM, 2010, pp. 31–39. ISBN: 978-1-4503-0133-6. DOI: 10.1145/1920261.1920267. URL: http://doi.acm.org/10.1145/1920261.1920267.

[20] Vassil Roussev. "DATA FINGERPRINTING WITH SIMILARITY DIGESTS". In: 2010, pp. 207–226.

[21] Wei Xu, Fangfang Zhang, and Sencun Zhu. "JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code". In: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*. CODASPY '13. San Antonio, Texas, USA: ACM, 2013, pp. 117–128. ISBN: 978-1-4503-1890-7. DOI: 10.1145/2435349.2435364. URL: http://doi.acm.org/10.1145/2435349.2435364.