Detecting client-side e-banking fraud using a heuristic model

T.Timmermans
tim.timmermans@os3.nl

J.Kloosterman
jurgen.kloosterman@os3.nl

University of Amsterdam
ABN AMRO Bank N.V.

July 11, 2013

Abstract

This research proposes and implements a heuristic model to detect client-side e-banking fraud caused by malware. Results show that the model is promising and is able to detect malicious injections from malware. To validate the developed model, an additional experiment is performed in which unknown web pages, adapted by recent malware are correctly classified based on historical, malicious pages of a bank. However, validation of the results with a more representative dataset is required. The classification process of a web page is performed with a mean of 0.176 seconds. Improvement of the developed model may lower impact on resources and execution time.

Supervisor:
Mark Wiggerman
mark.wiggerman@nl.abnamro.com

1 Introduction

For as long as online banking exists, criminal organisations are actively trying to commit fraud leveraging weaknesses in banking applications. Recent cases of fraud [1–3] have shown that even though the detection mechanisms have improved, malware continuously evolves to find weaknesses in banking applications that are exploited through infected clients.

Current malware infections include trojans, worms, viruses, ad- and spyware among others [3]. Uprising of trojans poses challenges as this type of man-in-the-browser (MitB) malware is able to dominate the web browser and is able to inject code in existing web pages to mask the original page and to hide its malicious behaviour to the end user.

One of the major problems with client-side security is that no control can be applied on the underlying platform. Especially when malware dominates the user's browser, every security measure implemented in the web page can be circumvented. Therefore, responses from the client can not be trusted. In addition to that, Man-in-the-Browser threats are very difficult to counter using solely web techniques such as Javascript, as these generally operate on a higher level and do not have access to the operating system.

Therefore, this research presents a server-sided model to detect malicious code on the client-side by validating the rendered web pages on a server. The analysis on the server consists of a classification with a set of heuristics, in order to detect malicious web pages that differ in their set of features and the probability that an unknown page has been modified by malware. To review the feasibility of the technique a proof of concept is created and the accuracy and execution time have been measured accordingly.

1.1 Research Question

For this research the main research question is:

To what extend is it possible to detect maliciously injected code into a web page using a heuristic model to counteract fraud, and what is the performance of such technique in terms of accuracy and execution time?

1.2 Project Scope

The scope of this research project is the design and implementation of a proof-of-concept application to detect e-banking fraud that poses a threat for customers of a bank. Every client-side defense mechanism put in place is vulnerable for interception and unintended manipulation. Thus, the decision is made to develop a server-sided model to collect the web pages that are susceptible to be used in an attempt to commit fraud. Although a central solution introduces additional security and performance problems by itself, mechanisms to detect fraud have previously not been able to determine if pages rendered on a client have been modified by malware.

In this research the web pages have already been obtained before they provide as an input for the model. A future implementation should provide a client-side mechanism to send web pages in an automatic way such that the response of a banking web site is not changed. The client-side solution required to aggregate the web pages in an automatic way is left out of the scope of this research. Section 3.3 however proposes a method for the client-side

implementation.

1.3 Outline

The paper is structured as follows. Existing studies related to the analysis and detection of malicious websites are described in section 2. The method used to detect fraud is detailed in section 3. The results are presented in section 4. Section 5 and 6 contain the discussion and a conclusion of the method and results. Finally, challenges for further work are described in section 7.

2 Related Work

In general, e-banking malware tends to inject malicious scripts and other elements into the web page of a bank. The work by [4] resulted in the development of a method to both analyse and detect malicious Javascript. A sandboxed environment based on the SpiderMonkey Javascript implementation was developed in order to deobfuscate, log and profile Javascript code. This solution can distinguish both benign and malicious code.

More research in this area is performed by [5]. The product of this research is a filter called *Prophiler*, which uses both static and dynamic syntax analysis techniques to examine millions of web pages for malicious content. The choice for separating static and dynamic syntax analysis was made to only put statically unclassified pages through dynamic analysis. Compared to a situation in which all web pages were put through dynamic syntax analysis, system load with the new filter is reduced by more than 85 percent.

Research performed by [6] developed a tool named *ZOZZLE*, which is a low-overhead solution for detecting and preventing JavaScript malware in the browser. It uses Bayesian classification and the experimental evaluation shows that the tool is able to detect JavaScript malware through static code analysis with a false positive rate of 0.0003%.

When comparing the outline of the research described in this paper and the referenced research, the data acquisition method is different. [5] is primarily of intent to apply research on drive-by-downloads and therefore needs to acquire as much data as possible from a list of unknown popular pages. For the research in this paper, it is essential to acquire both benign and malicious pages from a known collection of web pages in order to find erratic behaviour, but also to train a heuristic model accordingly. However, there is an overlap between drive-by-downloads [7] and the injections performed by e-banking malware. Finally, a subset of the static syntax analysis techniques applied on HTML web pages that have been described in [5] have been independently implemented in the extraction of features in this research.

3 Method

E-banking malware such as a trojan horse is able to inject code into the web pages a client requests from the bank's web site. Although the connection is secured to prevent man-in-the-middle attacks, it is not able to inspect how the web pages on the client are displayed. A Man-in-the-browser attack, in which malware hooks on browser events, is difficult to detect as almost anything on the client-side can be modified. In general the convention is that clients can not be trusted and that mechanisms to detect fraud are placed in a server-side architecture.

3.1 Overview

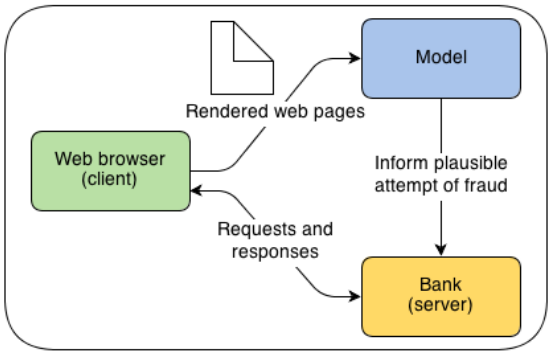

Figure 1: Overview of the model

Figure 1 describes a model to get insight whether a client is attempting to commit fraud based on the information present in the web page a browser has rendered. As an end user requests a page, the web browser will request that particular page and renders it upon retrieval by the render engine, e.g WebKit [1]. A concise comparison of web browser engines can be obtained from [2].

Given the characteristics of a browser engine a web page is displayed differently on popular web browsers such as Mozilla Firefox, Google Chrome and Internet Explorer. However, these minor differences are not the main problem, as a trojan horse can add additional elements to the received page before or while the page is actually shown to the user.

By obtaining the Document Object Model (DOM) [3] from a web browser, the DOM gives insight into how malware might have adapted a web page. If this data is aggregated and sent to a separate channel throughout the time a user is active in the e-banking environment, this data can be used as a starting point for further analysis.

3.2 Model

Comparable to the research by Canali, et al. [5], the method in this paper is also divided in several phases (see figure 2. The initial phase consists of the aggregation of data from the DOM for a set of clients. In the second phase, this set of data is analysed by extracting a selection of features that are stored in a central location. Third, the extracted features in the database are transformed to a vector of scores per web page using a preprocessor. With

[1] https://www.webkit.org/

[2] https://en.wikipedia.org/wiki/Comparison_of_web_browser_engines

[3] http://www.w3.org/DOM/

all the vectors as an input, the vectorised data is processed in a Bayesian classifier to compute a probability that a web page is either benign or malicious. Based on this score, the bank is informed that there are signs that a client is committing e-banking fraud.

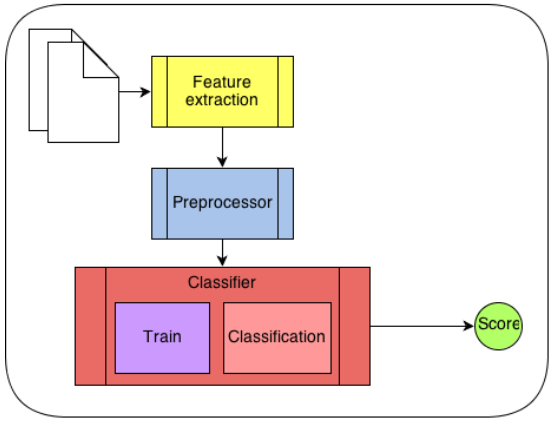

Figure 2: Model components

3.3 Aggregation of data

When a client requests a web page, the web browser requests that particular page. After receiving it, the page is rendered by the browser engine.

As the Document Object Model (DOM) from a web browser can be obtained using Javascript, the *document.documentElement.outerHTML* attribute [4] of the DOM contains a serialized HTML string of the rendered page. This attribute provides insight how malware modifies a web page. If this data is aggregated and sent to a separate channel throughout the time a user is active in the e-banking environment, this data can be used as a starting point for automated static analysis. As described in section 1.2, obtaining the data in an automatic way is left out of the project's scope. However, the next paragraph presents recommendations how this method can be implemented in a relatively secure manner.

Hiding techniques
Javascript can be used to send a web page, using non-blocking asynchronous requests, from a client over a secured channel (e.g HTTPS connection) to a system that aggregates these web pages for further analysis [8]. It is important to acknowledge that everything on the client-side can be manipulated. It is difficult to protect Javascript code, since it is not compiled into byte or binary code [9]. Therefore, a number of recommendations are presented how Javascript code can be obfuscated to make it difficult to tamper with.

Function Reordering Randomly reorder function declarations. Makes it harder to properly reference different parts of the code.

[4] `https://developer.mozilla.org/en-US/docs/Web/API/element.outerHTML`

Dead code Injection Randomly inject dead code, preferably derived from the source code. It can be isolated using opaque predicates.

Literal replacement Replace literals with randomly generated numbers of ternary operators.

String encoding Using different types of encoding to hide the plain text the source code.

The above recommendations are all part of the obfuscation process to make the source code harder to understand, even with the help of computing resources. The script should be placed in a random place in the HTML document upon each request. This makes it harder for attackers to strip the code from the page. Also, the source must be obfuscated with different parameters in order to serve multiple versions of the script. This is possible by using different encodings and using randomly injected code. Obfuscation techniques, algorithms and available tools are discussed in [9, 10].

3.4 Feature extraction

This component uses the input of captured web pages as a start to extract features. Both malicious and benign pages are analyzed to identify a representative set of features to construct a classification model. The features are divided into three categories: HTML, Javascript and URL features. The features are based on anomalies found during the analysis of the pages that distinguish benign from malicious pages.

The first category, HTML, contains features about the HyperText Markup Language (HTML) structure of the web page. The second category, Javascript, contains features based on Javascript snippets found in the HTML. The Javascript features focus on identifying malicious Javascript. The third category, URL, contains features for identifying malicious URLs that are found in the web page. Appendix A.1 shows the entire set of identified features and their descriptions.

The output of the feature extraction module is a *key value* list for each category. This output can be forwarded directly to the preprocessor or stored in a database.

3.5 Preprocessor

The preprocessor component transforms the output from the feature extraction into a vector array that is expected as input for the classifier. Besides the mapping of data, the preprocessor also contains important logic that is used to identify the maliciousness of URLs.

The features extracted from each URL are translated to a total *url score*. The scoring mechanism is based on a rule-based system that assigns points based on the outcome of a condition. For example, if the domain is outside the scope of the bank's website the score increases. Another example: an URL in the *src* attribute of a *script* tag does not contain a *.js* extension and has multiple parameters.

The URL's from the known benign pages are placed on a whitelist, since they are extracted from a trusted

source. Therefore, those URLs are skipped from the score assigning process in order to reduce processing time.

URL	Classification	Score
`https://www.abnamro.nl/nl/logon/`	benign	0
`https://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js`	malicious	30
`https://approvehost.net/nl/inl.php?un=LAB1_E532648AFE2B7396&bn=abnamro&st=undefined&get_drow=get`	malicious	50

Table 1: Example URLs and their assigned score using a rule-based approach

Table 1 shows three example URLs and their assigned score. The first URL resides within the scope of the bank's domain and is also listed on the whitelist, created during training, and therefore classified as benign. The second URL, a popular Javascript library, is injected by malware and used to modify elements on the page. Classified as malicious and assigned a score of 30. The last URL is also injected by malware and due to the used TLD and number of parameters it has a score of 50.

3.6 Classifier

The naïve Bayes classifier [11,12] is the model's core component. This is the process that identifies to which of a set of categories a web page belongs to. A Bayes probabilistic classifier is based on applying the Bayes theorem [13] that relies on the assumption that the features that were obtained from the preprocessor are independent.

There are two steps required in order to use the classifier: training and classification. Training is the process of taking content that is known to belong to specified classes and creating a classifier on the basis of that known content. Classification is the process of using the classifier with such training content and running it on unknown content to determine class membership.

The basis is a training set that contains class instances (malicious or benign), which are the principles of supervised machine learning [14]. The classifier used in this research can be compared to a Bayesian spam filter [15], which distinguishes 'spam' and 'nospam' categories. More specifically, the Naive Bayes algorithm for multinomial distributed data is used [16].

3.7 Dataset Evaluation

This section evaluates the dataset that is used to conduct the experiments. To gather samples for the creation of the dataset, benign samples of the banks web site are created with Internet Explorer, Firefox and Google Chrome. It is taken into consideration that users may have plugins installed that inject elements and code into the web site. These injections are benign and the classifier needs to be trained on this behavior in order to distinguish them from malicious characteristics.

Therefore, benign samples are created with Firefox and Google Chrome that have multiple plugins activated (1, 5 and 25 activated plugins). Currently, no statistics are published by both Google and Mozilla about the number of installed plugins. Accordingly, it is assumed that the majority of Chrome and Firefox users extended the default browser functionality with plugins. The created benign samples should ensure for a representative set that corresponds to the real world.

The experiments are performed with web pages from a major bank in the Netherlands. This bank provided a small number of known malicious samples. These samples are web pages, adapted by ZeuS, Citadel and SpyEye malware. However, the number of malicious samples is too limited in order to sufficiently train the classifier. Although this limitation, it is still possible to create a large dataset with mostly benign samples.

4 Results

This section describes the results derived from the experimental concept.

4.1 Feature Relevance

As described in section 3.4 the important features are identified by analyzing known samples that were modified by malware. However, an additional validation mechanism is required to determine the relevance of each individual feature. It is for example possible that a feature does not show enough deviation to distinguish a malicious from a benign page.

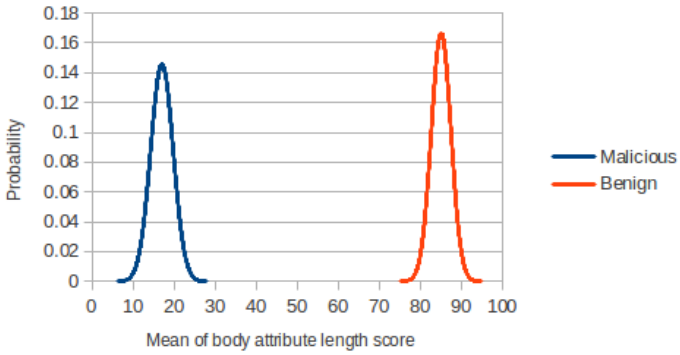

Figure 3: Normal distribution of the *body_attrib_length* feature

Therefore, for each feature, the mean and standard deviation for the benign and malicious dataset are calculated to verify their relevance. In figure 3, normal distribution is applied to visualize the distribution of the *body_attrib_length* feature between the different datasets. The figure shows that a large deviation exists between the two datasets for this feature. Hence, the feature can be used to distinguish between malicious and benign characteristics. Table 3 in A.2 shows how the mean and standard deviation for both malicious and benign web pages differ.

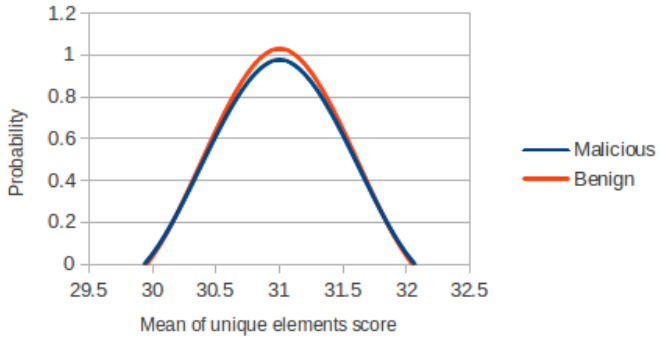

Figure 4: Normal distribution of the *unique_elements* feature

The opposite of figure 3 is shown in figure 4. The distribution does not show that there is a deviation between the two datasets for the *unique_elements* feature. Therefore, the feature is not relevant as it has no influence to distinguish a page from malicious or benign.

4.2 Performance

The performance of the system is divided into several parts:

- training of the classifier;
- feature extraction;
- classification of a web page.

Training of the classifier
The trainer fetches all the features from the database that are stored by the feature extraction. The features are processed by the preprocessor. Finally, the processed features are passed to the classifier to train on the data. Figure 5 shows the performance of the preprocessor and the trainer of the classifier.

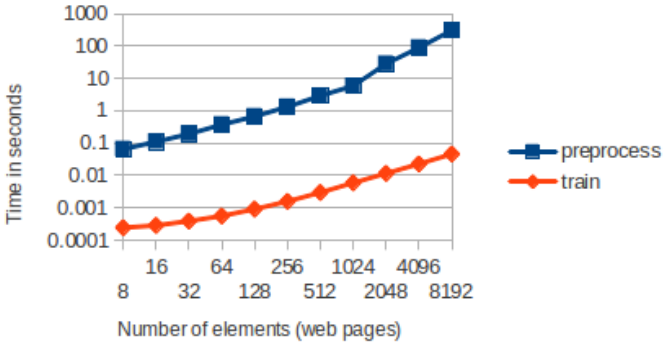

Figure 5: Performance of the trainer

It takes the preprocessor 307 seconds to process 8192 elements. The training of the classifier only takes 0.046 seconds. The preprocessor processes both HTML and URL features and highly depends on the database infrastructure. The training process consists of calculating the distribution of the data.

Feature Extraction
The feature extraction exists of two components: HTML features and features extracted from (Uniform Resource Locator) URLs found in a web page. Figure 6 shows the performance of the two components. Both variables show an increase in time when the character count increases in a logarithmic scale.

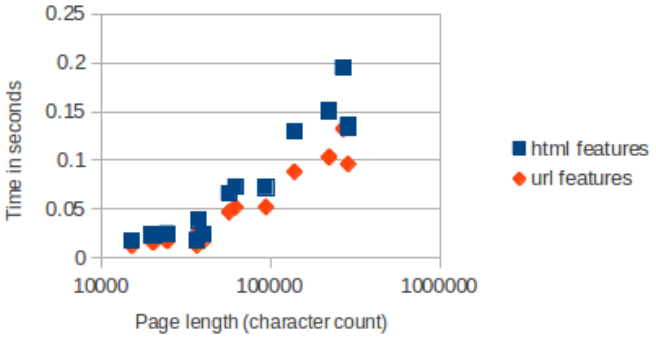

Figure 6: Performance of the feature extraction

Classification
The classification of an unknown page consists of three steps. First, features are extracted from the page. Subsequently, the features are processed by the preprocessor. The result from that process is a vector that is used to classify the page using the, already trained, classifier. Figure 7 shows the individual performance of the three different components that are required to perform the classification. The *execution time* variable is the sum of all components.

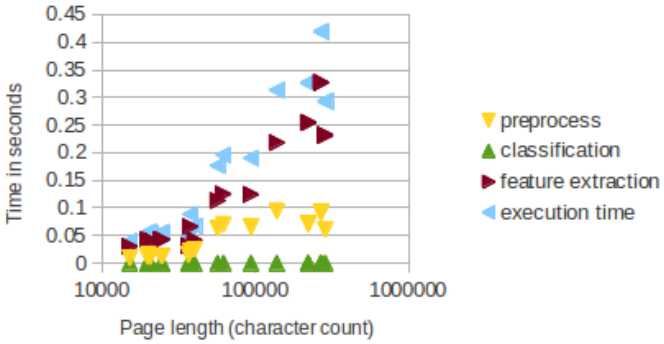

Figure 7: Performance of the classification

The increase of execution time is caused by the feature extraction and preprocess components that have to parse more data if the page length increases. The classification of a page takes only 0.0001 seconds, which is independent of the page length.

4.3 Classification Accuracy

As described in section 3.7 the used dataset contains a limited number of known malicious samples. Therefore, it is not possible to perform a valid test to measure the accuracy of the classifier. Although this limitation, an experiment is conducted to determine the current accuracy of the technique. The results from this experiment are an estimate on how the classifier could perform on novel data.

A test set with a total of 10 known malicious and benign samples is created and tested against the training set. This is an iterative process performed with an increasing amount of samples in the training set. The training set consists of mostly of benign samples due to the lack of known malicious samples. Only three malicious samples are used to train the classifier.

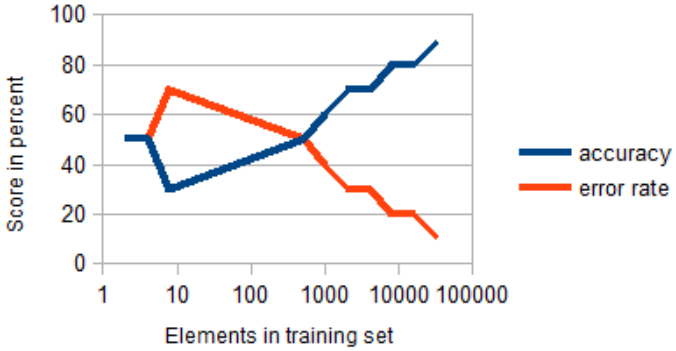

Figure 8: Classification accuracy

Figure 8 shows the results of the experiment. With only 8 samples in the trainer, the classifier is able to correctly classify 50% of the unknown instances in the test set. By increasing the number of benign samples, the classifier is more accurate. With 32,000, mostly benign, samples in the training set, the classifier reaches an accuracy of 90%.

Therefore, the hypothesis is formulated that a strong, benign baseline improves the classification process in order to detect unknown malicious pages. However, this can only be verified with a larger dataset and it is also required to verify the results from this experiment.

Model Validation

The results from the classification accuracy are promising (see figure 8). However, an experiment with a larger dataset is needed to verify the results. A simple experiment is conducted in order to validate that the model is able to correctly classify unknown malicious pages.

First, the classifier is trained on pages, adapted by ZeuS malware [17]. Subsequently, the classifier is tested on an unknown page, adapted by malware from the Citadel [18] family. In this test, the unknown page is classified as malicious with a probability of 100%.

5 Discussion

Although the developed concept is in an experimental state, the performance can be evaluated as reasonable, but further optimization will certainly reduce execution time. Areas of interest include a more extensive selection of features, a more efficient implementation to extract features from web pages and the usage of a high performance database system to lower the time to execute the preprocessor.

In addition to that, the decision to choose for a naïve Bayes classifier is primarily based on the fact that this classifier reaches its asymptotic error quickly with regards to the number of training examples [19]. Therefore, it is the preferred choice when using limited number of training instances.

Using the hold out method to evaluate the performance of the classifier, the dataset is partitioned into two mutually exclusive subsets. It is common to designate two third of the data to train the classifier and the remaining one third is used as test set to measure the performance [20].

The dataset used in this research does not allow for this cross-validation technique, as the dataset only contains a few number of malicious instances (see section 3.7). Due this limitation, the use of the m-fold cross-validation is also not possible. Therefore, a fixed test set is created with a total of 10 malicious and benign instances. The result from the classifier accuracy experiment (see section 4.3) can only be used as an estimate for further research.

Furthermore, due the shortage of malicious instances in the dataset, the used method to validate the model can be inappropriate and could lead to over-fitting. This term describes the situation in which a classifier is more accurate in classifying known data than predicting novel data. This is in general a problem when using small training sets [21].

Another subject of discussion are the privacy implications involved in an implemented model. If a client sends rendered web pages to a validation server, the implementation should consider the presence of privacy related data in these pages. In case of a financial organization, it is advised to handle that data with the same care as other sensitive customer data.

6 Conclusion

This research proposed and implemented an heuristic model for client-side fraud detection. Although the used dataset contains a limited number of malicious instances, the results from the experiments are promising. The classifier reaches an accuracy of 90%, but needs validation with a larger, more complete dataset. To validate the model and the used technique, a second experiment is conducted. By training the classifier on malicious pages, adapted by Zeus malware, it correctly classified an unknown page, adapted by Citadel malware as malicious.

The performance measures show that the classification of an unknown page is performed with a mean of 0.176 seconds. This also includes the feature extraction and preprocess operation. The training of the classifier with 8192 instances only takes 0.046 seconds. The preprocessing of the data, before the training, is a more expensive operation which takes 307 seconds for 8192 instances. After the preprocessing and training of the data, the classifier can be kept in memory and re-training is only required for new train data.

To conclude, with some certainty it can be ensured that the developed model is feasible to counteract fraud. Based on the used data set and validation method, the reached accuracy is satisfactory and can be used as an estimate for further research. The execution time to classify an unknown page is within bounds, since the requests can be handled asynchronous. However, since it is an experimental concept, further improvements can be made to lower

the impact on resources and optimizing execution time.

7 Further Work

Classification accuracy validation The major challenge for further work is validating the classification accuracy (see section 4.3). The limited number of malicious instances in the training set are not representative for the accuracy result. Although the classifier reaches an accuracy of 90%, this result is achieved by training the classifier on mostly benign instances. Also, the test set consisted more malicious instances than present in the training set. The used validation method might be inappropriate and a new experiment with a more complete dataset is required.

Hybrid implementation All components in the model are implemented on the server side. Therefore, the clients send the rendered web pages to a validation server. The easiest method, for malware, to by-pass the validation is executing the injections after the page validation process has completed. In order to counteract this, multiple validations of a page are required.

Sending a page multiple times to the validation server can lead to performance issues and an increase in bandwidth. To lower the impact on the validation servers, the *feature extraction* process can be implemented on the client-side using Javascript. Although such implementation significantly lowers the size of the requests, it is possible for attackers to disassemble the code and understand on which features the page is validated. Research on this field is required to determine if such a solution is feasible to implement.

Optimization of the model As an experimental proof of concept, the model is implemented in Python. The performance results (see section 4.2) show that most time is required for the feature extraction process. The feature extraction is implemented using the Beatiful Soup[5] library for Python. Optimizing the code, using another library or using a lower level programming language can improve performance.

Increase number of features A total number of 26 features are identified from the HTML, Javascript and URLs. Probably, more features can be identified in order to make the classifier more accurate.

References

[1] Lucian Constantin. Researchers warn of increased zeus malware activity this year. "https://www.networkworld.com/news/2013/052413-researchers-warn-of-increased-zeus-270142.html", May 24, 2013.

[2] emc.com. Phishing in season - tax time malware, phishing and fraud. "http://www.emc.com/collateral/fraud-report/rsa-april-2013-fraud-report.pdf", April 2013.

[3] Panda security press. Pandalabs q1 report: Trojans account for 80set new record. "http://press.pandasecurity.com/news/pandalabs-q1-report-trojans-account-for-80-of-malware-infections-set-new-record/", May 3, 2013.

[4] Ben Feinstein, Daniel Peck, and I SecureWorks. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. *Black Hat USA*, 2007, 2007.

[5] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web*, pages 197–206. ACM, 2011.

[6] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, pages 33–48, 2011.

[7] 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*, pages 281–290. ACM, 2010.

[8] Linda Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.

[9] Jiancheng Qin, Zhongying Bai, and Yuan Bai. Polymorphic algorithm of javascript code protection. In *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, volume 1, pages 451–454. IEEE, 2008.

[10] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. Jshadobf: A javascript obfuscator based on multi-objective optimization algorithms. In *Network and System Security*, pages 336–349. Springer, 2013.

[11] Eamonn Keogh. Naïve bayes classifier. "http://www.cs.ucr.edu/~eamonn/CE/Bayesian%20Classification%20withInsect_examples.pdf".

[12] Gama J. Bayesian learning: An introduction. "http://www.dcc.fc.up.pt/~ines/aulas/0809/MIM/aulas/bayes08.pdf", 2008.

[13] Stanford Encyclopedia of Philosophy. Bayes' theorem. "http://plato.stanford.edu/entries/bayes-theorem/", Revised September 30, 2003.

[14] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. The MIT Press, 2012.

[15] Ion Androutsopoulos, Georgios Paliouras, Vangelis Karkaletsis, Georgios Sakkis, Constantine D Spyropoulos, and Panagiotis Stamatopoulos. Learning to filter spam e-mail: A comparison of a naive

[5] http://www.crummy.com/software/BeautifulSoup/

bayesian and a memory-based approach. *arXiv preprint cs/0009009*, 2000.

[16] David D Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *Machine learning: ECML-98*, pages 4–15. Springer, 1998.

[17] Nicolas Falliere and Eric Chien. Zeus: King of the bots. *Symantec Security Respons e (http://bit. ly/3VyFV1)*, 2009.

[18] Ran Sherstobitoff. Inside the world of the citadel trojan. McAfee Labs, 2013.

[19] A Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in neural information processing systems*, 14:841, 2002.

[20] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, volume 14, pages 1137–1145, 1995.

[21] George Forman and Ira Cohen. Learning from little: Comparison of classifiers given little training. In *Knowledge Discovery in Databases: PKDD 2004*, pages 161–172. Springer, 2004.

[22] Ronald R Coifman and M Victor Wickerhauser. Entropy-based algorithms for best basis selection. *Information Theory, IEEE Transactions on*, 38(2):713–718, 1992.

[23] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

A Features

A.1 Feature Overview

This section describes an overview of the features that are selected from the HTML, Javascript and URLs found in a web page.

A.2 Feature Distribution

B Implementation Details

This section presents an overview of the software that is used to develop the proof of concept.

B.1 Used Software

- OS: Ubuntu Server 12.04
- Python 2.7.1;
- Python scikit-learn [23] (used to implement the Naive Bayes classifier);
- Python NumPy[6];
- Python Beatiful Soup[7] (used for the feature extraction)
- Database: MySQL-server 5.5.31.

B.2 Source Code

The source code developed for the experimental concept is published online as-is at the following location: `https://www.os3.nl/2012-2013/students/ttimmermans/rp2`. Be aware that the software is in an experimental state and that it is provided as-is.

[6] `http://www.numpy.org`
[7] `http://www.crummy.com/software/BeautifulSoup/`

HTML	
Feature	**Description**
style_percentage	The ratio of inline styles on the page
script_percentage	The ratio of javascript code on the page
iframe_elements	Number of iframe elements
script_elements	Number of script elements
style_elements	Number of style elements
input_elements	Number of input elements
hidden_fields	Number of hidden fields on the page. Also includes fields, hidden by CSS (*display:none and visibility:hidden*).
unique_elements	Count of unique HTML elements
input_ids	Count of unique IDs
unique_class	Count of unique class names
inline_style_length	Total length of all the inline styles on a page
embed_elements	Number of embed elements
object_elements	Number of object elements
html_attrib_length	Total length of all attributes used in the *html* tag
body_attrib_length	Total length of all attributes used in the *body* tag
URL	
tld	Top Level Domain (TLD) used in the URL.
domain	Domain in the URL
subdomains	Sub domains found in the URL
num_parameters	Number of parameters that are used in the URL
schema	Schema in the URL
out_of_scope	Check if the domain goes outside the scope of the banks web site.
length	Total length of the URL
Javascript	
entropy	The entropy of the Javascript code, using Shannon Entropy [22]
whitespace_percentage	Percentage of whitespace characters found in a Javascript snippet
dw_count	Number of calls made to DOM functions that can alter the page (*document.write, document.createElement etc.*).
eval_count	Number of calls made to the *eval* function, which is often used in obfuscated Javascript.

Table 2: Overview of the HTML, Javascript and URL features.

	malicious	**benign**
style_percentage		
mean	2.1455126386	6.8277776401
standard deviation	0.2643708541	1.0484134275
script_percentage		
mean	31.9801533973	28.9844513099
standard deviation	1.8917516104	1.6493897441
iframe_elements		
mean	0	0
standard deviation	0	0
script_elements		
mean	23	29
standard deviation	1.0801234497	1.5491933385
style_elements		
mean	11	7
standard deviation	1.779513042	1.5
input_elements		
mean	38	24
standard deviation	2.0816659995	0
hidden_fields		
mean	20	11
standard deviation	1.7320508076	1.6431676725
unique_elements		
mean	31	31
standard deviation	0.4082482905	0.3872983346
unique_ids		
mean	52	50
standard deviation	1.6072751268	1.3416407865
unique_classes		
mean	65	58
standard deviation	1.8257418584	1.4832396974
inline_style_length		
mean	522	291
standard deviation	9.9289140057	8.0187280787
embed_elements		
mean	0	0
standard deviation	0	0
object_elements		
mean	1	0
standard deviation	0.5773502692	0
html_attrib_length		
mean	16	36
standard deviation	0	2.8460498942
body_attrib_length		
mean	17	85
standard deviation	2.7386127875	2.3979157617
url_score		
mean	257	0
standard deviation	6.0277137733	0

Table 3: Distribution of features based on the dataset used in this research.