Recognizing network traces of various web services

Alexandros Hollender, Alexander Schaub

I. INTRODUCTION

Security on the Internet has become a major topic of interest during the past few years. Some major companies, for example, try to promote the use of HTTPS everywhere (Google announced they might improve the PageRank of websites using this technology). HTTPS also enables users to protect their privacy over open Wifi-hotspots. However, there are means of retrieving private information using so-called side-channel attacks. In short, one can for example analyze the sizes of the packets sent and received by some user, and gain access to a lot of information on what they are doing. This idea has been applied to bank-websites, where the investment portfolio could be retrieved using only the packet size, or on a health website, where the disease a person was looking for could also be retrieved in a similar matter, although both websites are protected using HTTPS [1]. We also investigated this kind of threat in previous work [7] that was published in the CRISIS'14-conference. We focused on the auto-completion functionality of search engines, which enabled us to retrieve the search term the user was looking for in various situations.

In this project, we will expand our work by building a classifier that is able to detect whenever a user visits the homepage of a search engine and uses its autocompletion feature. Since our previous work enabled us to relate the corresponding network traces to the user input, we would therefore be able to scan large sets of network traces and retrieve the search terms looked for by some user. Our project comprises two main parts: first, detecting whenever a users loads the homepage of a search engine (among a defined set of search engines). Second, detect the exact network trace corresponding to the packets exchanged while the autocompletion feature is being used. Using the first part, we would be able to detect which search engine was being used, while the second part allows us to target this specific search engine and retrieve the relevant packets.

II. RELATED WORK

The first part of our problem is quite extensively treated in [4] where a multinomial Naive Bayes classifier

is used. Several optimizations presented in that publication could be used to further improve our classifier. Overall, this proves the feasibility of the first part. Our application being quite specific, we did not find any previous work treating the second part of our problem. However, there is some work concerning the more general problem of analyzing network traces using machine learning techniques.

A big part of the literature on this subject focuses on traffic classification. Given a network trace, the objective is to extract the different flows contained in it and to identify which protocols and applications they correspond to. Possible approaches include the Naive Bayes classifier [12], the EM clustering algorithm [5] or unsupervised Bayesian classifiers [10].

Another problem that is also quite extensively treated is network anomaly detection. The goal is to be able to recognize unusual activity in a network that is caused by attacks or intrusions. Neural networks [11], nearest neighbor [2] and SVMs [3], [8] are some techniques that have proven very useful for this problem.

However, the majority of the network-related literature using machine learning assumes that there is more information available than in our setting. Most approaches use information located in the packet headers, which we suppose to be unavailable because of encryption. Only few use a similar setting to ours, where only timing and packet size are available. Some examples are [9] where this setting is used for traffic classification and [4] as stated earlier.

III. DATA SETS AND FEATURES

Since no data sets for this quite particular application are available, we generated our own data sets through an automated process. Specifically, we used Selenium ¹ in order to control the web browser, load pages and trigger the auto-completion feature, and the jNetPcap library ² in order to intercept the packets and thus store the packet sizes. Because we must wait until the pages are fully

¹http://www.seleniumhq.org/

²http://jnetpcap.com/

loaded, it takes around 15s to build one training example. However, we can build as many training examples as needed.

A. Identifying loaded webpage

For this part of the project, each training example consists of the web trace captured when loading a certain webpage. The number of packets captured usually ranges from 500 to 3000 packets, depending on the website loaded, but also slightly fluctuating each time we load the same website. To obtain feature-vectors of constant size, we define the j-th feature to be the number of packets of size j that appear in the captured web trace, i.e.

$$x_j = \#\{packet \in T : size(packet) = j\}$$

where T is a raw captured web trace. Because of limitations in the protocols used to transfer data on the Internet, the size of a packet cannot exceed 1514 bytes. Thus, each training example is a 1514-dimensional vector, counting the number of occurrences of each packet size. It is important to note that by using those features, we lose some information that was contained in the raw captured web trace, e.g. the order of the packets. However, as we will see later, the features we use retain enough information for our application, while also being much easier to work with than the raw web traces. Using our automated tool we were able to build 100 training examples for each of the websites we want to identify.

B. Auto-completion trace detection

For this part, a training example corresponds to the \sim 20 packets exchanged between the user of an autocompletion feature and the server of the search engine. Each training example corresponds to the network activity triggered by a user entering one more letter of its request, and causing Google to update the suggestions being displayed. We built around 200 of those training examples. In order to get web traces corresponding to the "average" browsing behaviour (without auto-completion usage), we recorded the network activity corresponding to a surf session comprised of popular websites (such as Facebook, Wikipedia, news websites, etc.). The test set corresponds to a browsing session including one or more uses of the auto-completion feature. During the browsing sessions, we recorded about 10,000 packets. Finally, we simulated a mixed session (popular websites and search engine use) that would be used to test our algorithms, and captured about 200,000 packets, with a total of 40 uses of the auto-completion feature. The same representation of the feature vectors as in the first part is used in the second part of the project.

IV. METHODS AND RESULTS

A. Identifying loaded webpage

For this part we mainly used a Naive Bayes classifier with a multinomial event model and Laplace smoothing. Specifically, we used the training set to estimate

$$p(\text{size } j|\text{webpage } i)$$

i.e. the probability that a given packet in the web trace of webpage i has size j. This yields a multinomial distribution over the packet sizes for a given webpage i. When estimating those probabilities we use Laplace smoothing to avoid any of them being equal to 0 (very important as a lot of packet sizes never occur in the training set). We also use the training set to estimate p(webpage i).

Using the Naive Bayes assumption we can then estimate the probability that a given web trace corresponds to webpage i as

$$\prod_{j=1}^{1514} p(\text{size } j|\text{webpage } i)^{x_j} p(\text{webpage } i)$$

divided by the probability that the web trace occurs, which does not depend on i (and we can thus leave out when maximizing with respect to i). In this expression x_j is the number of packets of size j occurring in the given trace. The most probable webpage for the given web trace is the webpage i that maximizes this expression.

We implemented this algorithm and first tried it on a binary classification problem: identifying whether a web trace corresponds to loading Google (google.com) or Wikipedia (en.wikipedia.org). For both of these webpages, a web trace corresponds to $\sim\!600$ packets, so we thought it would be interesting to see if we can differentiate them. Using 5-fold cross-validation on our dataset containing 100 Google traces and 100 Wikipedia traces we obtained an average test error of 0%! We were quite surprised, but after taking a closer look at the dataset, we understood why the classifier worked that well. We plotted the average number of packets of each size for both cases on **Fig 1** and **Fig 2**.

We also estimated how indicative of each website each packet size is. A packet size j is indicative of website Google if

$$\log p(\text{size } j|\text{Google}) - \log p(\text{size } j|\text{Wikipedia})$$

Fig. 1. Google: average number of packets of size 1200-1514

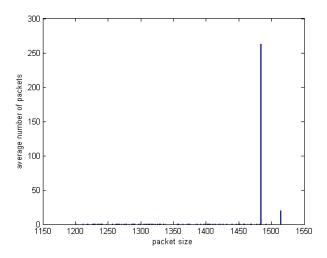
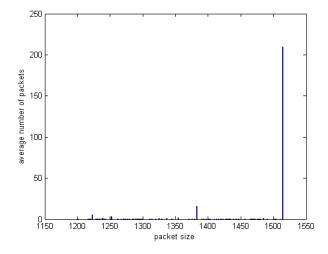


Fig. 2. Wikipedia: average number of packets of size 1200-1514



is large, where we use the probabilities estimated by our Naive Bayes algorithm. We thus found out that the most indicative packet size for Google is 1484, and the most indicative packet size for Wikipedia is 1383. Those values make sense if we look at the graphs (**Fig** 1 and **Fig** 2). Even though packet size 1514 appears a lot for Wikipedia (~ 200 times on average), it appears relatively often for Google too (~ 25 times on average). Thus, Naive Bayes finds the other two packet sizes more indicative of each class.

The next step was multi-class classification. We added 5 other webpages to the dataset: Amazon (amazon.com), Bing (bing.com), Facebook (facebook.com), Yahoo (yahoo.com) and Youtube (youtube.com). Thus, our dataset now contained 700 web traces corresponding to 7 different webpages. Using 5-fold cross-validation on this

dataset with our Naive Bayes algorithm we obtained an average test error of 0.14%. This was quite surprising as we did not expect Naive Bayes to work as well with 7 classes as it did with 2.

However, as it turned out, we had been quite lucky. Continuing on our track, we added 3 more webpages to the dataset: Twitter (twitter.com), Ebay (ebay.com) and LinkedIn (linkedin.com). Running Naive Bayes on this 10-class problem yielded an average test error of 44%. Taking a closer look at the results, we noticed that most of the Google, Youtube and Ebay test examples where classified as LinkedIn, and most of the Twitter test examples were classified as Yahoo. By adding those 3 additional webpages, we had added some packet size distributions that were too close to the other ones and Naive Bayes was no longer able to differentiate them properly.

As a result, we decided to try another standard algorithm on our dataset: logistic regression. Our problem being a multi-class classification problem, we trained one classifier for each class. The positive examples were the examples corresponding to the class, and the negative examples were those corresponding to all other classes. We then classified each test example to the class whose classifier gave the highest value. Using 5-fold cross-validation yielded an average test error of about 1% for both the 7-webpage and the 10-webpage case. Thus, logistic regression seems to have a comparable performance on the 7-webpage case, but, unlike Naive Bayes, it is able to keep that performance on the 10-webpage case.

However, logistic regression has a huge disadvantage with respect to Naive Bayes: it runs much slower. So, if the webpages we want to classify have packet size distributions that are different enough, using naive Bayes is certainly the best idea. This is especially true in a setting where the classifier has to be re-trained quite often, because most webpages are not static objects, but change daily. However, if some packet size distributions are too close, Naive Bayes will not work, and we will have to use logistic regression.

B. Auto-completion trace detection

We used the 200 traces corresponding to the autocompletion feature for the search engines Google and Bing, as well as the "average" browsing session, to train the Naive Bayes classifier. For each of the search engines, we then tried to find the packets corresponding to the autocompletion feature in the test web-trace. Since the number of packets corresponding to one use of the autocompletion feature is fairly constant (around 20 for Google, 30 for Bing), we used a sliding window in order to determine the positions of those traces. For example, for Google, we first considered the 20 first packets (packets 1 to 20) of the test trace, used the Naive Bayes classifier in order to label them *Google* or *non-Google*, then moved to the packets at offset 2 to 21, classified again, etc.

Since a lot of packets are overlapping between two successive windows, we expect our algorithm to find much more probable autocompletion traces than the actual number of times we used this feature. Again, the feature space has 1514 dimensions, the elements of those vectors are the number of packets of a given size in a sequence.

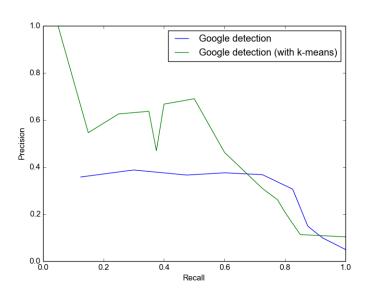
The Naive Bayes classifier tends to find most of the auto-completion traces, but at the cost of a large number of false positives. Many of those false positives are to be expected, but the rate of false positives remains high. Therefore, we have to use an other algorithm in order to determine more reliably the traces corresponding to the auto-completion feature, or use the Naive Bayes as the first step of a multi-layered algorithm.

We decided to implement k-means clustering as the second layer. The idea is to remove packet sequences that are misclassified by the Naive-Bayes classifier, for example a sequence containing a large number of packets that have the same size and that happens to be the size of one of the packets appearing often in our positive training data. The number of clusters has been determined by trial-and-error, we settled with 10 clusters. Once the 10 clusters have been determined by k-means clustering, we select the one (in the case of Bing) or two (for Google) clusters whose centroids are the closest from the centroid of our positive training data. Since the clustering is stochastic, we run the clustering several times and then select the results for which the closest centroid had the smallest distance to the centroid of the positive training data among all the runs. K-means was applied using the scikit Python library [6].

Lastly, we also modified the way we computed the precision and recall. Only counting as positive matches the offsets which exactly match the offsets we recorded seemed too restrictive. Therefore, we counted as positive match any offset that was close enough to the recorded

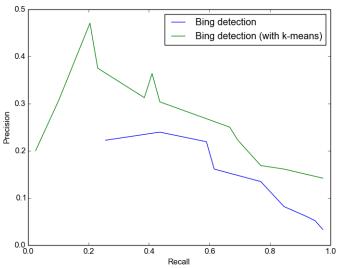
offsets, but for every recorded offset we would not count more than one positive match. For example, if we recorded that the packets at offsets 10 to 24 correspond to the use of the autocompletion feature, and after the k-means clustering we get as results the offsets, 8, 12, 30 and 31, then we would have a recall of 1 (the usage of auto-completion was correctly detected by offsets 8 and 12, which are "close-enough" to 10) and a precision of $\frac{1}{3}$ (the offsets 30 and 31 were misclassified, and one usage of auto-completion was correctly detected, thus, the precision would be $\frac{1}{1+2} = \frac{1}{3}$). Figures 3 and 4 show the precision-recall curves for Google and Bing respectively, with and without using k-means clustering as a second layer. Different precision / recall values have been obtained by varying p(y), the probability for a given sequence of packets being one corresponding to the usage of auto-completion.

Fig. 3. Precision vs recall for Google traces



As the results show, applying k-means clustering after Naive Bayes effectively removes unrelated packets sequences while keeping the correctly classified ones. Especially in the case of Bing, filtering the result from the first stage using k-means clustering removes a lot of misclassified packet sequences. The shapes of the precision-recall curves may seem strange. The noise in the data might explain this, as well as the large amount of randomness (it is not even sure that the positive training data and the part of test data that should correspond to the auto-completion are drawn from the same distribution, as those distributions of packet sizes vary in time).

Fig. 4. Precision vs recall for Bing traces



We also tried SVM and one-class-SVM to classify the sequences. It did not seem to work - one-class-SVM did not recognize a single trace, and SVM had a too low precision.

V. CONCLUSION

For the first part, we were able to identify a loading webpage amongst 7 of the most popular webpages with very high accuracy using a Naive Bayes classifier. However, extending the set of possible webpages hugely increases the classification error, and we had to use logistic regression to obtain good results in that case. Whether we can use Naive Bayes or not depends on the exact nature of the packet size distributions of the webpages. Future work on this topic could include testing on even more webpages to see if logistic regression continues to perform well, as well as examining other classification algorithms that run faster than logistic regression. Another possibility would also be to use feature selection to decrease the dimension of the feature-space and make logistic regression run faster.

For the auto-completion detection, Naive-Bayes with additional k-means clustering seems to work quite well. While there are still a non-negligible number of false positives, we managed to reduce the sequences of packets which might contain auto-completion usage from about 200,000 to less than 200 with most settings. One particular aspect we did not consider yet is the ordering of the packets. Although the exact ordering of

the packets with the same purpose varies sometimes, it is not completely random. This is not taken into account by our approach, which is therefore more robust (we miss less sequences) but less precise (we still have false positives). A third layer of filtering might be the solution, but it would be more complex to design than Naive-Bayes or k-means clustering.

Overall, it seems that classical machine-learning methods can be adapted for network analysis purposes, although the way the data is processed and represented requires extra care. By combining both of our findings, it seems possible to scan through large web traces in order to determine which search engine might have been used during a browsing session and when the auto-completion feature had been used. These traces can, for example, be easily obtained by a standard laptop with off-theshelf hardware running the adequate (free) software and installed near a busy public or even private WiFi hotspot, or by intercepting the data at router level. Using our prior work [7], it would be possible to determine which search terms have been looked for by the user, only looking at the sizes of the exchanged packets. Although it is not possible to do so anymore on Google - the size of the packet containing the relevant data we exploited in our earlier paper has been completely randomized now, rending side-channel attacks on this particular feature probably unfeasible - there are a lot of web services that have not yet been protected against these attacks.

REFERENCES

- [1] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 191–206. IEEE, 2010.
- [2] Levent Ertoz, Eric Eilertson, Aleksandar Lazarevic, Pang-Ning Tan, Vipin Kumar, Jaideep Srivastava, and Paul Dokas. Mindsminnesota intrusion detection system. *Next Generation Data Mining*, pages 199–218, 2004.
- [3] Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Sal Stolfo. A geometric framework for unsupervised anomaly detection. In *Applications of data mining in computer security*, pages 77–101. Springer, 2002.
- [4] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In Proceedings of the 2009 ACM workshop on Cloud computing security, pages 31–42. ACM, 2009.
- [5] Anthony McGregor, Mark Hall, Perry Lorier, and James Brunskill. Flow clustering using machine learning techniques. In Passive and Active Network Measurement, pages 205–214. Springer, 2004.
- [6] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikitlearn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [7] Alexander Schaub, Emmanuel Schneider, Alexandros Hollender, Vinicius Calasans, Laurent Jolie, Robin Touillon, Annelie Heuser, Sylvain Guilley, and Olivier Rioul. Attacking suggest boxes in web applications over https using side-channel stochastic algorithms. In Risks and Security of Internet and Systems, pages 116–130. Springer, 2014.
- [8] Taeshik Shon, Yongdue Kim, Cheolwon Lee, and Jongsub Moon. A machine learning framework for network anomaly detection using svm and ga. In *Information Assurance Work-shop*, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC, pages 176–183. IEEE, 2005.
- [9] Charles Wright, Fabian Monrose, and Gerald M Masson. Hmm profiles for network traffic classification. In *Proceedings of* the 2004 ACM workshop on Visualization and data mining for computer security, pages 9–15. ACM, 2004.
- [10] Sebastian Zander, Thuy Nguyen, and Grenville Armitage. Automated traffic classification and application identification using machine learning. In Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on, pages 250–257. IEEE, 2005.
- [11] Zheng Zhang, Jun Li, CN Manikopoulos, Jay Jorgenson, and Jose Ucles. Hide: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification. In *Proc. IEEE Workshop on Information Assurance and Security*, pages 85–90, 2001.
- [12] Denis Zuev and Andrew W Moore. Traffic classification using a statistical approach. In *Passive and Active Network Measurement*, pages 321–324. Springer, 2005.