HeadPrint: Detecting Anomalous Communications through Header-based Application Fingerprinting

Riccardo Bortolameotti
University of Twente
r.bortolameotti@utwente.nl

Thijs van Ede
University of Twente
t.s.vanede@utwente.nl

Andrea Continella
UC Santa Barbara
conand@cs.ucsb.edu

Thomas Hupperich
University of Muenster
thomas.hupperich@wi.uni-muenster.de

Maarten H. Everts
University of Twente
maarten.everts@utwente.nl

Reza Rafati
Bitdefender
rrafati@bitdefender.com

Willem Jonker
University of Twente
w.jonker@utwente.nl

Pieter Hartel
Delft University of Technology
pieter.hartel@tudelft.nl

Andreas Peter
University of Twente
a.peter@utwente.nl

ABSTRACT

Passive application fingerprinting is a technique to detect anomalous outgoing connections. By monitoring the network traffic, a security monitor passively learns the network characteristics of the applications installed on each machine, and uses them to detect the presence of new applications (e.g., malware infection).

In this work, we propose HeadPrint, a novel passive fingerprinting approach that relies only on two orthogonal network header characteristics to distinguish applications, namely the order of the headers and their associated values. Our approach automatically identifies the set of characterizing headers, without relying on a predetermined set of header features. We implement HeadPrint, evaluate it in a real-world environment and we compare it with the state-of-the-art solution for passive application fingerprinting. We demonstrate our approach to be, on average, 20% more accurate and 30% more resilient to application updates than the state-of-the-art. Finally, we evaluate our approach in the setting of anomaly detection, and we show that HeadPrint is capable of detecting the presence of malicious communication, while generating significantly fewer false alarms than existing solutions.

KEYWORDS

application fingerprinting, network security, anomaly detection

ACM Reference Format:
Riccardo Bortolameotti, Thijs van Ede, Andrea Continella, Thomas Hupperich, Maarten H. Everts, Reza Rafati, Willem Jonker, Pieter Hartel, and Andreas Peter. 2020. HeadPrint: Detecting Anomalous Communications through Header-based Application Fingerprinting. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3341105.3373862

1 INTRODUCTION

Data breaches are a major concern for enterprises across all industries due to the severe financial damage they cause [25]. Although many enterprises have full visibility of their network traffic (e.g., using TLS-proxies) and they stop many cyber attacks by inspecting traffic with different security solutions (e.g., IDS, IPS, Next-Generation Firewalls), attackers are still capable of successfully exfiltrating data. Data exfiltration often occurs over *network covert channels*, which are communication channels established by attackers to hide the transfer of data from a security monitor.

Detecting data exfiltration over network covert channels is difficult because the exfiltration can have many shapes and patterns in the network traffic, which make it problematic to model its characteristics. For example, an attacker can obfuscate data and exfiltrate it in small chunks over a long period of time, or she can transmit all the information at once. Characterizing exfiltration patterns over the HTTP protocol becomes even harder, because it mostly contains heterogeneous content. It is not a coincidence that malware often uses HTTP [19, 31, 38] for its communication, since the protocol allows any application to insert arbitrary values in many parts of the messages (e.g., `Cookie`, `Body`, `URI`, etc.).

For these reasons, researchers proposed different *anomaly-based* detection techniques [3, 5, 9, 29], which focus on modeling the *normal* traffic rather than modeling the possible patterns of exfiltration attempts. Data exfiltration attacks are detected by identifying traffic deviating from the normal model. The DECANTeR [5] system proposed passive application fingerprinting (PAF) as a solution to detect anomalous outbound traffic. The idea behind a PAF system is to passively monitor the traffic of a machine and to learn the network characteristics of each application (i.e., software) communicating from that machine. These network characteristics are used to create application fingerprints. Once the fingerprints are created, the system monitors the machine's traffic. An alert is triggered when the system observes network messages with characteristics deviating from installed applications. An anomaly may represent an attempt of exfiltration, or C&C communication, of a newly installed (malicious) application. Since network security devices often do not have access to the victim machines, it is important that fingerprints are generated by only passively observing their traffic. This work

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373862

focuses on PAF because it has shown better detection performance than other existing techniques [5].

DECANTeR generates many false alerts, and it assumes a continuous human intervention to investigate false alerts in order to update the fingerprints. The problem of investigating many false alerts is known as *alert fatigue*, where operators in a Security Operation Center are overlooking real security issues because of too many false alarms. The problem of false alarms is well known also in academic literature [30]. The cause of the false alarms in DECANTeR is the fingerprinting method based on a small set of fixed and hand-picked features, which intrinsically causes the following limitations: 1) low accuracy in identifying application messages for which these features are not discriminative (e.g., web scripts in browser traffic); and 2) the fixed selection cannot deal with dynamic changes introduced by application updates.

In this work, we propose a novel fingerprinting technique that makes PAF for anomaly detection more practical by reducing the false alerts and by removing the need for human intervention, while still being able to detect malicious connections. We present HeadPrint, a technique to fingerprint outbound HTTP traffic generated by real-world applications. HeadPrint automatically infers the relevant characteristics of applications traffic by only examining the protocol headers of their messages. The idea is that headers intrinsically include the semantics of network messages and so allow to discriminate different applications. Specifically, our approach is based on two different types of information: 1) header sequences (i.e., the order of the header fields), and 2) the values associated with the header fields. We apply a technique based on entropy to *automatically* recognize the headers that are most discriminating for a certain application. Finally, after generating fingerprints, our algorithm evaluates how similar new messages are to a specific application fingerprint, and learns the optimal decision function to determine whether the message belongs to that application. The automated identification of characterizing headers, together with the combination of two orthogonal message characteristics, namely the header sequences and the values associated with them, represent the technical novelty of our work.

We implement HeadPrint and we evaluate its performance in terms of fingerprinting accuracy, resilience to application variations like software updates, and detection performance in the setting of passive application fingerprinting for anomaly detection. The evaluation was performed with a dataset containing real-world network traffic from an international organization. We have compared the results of HeadPrint with the current state-of-the-art, DECANTeR [5], showing a significant improvement in all aspects of the evaluation. On average, our fingerprinting technique shows an improvement of 20% in accuracy and 30% in update resilience. Regarding the detection performance, HeadPrint generates significantly fewer false alerts, while being able to detect malicious HTTP communication correctly. Finally, HeadPrint does not rely on the assumption of a human operator that manually monitors and updates the fingerprints.

2 HEADPRINT

The intuition behind our fingerprinting technique is that the network messages of an application share similar *header values*, and *header sequences* over time. In other words, we can distinguish applications by learning what values an application associates with each specific header (i.e., header values), and by learning in what order the headers are inserted in each application message. These characteristics complement each other: while the header values capture the semantics of the applications, the header sequences catch their implementation details.

2.1 Overview

In HeadPrint, each application fingerprint is represented by two distinct models: a *header-value entropy* model and a *header sequence* model. The process of generating a fingerprint for an application a works as follows: (1) we collect the set of messages M generated by application a; (2) we create from M the two different models $\mathtt{VAL}_a$ and $\mathtt{SEQ}_a$, which represent the fingerprint of the application $F_a = (\mathtt{VAL}_a, \mathtt{SEQ}_a)$.

Once the fingerprint is generated, we can evaluate whether a new message t has been generated by application a by comparing t with the models of F_a. This process is performed in two steps. First, we evaluate how similar the header values and the header sequence t are with the respective models $\mathtt{VAL}_a$ and $\mathtt{SEQ}_a$ in F_a. We achieve this by using tailored similarity functions, which output scores between 0 and 1 (= identical). Second, a decision function d checks whether both scores are above (automatically determined) thresholds. If so, it attributes the message t to the application a.

2.2 Header-Value Entropy Model

The header-value entropy model $\mathtt{VAL}_a$ for application a consists of a set of (h, V) tuples, where h is a header field and V is the multiset of values assigned to h, for all messages M. The underlying assumption for this model is that an application a generates messages such that the values corresponding to certain headers often reoccur. Hence, we expect that a new message t from application a also contains header values that have previously been observed, and are therefore contained in $\mathtt{VAL}_a$.

Unfortunately, not all header values exhibit recurring behavior. When a header field value changes very often, it is likely that a new message from the same application will also contain a different value for the same header field. Non-recurring headers are not characteristic for an application, therefore the header-value entropy model identifies and discards them. In order to recognize if a header h and its corresponding values V are characteristic for the application, we compute the *entropy* $H(\cdot)$ for the values in V. We use entropy because it is a measure for *unpredictability.* If the entropy is low, the values in V can be considered "predictable," i.e., they are likely to reoccur over time, and that the tuple (h, V) is considered characterizing for the application. In case of high entropy values, the tuple is discarded. Consequently, we define $\mathtt{VAL}_a$ as the set of (h, V) tuples, where $H(V) \leq \alpha$, and α is a threshold that defines the highest entropy value V can have to be considered "predictable".

To check whether a new message t has similar values to application a, we evaluate $value\_sim(t, \mathtt{VAL}_a) \in [0, 1]$, which is a similarity function for categorical data. It evaluates how similar the values in message t are to the historical messages of application a (i.e., represented by $\mathtt{VAL}_a$). The function identifies the subset of headers $\{h_i\}$ that represents the intersection between the headers in

t and in VAL_a. For each header in $\{h_i\}$, if the value corresponding to h_i in message t is present in the historical values V_i where $(h_i, V_i) \in \mathsf{VAL}_a$, then we consider the value to be a *match*. The similarity score is the number of matches divided by the number of headers in $\{h_i\}$. This technique for computing the similarity for categorical data is called Overlap [4].

2.3 Header Sequence Model

The header sequence model SEQ_a consists of the set of header sequences of all messages M observed for application a, where a header sequence is defined as the list of headers occurring in a message. The underlying assumption of this model is that application a uses a specific header sequence when it generates its messages. Therefore, we expect that a new message t from application a uses a similar (if not the same) header sequence that has previously been observed from a.

Although the majority of applications uses a fixed set of header sequences for their communications, this is not true for all applications. For instance, the header sequences of browser messages sometimes depend on third-party JavaScript code, which injects additional headers, thereby making it very hard (if not impossible) to find an exact match. Nonetheless, most of the time these new header sequences are still similar (to a certain degree) with those previously observed from the same application. For this reason, the concept of similarity for the header sequences is effective.

To evaluate whether a new message t has a similar header sequence to application a, we compute $seq\_sim(t, \mathsf{SEQ}_a) \in [0, 1]$, a function that computes a similarity score between the header sequence of t and each sequence in SEQ_a. We use the sequence similarity metric proposed by Bakkelund [1], which is based on the *longest common subsequence* (LCS) problem. The similarity score is defined as the length of the LCS between the two sequences, divided by the length of the longest sequence. In other words, the larger the longest common subsequence between two sequences is, the more similar the sequences are. After we compute the scores, we return the maximum. We choose to rely on LCS because subsequences are not required to occupy consecutive positions within the original sequences, while this is not the case for the longest common substring problem. It is difficult to predict where new headers can be observed in a header sequence, because it may depend on specific web application scripts. Therefore, LCS can still identify two messages as similar despite new headers are inserted, because previously known headers occur with the same relative order, thereby generating a long subsequence.

2.4 Decision Function

When we compare a message t with a fingerprint F_a, we first compute two distinct scores x and y, which are the output of $value\_sim()$ and $seq\_sim()$, respectively. These scores are then evaluated by a decision function $d(x, y) \in \{0, 1\}$, which determines whether t is a message of application a or not.

In other words, d is a binary classifier for a two dimensional plane, where $value\_sim()$ and $seq\_sim()$ similarity scores represent the axes (i.e., x and y, respectively). The classifier decides whether the point (x, y) is in an area of the plane where messages are considered to match the fingerprint, or not. Figure 1 shows the distribution of the scores of messages when compared with the fingerprint of an instance of Firefox. The green crosses represent messages from all the applications installed on the host that *are not* Firefox, while the blue circles represent messages from Firefox. When t is compared with the fingerprint of its application, we expect a point in the top-right part of the plane as both similarity functions should yield scores approaching to 1. Given that the models underlying the fingerprint of most applications differ vastly from each other, a high similarity score is only produced when a message is compared to the fingerprint of the application it originated from. Hence, we can create a *general* decision function d for all fingerprints.

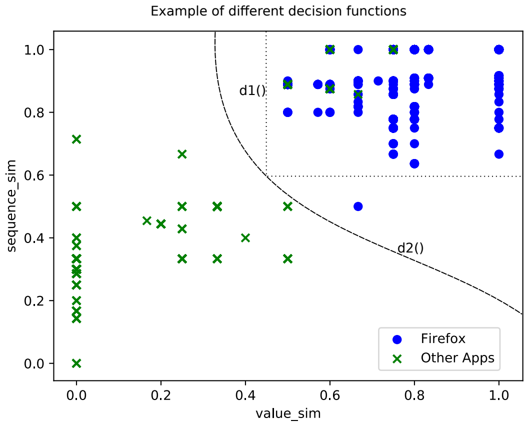

Figure 1: Distribution of messages after they have been compared with the fingerprint of Firefox. Crosses represent messages from applications other than Firefox. Dots represent messages from Firefox. The straight and dashed lines represent two possible decision functions.

As shown in Figure 1, there can be different ways to determine a function d. We propose and evaluate two different approaches: 1) Static-rules, where we use a threshold-based decision function (i.e., similar to $d1()$ in Figure 1) that we define according to our domain knowledge; 2) ML-classifiers, where we train a classifier using machine learning (e.g., similar to $d2()$ in Figure 1).

Choosing the Decision Function. We can define a static rule as a decision function because messages receive a high similarity score when evaluated against their application fingerprint and low otherwise, thereby creating a predictable distinction. In this work, we chose to evaluate threshold-based rules that consider a similarity score (x, y) to be a match if $x > X$ and $y > Y$, where X and Y are two thresholds.

Although a predetermined function has the advantage of neither requiring a learning phase nor being computationally expensive, it is likely a sub-optimal solution in terms of classification performance. At the expenses of an ad-hoc learning phase, machine learning can be used to learn a decision functions from the data, which likely provides a more precise solution. However, to properly learn a decision function, we need to obtain a representative dataset for a binary classification problem, where the positive class represents the scores of messages compared with their application fingerprint, and the negative class represents the score of messages compared with other applications fingerprints.

The generation of such a dataset is independent from the fingerprinting approach itself, and it can be achieved in two different ways, looking at the available resources. One method is to generate a dataset by analyzing many applications offline. Although this setup can provide a precisely labeled dataset, it can require a lot of resources. Alternatively, the dataset can be built by monitoring the traffic for a set of hosts. This setup requires fewer resources, and it can contain data contextualized to the network where the system would ultimately be deployed, but the dataset is unlabeled. In this case the `User-Agent` can be used as groundtruth, since it commonly represents a unique string identifying a specific application. Consequently, when a message is compared with a fingerprint having a *similar* `User-Agent` value (e. g., variations of version should be taken into account due to updates), the similarity score (x, y) is labeled with 1 (positive class), and 0 otherwise (negative class). In this work, we use the latter approach.

3 HEADPRINT FOR ANOMALY DETECTION

The anomaly detection setting works as follows: (1) during a training phase, we passively learn the fingerprints of applications installed on a machine; (2) during testing phase (i.e., when the system is live) we verify for each message whether it matches with any known fingerprint, and, in case no match is found, the message is considered *anomalous*, since it is generated from an *unknown* application. An alert is triggered when a set of anomalous messages, which share the same destination, transmit more than a certain amount of data, which is defined by a threshold T. Please note that the *training is always passive*, thus no access on the host is required.

3.1 Threat Model

We assume a network monitor is deployed in an enterprise environment to monitor and extract network information from servers and workstations. The monitor sends this information to a back end system responsible for analyzing the network information and identifying anomalies in the traffic. This is a common setup in practice. We assume that the attacker cannot compromise the monitor and the back end system. We also assume the malware uses HTTP to communicate over the network. HTTP is still one of the most used protocols by malware [19, 31, 38]. A main reason for this is the fact that HTTP traffic is usually not blocked by firewalls and malware can camouflage the data exfiltration within huge volumes of benign HTTP traffic. HeadPrint can be applied to HTTPS in enterprise scenarios where TLS-proxies are being deployed to inspect encrypted traffic. This scenario is rather popular in enterprises where data exfiltration is a major concern.

3.2 Training Fingerprints

We assume the training phase to be *trusted*, meaning we expect that machines are not compromised during the training period. In practice, the training phase can be assisted by deployed security tools, which can help identify malicious hosts, thereby avoiding the generation of fingerprints for known malware communication.

3.2.1 Convergence. A distinctive aspect of HeadPrint is how it automatically determines the required amount of data needed to train a certain application's fingerprint. Intuitively, the amount of training data affects the robustness of our fingerprints in terms of fingerprinting accuracy. However, this parameter strongly depends on the complexity of the application that HeadPrint fingerprints. For instance, some applications always produce the same network traffic, and hence can be accurately fingerprinted after a few messages, while others (e.g., browsers) might have a very dynamic network behavior. Therefore, similarly to [8], HeadPrint divides the training phase in intervals, repeatedly tests the generated fingerprints after each interval, and stops the training when new training data does not add information to the fingerprint. We call this a *converged state*. In practice, we achieve this by testing the freshly generated fingerprints at intermediate steps, HeadPrint can understand when it collected enough data to model the application's network behavior. In practice, we say that an application reaches convergence when we do not see any prediction mistake for its fingerprints for K consecutive intervals.

More precisely, HeadPrint performs the following steps: (1) collecting the data of an application for a certain time interval i; (2) splitting the collected data in training and testing datasets using 70-30 split while preserving the chronological order of the requests; (3) training the fingerprint and tests it; (4) if there are no prediction mistakes a success counter *sc* is increased by 1, otherwise *sc* is set to 0; (5) moving to the next interval $i + 1$; (6) finally, when *sc* reaches a convergence threshold K (i.e., the minimum number of consecutive successful iterations), the system stops the training and returns the generated fingerprints for that application. The data collected in each interval is aggregated with the previously collected data from other intervals. Also, because some applications might never reach convergence, we set a maximum number of intervals.

3.2.2 Training Fingerprints. HeadPrint monitors a host traffic and it clusters web requests with "similar" `User-Agent` values, and it applies the aforementioned training process to each cluster. `User-Agent` values are commonly used by benign applications as unique identifiers to be correctly recognized by web servers. Since we assume the training phase to be trusted, no application would actively try to mimic other applications values. Thus, it is likely that requests generated by the same application, within a specific time interval (e.g., training period), have the same, or similar (e.g., version number increased) `User-Agent` value. The training process stops when a fingerprint $F_{a_i} = (\mathtt{VAL}_{a_i}, \mathtt{SEQ}_{a_i})$ is trained for each cluster. The set $F = \{F_{a_i}\}$ represents the set of fingerprints of a machine. HeadPrint requires a specific training timeout that forces it to stop generating new clusters, otherwise, in the unlikely case new applications are regularly installed, HeadPrint would keep producing new clusters, thereby never ending the training. We set this timeout to 5 days.

3.3 Testing New Messages

In anomaly detection we are interested in identifying messages originating from unknown applications. We achieve this by comparing each new message t against all fingerprints in F. If t does not match with any fingerprint, then we consider t to be generated by an unknown application. From now on we refer to these messages as *anomalous* messages. The comparison between a message t and a fingerprint $F_{a_i} \in F$ is achieved by computing $\text{test}(t, F_{a_i}) \in \{\text{True}, \text{False}\}$. The function returns True only if the decision function d, as defined in Section 2.4, returns 1 for the similarity scores between t and F_{a_i}.

Alerts. Due to the high volume of HTTP traffic and its heterogeneity, the system triggers an alert only when it observes anomalous messages transmitting an amount of data greater than a threshold T within a certain period of time (e. g., five days). Anomalous messages are aggregated per destination into clusters. Destinations are represented by domains (i.e., SLD and TLD) and destination IPs. Domain names are identified through the `Host` header field. Each cluster has an attribute describing the amount of data transmitted toward the destination by the messages within the cluster. Every time a new message is added to the cluster, we update the data transmitted with the minimum edit distance between the new message and the previous messages. This way of computing outgoing information was proposed in [5]. Once the data transmitted exceeds a threshold T, an alert is generated containing all the messages as evidence. To avoid duplicated alerts, we verify if the new alerts share messages with previous alerts. If so, we merge the two alerts together. We check both domains and IPs to make it harder for attackers to exfiltrate data without being detected, because it forces them to use multiple domains and IPs if they want to avoid detection. The idea is to deter evasion techniques such as Fast-flux [20].

4 IMPLEMENTATION

We describe the implementation details of HeadPrint, such as how we chose the decision functions based on thresholds and machine learning. We implemented HeadPrint entirely in Python.

Threshold-based Decision Functions. We know that (most of the time) a message obtains a high similarity score when it is tested with the fingerprint of the same application. However, a perfect match does not always occur, because messages may use different headers or values over time. Moreover, we observed that scores from the header-value entropy model are usually lower than the header sequence model. Thus, we verify the performance of HeadPrint using two different threshold-based decision functions: 1) we use 0.7 as value for both thresholds; 2) we use 0.65 for the header value scores and 0.75 for the header sequence score. These values have been chosen after an empirical evaluation on a small subset of data, and they allow us to compare the differences in classification performance by using equal thresholds for both scores, and two different values where the header-value score has a lower threshold.

Machine Learning-based Decision Functions. We followed the steps in Section 2.4 to create a training dataset to train different classifiers, which we use to evaluate our fingerprinting approach. Specifically, we did the following: (1) we randomly selected a subset of 60 hosts out of 302 from our organization dataset, in order to obtain a representative dataset with a diversity of applications that might be analyzed by our tool after being deployed; (2) we used the approach discussed in Section 3.2 to train the fingerprints (see the training setup in Section 5), and we use the remaining data for testing; (3) we tested each message against the host fingerprints, and we labeled the score (x, y) with 1 if the *User- Agent* value in the fingerprint is similar to the one in the message, and 0 otherwise[1]; (4) we grouped together all scores with label 1 and label 0, respectively; (5) we under-sampled the overly represented negative class, which had almost 20 times more datapoints than the positive class, by randomly picking samples with replacement using `RandomUnderSampler` from the `imblearn.under_sampling` Python library; (6) we used the obtained (balanced) dataset of 2M datapoints to train three different classifiers: Adaboost, NearestNeighborsClassifier (with k=5) and SGD[2], using the sci-kit library [23]. We chose these classifiers because they cover both the linear and non-linear case, and they can be trained in a short amount of time despite millions of datapoints. Finally, we note that other classifiers can be used with HeadPrint, as long as they can work with two numerical features.

5 EVALUATION

We evaluated our approach using a dataset, represented as a set of Bro [22] HTTP logs, obtained from the network of an international organization. The logs contain only HTTP metadata, more precisely the HTTP headers without the body of the messages. Only outgoing traffic generated by clients in the network is included; server responses are not present. The dataset contains the network traffic of 302 hosts, from three different subnets, for a period of 40 days, for a total of 3.87 million HTTP requests. The collected traffic was generated only by hosts with static IPs, in order to be able to correctly identify hosts over time. Hosts represent mostly workstations in the premises of the organization, and servers. Traffic has been captured on port 80 towards external network addresses. In other words, HTTP traffic within internal services has been excluded. HTTP messages without headers are filtered out.

5.1 Experiment Setup

To evaluate the fingerprint accuracy and the update resilience, we need a labeled dataset to verify whether messages and fingerprints correspond to the same application. Considering the large size of the dataset, it was not practical to manually label each individual message. Moreover, we could not install an agent on each machine to help labeling the data, because we did not have access to the machines. Thus, we decided to consider a message and a fingerprint to belong to the same application, if their `User-Agent` values are similar. As discussed in Section 3.2, `User-Agent` values are often used as unique identifiers by benign applications. Thus, we use these values to label the traffic of different applications. Although the `User-Agent` is used as groundtruth, fingerprints cannot rely only on its value, because the `User-Agent` value changes over time due to software updates. This would lead to many false alarms, and the solution would not be practical.

In our experiments, we use the following HeadPrint parameters. Regarding the training phase, we use a limit of 20 intervals of 6 hours each for the training (5 days in total), and a convergence threshold K equal to 3. We picked these values after a small-scale empirical evaluation, which showed that these parameters do not significantly affect the performance of our tool. Regarding the header-value entropy model, we performed some empirical tests to

[1] We consider two values to be similar if the Python library function `difflib.SequenceMatcher().ratio` returns a value greater or equal than 0.9. The similarity function is an elaborated version of a pattern- matching algorithm based on the gestalt approach [27].

[2] In our evaluation we have used all default parameters of the scikit library.

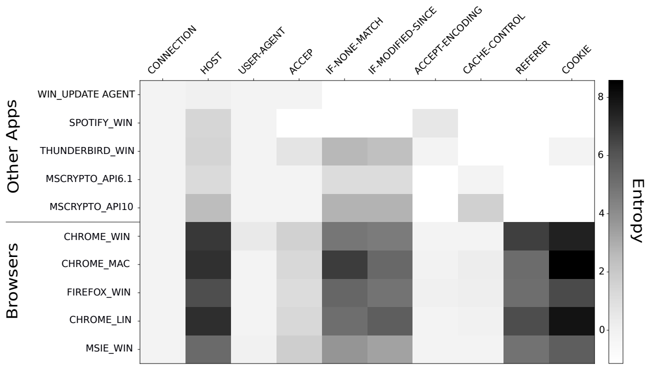

Figure 2: Heatmap representing the entropy of header values for 10 popular HTTP header and 10 different applications, including five different browsers. White cells, where entropy is equal -1, mean that the header has never been used by the application. When entropy is equal 0, it means the application uses a constant value for the header.

choose an α value that is representative for predictable header values. We tested different values and observed which headers where considered predictable. Using domain knowledge we decided to choose $\alpha = 1$, as it returned the most representative results. Overall, our approach has evaluated more than 3.4M messages, and used about 430k messages for training. Please note that in our evaluation HTTP `BODY`, `Method`, `Version` and `URI` are *not* considered HTTP headers. For a fair comparison, since DECANTeR evaluates fingerprints and not individual messages, if a fingerprint is considered a true positive (TP), then we consider the amount of messages observed by DECANTeR to create that specific fingerprint to be TP as well. The same method was applied in [5] to compare DECANTeR to other techniques.

5.2 Characterizing Application Headers

Header Values. Applications associate different values to HTTP headers. Figure 2 shows how 10 distinct applications associate values differently to the headers. The heatmap shows the entropy of header values observed by applications (rows) for common HTTP headers (columns). Blank cells represent an entropy value of -1, which means the header was not observed for that application. We can see that different applications use different headers and their values are more predictable (i.e., light cells) or less predictable (i.e., dark cells). HeadPrint is capable of identifying these differences in headers usage, and it embeds their values in each application fingerprint, thereby achieving a more accurate characterization of the application. Header values may change over time for some application. In case of software updates, headers such as `User-Agent` can change [13, 33]. These updates do not disrupt our approach, in case enough training data is collected, because most of the headers remain the same, yielding still a high similarity score.

Header Sequences. The majority of applications fingerprints in our dataset have a single header sequence, which are on average composed of four or five headers. Another large amount of applications shows fewer than ten header sequences. Since HTTP has a limited set of commonly used headers, this can lead to potential *collisions*, meaning that two or more applications on the same machine share one or more header sequences. Indeed, we found cases of collision in our dataset. Thus, header sequences should be used together with other features to distinguish applications.

Collisions are rare among browsers, even if their fingerprints are characterized by a large number of header sequences. This has two reasons: few browsers are installed on each host, and browsers from different vendors use standard headers (e.g., `Host`, `User-Agent`, `Connection`, `Accept`) in different orders. Hence, header sequences a good metric to distinguish browsers' messages. The reason behind the large number of sequences is twofold: (1) browsers implement complex functionalities, uncommon for other applications, which require special HTTP headers, and (2) web pages can influence (e.g., via JavaScript) the headers of browser messages. Fortunately, these header sequence changes do not disrupt our approach, because unforeseen header sequences for new browser requests show a logarithmic behavior. Thus, the more the browser communicates, the rarer it becomes to find new header sequences. Moreover, if enough samples are included in the fingerprint generation, new header sequences show high similarity scores.

5.3 Fingerprinting Accuracy

We consider a message to be: *true positive* (TP) if the returned fingerprint with maximum score corresponds with the application generating the message; *false positive* (FP), if the returned fingerprint does not correspond with the application generating the message; *false negative* (FN) if no fingerprint is returned (i.e., the decision functions identified all similarity scores to be too low to match any fingerprint), but the fingerprint of the originating application is known; and *true negative* (TN) if no fingerprint is returned, and indeed there is no fingerprint for the originating application because it is unknown (i.e., not present in the training). Precision is computed as $TP/(TP + FP)$, Recall as $TP/(TP + FN)$, and Accuracy as $(TP + TN)/(TP + TN + FP + FN)$.

HeadPrint evaluates a message against multiple fingerprints, and returns only one fingerprint. The system is accurate if it returns the fingerprint of the correct originating application. Thus, HeadPrint can be adjusted to return only the fingerprint that yielded the highest score. Since (1,1) is the maximum score possible, the highest similarity score (x, y) can be defined as the closest point (e. g., using the Euclidean distance) to (1,1). Similarly for DECANTeR, we evaluate the fingerprint with highest similarity score. For a fair comparison, we assume DECANTeR does not rely on the operator to update the fingerprints.

Our evaluation shows that HeadPrint is overall more accurate than DECANTeR. Table 1 shows the classification performance of both HeadPrint, using different classifiers, and DECANTeR. Most importantly, the recall of HeadPrint is on average significantly higher. A low recall indicates that many messages from an application do not match its corresponding fingerprint. This means that DECANTeR cannot consistently identify the traffic of a trained application. The reason behind the low recall is the few hand-picked features DECANTeR fingerprints rely upon. These few features show their limitations in case of software updates, where small changes in the network messages are enough to cause misclassification. Additionally, DECANTeR does not classify correctly the

Table 1: Classification performance of HeadPrint and DECANTeR for the generic use-case of application fingerprint, where we measured if messages were correctly associated with their application fingerprint.

Decision Function	Precision [%]	Recall [%]	Accuracy [%]
Thr. 0.7-0.7	99.20	91.03	91.04
Thr. 0.65-0.75	99,19	93.46	93.21
Adaboost	98.89	96.16	95.44
NearestNeighbor5	98,45	92.16	91.40
SGDHingeL2	99.30	90.68	90.74
DECANTeR	99.62	73.47	75.09

traffic generated by web scripts run in browser. This is a problem that the authors noticed in the original work. HeadPrint does not suffer these issues as much, as shown by the high Recall, because it relies on two orthogonal message characteristics, which overcome the limitations of one another. When the header value changes, the order of the headers remains consistent. Viceversa, when new headers appear and the header sequence is affected, the header values remain rather consistent. Moreover, HeadPrint does not rely on a small set of predefined features to model the message content, but it *automatically* identifies the most characterizing content directly from the traffic of each application.

Although DECANTeR shows an overall higher Precision, meaning that when it associates a message to a fingerprint, the message is more likely to be generated by that application. However, the difference with HeadPrint is small (1.2% for the worst performing model). HeadPrint shows overall a better performance trade-off by showing both a high Recall and Precision, independently from the decision function. This is also reflected by the higher Accuracy, which is often a good overall performance indicator. Adaboost has higher accuracy than other classifiers. Adaboost identifies non-linear patterns that helps it optimizing the classification performance, which is not possible to do with threshold-based functions.

To conclude, this evaluation shows that HeadPrint is an accurate solution for fingerprinting application traffic with an accuracy of 95.44% with the most accurate model. Moreover, HeadPrint is significantly more accurate than DECANTeR, and it is therefore a better approach for passive application fingerprinting.

5.4 Update Resilience.

In this experiment we evaluate how HeadPrint and DECANTeR performs in case of application updates, where the traffic of applications changes over time [13, 33]. We consider only those applications that did not contain an update during their fingerprint generation, but only during testing. Specifically, we consider all messages having a `User-Agent` value different, but similar, (e.g., increased version number) than the one used in training. We then verified whether each message was correctly assigned to its application fingerprint or not. The total number of messages in our dataset generated by updated application is 962,915.

In our analysis, HeadPrint correctly associates the 92% of updated application messages to the originating fingerprinting. This is the best case scenario, represented by the Adaboost classifier. In the worst case scenario represented by the SGD classifier, HeadPrint correctly associates the 75% of updated application messages. On the other hand, DECANTeR can associate only 54% of the messages. The main reason why DECANTeR poorly performs in this task is the few hand-picked features it relies upon, which cannot capture the characteristics of an application after being updated. The reason why HeadPrint is resilient to updates is shown in Figure 3. Software updates permanently change *some* of the header values that are always in application messages, as shown by the red circles. However, updates do not seem to affect the header sequences. HeadPrint correctly identifies update application messages, because the similarity scores remain high (i.e., top-right area of the plane) and the decision functions can make the correct decision.

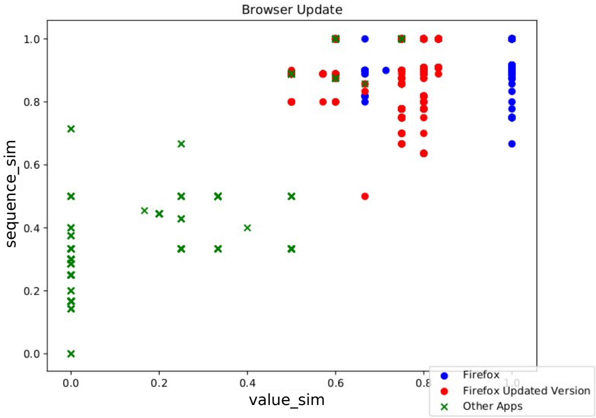

Figure 3: Similarity scores in case of browser update.

Updating HeadPrint's Fingerprints. The fingerprints of HeadPrint can be updated over time, thereby improving their accuracy and resilience against traffic variations. For instance, whenever a message is considered similar to a fingerprint, we can add the header values and the header sequence in the models, and regenerate the fingerprint. However, we decided not to investigate such updating mechanism, because HeadPrint is used in adversarial settings. Thus, an attacker can try to abuse weaknesses in the updating process in order to ensure her evasion. We consider the design of a secure updating mechanism, for a detection model in adversarial setting, as a separate research question.

5.5 Detection Capabilities

We compare the detection capabilities of our approach and DECANTeR, and we assume DECANTeR leverages the help of an operator to update the system. We analyze our dataset using our implementation of HeadPrint, and the publicly available implementation of DECANTeR[3]. Since our dataset was collected from a real network, and not from an experimental setup, it is not labeled. In other words, we did not have access to the groundtruth. Thus, we do not know what connections or hosts are malicious or benign. This is a typical setting for these experiments on real-world traffic. Hence, we decided to label the dataset with the help of a threat intelligence provider. We match the URLs and destination IPs of

[3] https://github.com/rbortolameotti/decanter

our dataset against their indicators of compromise (IOCs). IOCs is information extracted by (often automated) large scale malware analyses. Messages that match an IOC are marked as suspicious, benign otherwise. Suspicious messages are removed from the training set. Finally, we manually analyze, with the help of the threat intelligence provider, the alerts generated by the two systems.

We acknowledge that our labeling approach can neither guarantee the training set to be uncorrupted nor guarantee that all infected hosts are correctly identified, because threat intelligence is not perfect. We inspected the messages matching the IOCs to the best of our abilities, and we found no clear evidence of malicious communication, but only suspicious network behavior. We consider our labeling approach a *best-effort* to analyze our solution with a dataset of a real-world network traffic.

Alert Thresholds. The results in Table 2 show that for HeadPrint, the higher the threshold the smaller is the number of alerts. Adaboost shows better performance than other classifiers. For Adaboost, the lowest threshold triggers 2,355 alerts, which represents 1.04% of all messages. With higher exfiltration thresholds, such as 5,000 or 7,500 bytes, the number of alerts decreases to 345 (0.3%) and 212 (0.2%), respectively. This trend holds for all classifiers. The percentage of messages considered anomalous is below 0.5% for rather small thresholds. For example, if we consider Adaboost with a threshold of 5KB for five days of exfiltration, we would obtain 345 alerts over 35 days (10 alerts per day on average), which is significantly lower than the average 80 alerts per day of DECANTeR.

The threshold does not affect as much the number of alerts generated by DECANTeR, because DECANTeR has additional checks that trigger alerts independently from the threshold. These checks often generate false alerts when DECANTeR mislabels benign browser traffic. Table 2 shows that even with considerable human effort in updating the system, *DECANTeR generates more (false) alerts (i.e., false positives) compared with HeadPrint. This result is the direct consequence of not being able to correctly identify applications traffic, and it is represented by the Recall value in Table 1.*

Detection of Suspicious Hosts. We manually inspected the alerts generated by HeadPrint and DECANTeR. For HeadPrint we analyzed the scenario that has Adaboost as classifier and a threshold of 5KB. We opted for this threshold because it is low enough to potentially detect exfiltration attempts. To check whether our alerts actually came from infected machines, we leveraged two sources of threat intelligence: VirusTotal and a threat intelligence provider. The former did not show any malicious behavior, whereas the latter provided us with 18 indicators of compromise for our dataset. Note that an indicator of compromise *may* indicate a compromise.

Table 2: Number of alerts generated by different classifiers.

Decision Function	Alert Thresholds (kB)							
	1	**2.5**	**5**	**7.5**	**10**	**20**	**50**	**100**
T. (0.7,0.7)	4343	1428	746	476	346	154	44	18
T. (0.65,0.75)	3296	1117	580	354	262	113	33	12
Adaboost	**2355**	**721**	**345**	**212**	**157**	**67**	**17**	**9**
NearestNeigh.	4218	1288	588	375	270	109	30	11
SGDHingeL2	4543	1474	759	486	350	153	46	17
DECANTeR	2861	2739	2666	2609	2580	2544	2512	2486

The indicators provided by the third-party company involved requests generated by 13 different hosts. For 5 of these hosts we were able to confirm that the indicators were false positives. For the other 8 hosts, we inspected the alerts generated by both HeadPrint and DECANTeR. For HeadPrint we found evidence of suspicious behavior of 2 of these hosts, while with DECANTeR we found 3 suspicious hosts. These hosts generated traffic associated with browser hijackers or generic adware. The alerts included HTTP requests transmitting several thousands bytes encoded in the URI. For the 5-6 remaining hosts flagged by IOCs, we did not find any evidence of suspicious behavior, but we cannot exclude that they were infected. After analyzing the indicators, we inspected all the remaining alerts. For HeadPrint we identified 3 other hosts showing suspicious connections, which did not show suspicious behavior according to the indicators. One of these hosts showed highly suspicious behavior (i.e., unknown software uploading 50Kb-100Kb of data toward cloud services), which certainly worth of extra investigation. On the other hand, DECANTeR found 5 hosts showing anomalous connections, such as sequences of POSTs to suspicious destinations, and large quantities of anomalous fingerprints from the same host but with many different (and inconsistent) `User-Agent` values.

Overall, both systems show similar capabilities in detecting anomalous HTTP traffic, as it was reflected by the TNR in Table 1. Although DECANTeR identifies more anomalies than HeadPrint with Adaboost, which is an important aspect of NIDS, it also generates more false alerts, which is also a crucial aspect in NIDS [30]. In practice, a NIDS runs multiple detectors, thus it is important that each detector does not trigger many false alerts to avoid *alert fatigue*. Thus, our approach provides a better performance trade-off for a practical solution, because *HeadPrint detects anomalous connections while triggering significantly fewer false alerts and without requiring any human intervention.*

Malware Experiment. We evaluate HeadPrint with dataset containing data exfiltrating malware (DEM) provided by DECANTeR, in order to show the capabilities of HeadPrint to identify malicious connections. We followed the same steps discussed in [5]: 1) we trained the fingerprints from traffic generated by a (non-infected) VM, and 2) we tested the traffic of malicious samples, executed in the same VM, against the trained fingerprints.

In this analysis HeadPrint detected 78% (i.e., 49 samples out of 59) of malicious samples. 13 samples were missed because they did not generate enough data to reach the threshold of 1Kb needed to trigger the alert. More importantly, malicious messages did not match the trained fingerprints. Thus, the reason why HeadPrint missed the samples was due to the short-lived execution of malware in the virtual environment (i.e., little data was generated). This result confirms the ability of HeadPrint to detect malicious connections.

5.6 Detection of Mimicry Attempts

We consider the scenario where a malware generates network messages following a predefined format that correctly uses header sequences and values of a known application. In this experiment we assume malware tries to mimic browser messages, since they are common applications and generate heterogeneous traffic. We assume that malware implements these formats before infecting its victim. Messages contain header values that are common for each specific header (e.g., `User-Agent` string common during the time of data capture). We compare the mimicry messages against the fingerprints of each host, and we consider the mimicry to be successful if the message matches with at least one fingerprint.

Table 3 shows the percentage of hosts for which HeadPrint would have identified the mimicry attempt. The results show that HeadPrint works against this type of mimicry attacks. First, even if the mimicked message represents exactly the format of a known application, if the target application of the mimicry is not installed, then mimicked messages are flagged as anomalous. Second, the values used in the mimicked message headers may differ from those used by the same browser of the compromised system. For example, the user may run an old version of an application which the malware did not expect.

6 LIMITATIONS

Encryption. HeadPrint does not work on encrypted data, because the header and their values cannot be observed. Therefore, HeadPrint can analyze all web traffic in network environments, such as enterprises, where TLS-proxies are available. Although TLS-proxies are not an optimal solution due to their security and privacy concerns [11], they are still being deployed in enterprise environments to enhance network visibility and to protect the most sensitive parts of their networks, by inspecting the traffic with advanced security solutions. Finally, HTTP is still a commonly used protocol for malicious communications [19, 31, 38], therefore, HeadPrint can still be used to analyze plaintext traffic to detect such threats.

Evasion. HeadPrint is capable of detecting mimicry attempts where the attacker chooses a priori the characteristics of the application to mimic. However, the attacker can evade detection by exfiltrating data very slowly, such that the threshold is never exceeded. This limitation is inherent in any threshold-based detection system. The problem of detecting low-throughput data exfiltration (i.e., low throughput covert channels) is known to be particularly hard to solve [37], because the attacker has a large number of options to hide its data within the messages.

Passive application fingerprinting (PAF) assumes that malicious communications show deviating network characteristics from the application installed on the monitored hosts. A piece of malware can break the fundamental assumption of PAF, by creating network messages that are not distinguishable from the installed applications. This can be achieved either by using an installed application (e.g., headless browser) to communicate, or by mimicking the traffic observed from installed applications. The former requires specific applications to be installed, thus it cannot be applied to any scenario (e.g., browser may not be installed on a server). The latter requires network card access to read the traffic of the compromised system. Both these techniques break the assumption behind PAF. This limitation affects *any* anomaly detection system that assumes that malicious messages show different characteristics from the normal traffic of the infected machine [3, 5, 29]. Although these techniques are difficult to detect for anomaly-based NIDS, they can still be mitigated using complementary solutions [6].

Table 3: Percentage of hosts where a mimicry would fail.

Classifiers	Mimicry Failure [%]		
	Chrome	Firefox	Safari
Threshold 0.7-0.7	97.68	87.09	80.46
Threshold 0.67-0.75	74.83	80.13	74.83
Adaboost	76.82	64.24	57.95
NearestNeighbors5	76.82	67.22	59.93
SGDHingeL2	98.68	85.76	80.46

7 RELATED WORK

Mobile Application Fingerprinting Several works have been proposed to classify mobile apps traffic. Dai et al. [10] proposed an emulator to analyze mobile apps, which also extracts app fingerprints combining invariant strings in URLs and domains. Xu et al. [34] introduced a similar approach that uses a concatenation of hostname and key-value pairs in the query of URI to generate fingerprints. Moreover, their approach improves the fingerprints by adding information as soon as new patterns of the same app are identified. Miskovic et al. [18] introduced AppPrint, which similarly evolves its fingerprints, but it uses different application identifiers to generate the seeding fingerprint and a different method to correlate new flows belonging to the application. Yao et al. [35] proposed to fingerprint applications using a set of rules that combines application identifiers and their position within headers. Taylor et al. [32] proposed AppScanner, an approach that leverages statistical features about applications flows to classify applications traffic, thereby extending their classification to encrypted traffic.

These fingerprinting techniques are not passive, because they require a controlled environment to generate the application traffic and create the fingerprints from it. Therefore, they are not suitable for passive application fingerprinting. Furthermore, the features used in mobile application fingerprinting are not suitable to model common workstation applications such as browsers (e.g., features based on URIs and hostname), and other chosen features (e.g., advertisement identifiers) can be found only in mobile traffic.

Browser Fingerprinting Browser fingerprinting, which goal is mainly to track web users across the Internet, has also received a lot of attention in research [2, 7, 12, 14, 15]. Most of these fingerprinting techniques require *active* fingerprinting, meaning that a server crafts a specific response in order to identify patterns that may identify a specific browser. For example, JavaScript code can be embedded in the page to retrieve extra information from the application or the host to make the fingerprint unique.

These techniques are not applicable for two reasons: firstly, they cover only browsers which is a subset of applications running on workstations, secondly in our network monitoring use-case the fingerprint has to work passively, meaning that there is no possibility to query applications for extra information.

Other fingerprinting techniques Automatic malware signatures generation is another variant of well studied "application" fingerprinting [21, 24, 26, 36]. Essentially, malware are applications and signatures are characteristics of the network behavior of such applications. The goal of these techniques is to identify matching requests that are certainly generated by malicious software, but not *all* requests, which is our goal.

OpenAppID [16] is an application detection language provided by Cisco [17] and used in popular IDSs such as SNORT [28], which provides rules to identify clients, web application, and application protocols. However, OpenAppID cannot be used in our setting because it does not automatically infers fingerprints from observed traffic, but it requires administrators to manually generate the signatures to recognize the applications. Moreover, these signatures heavily rely on the `User-Agent` value. Therefore, the rules are not robust to applications variations such as updates, and likely to generate false positives in the setting of anomaly detection.

8 CONCLUSIONS

In this work we proposed HeadPrint, a novel approach for passive application fingerprinting examining protocol headers to differentiate between various applications. Overall, HeadPrint achieves overall better performance than the current state-of-the-art, while not suffering from the same practical limitations. Specifically, our method shows capabilities in detecting malicious communications, while raising significantly fewer alerts and not requiring human intervention to be maintained. Thus, HeadPrint brings PAF for anomaly detection closer to practice. Lastly, although HeadPrint has been evaluated on HTTP, its underlying fingerprinting technique is protocol agnostic and may be applied to other protocols as well (e.g., TLS Handshake Protocol). We consider the applicability of HeadPrint to other protocols as future work.

REFERENCES

[1] Daniel Bakkelund. 2009. An LCS-based string metric. *Olso, Norway: University of Oslo* (2009).

[2] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. 2011. User tracking on the web via cross-browser fingerprinting. In *Nordic Conference on Secure IT Systems*. Springer, 31–46.

[3] Kevin Borders and Atul Prakash. 2004. Web tap: detecting covert web traffic. In *Proc. of the conference on Computer and Communications Security*.

[4] Shyam Boriah, Varun Chandola, and Vipin Kumar. 2008. Similarity Measures for Categorical Data: A Comparative Evaluation. In *Proc. of the International Conference on Data Mining*.

[5] Riccardo Bortolameotti, Thijs van Ede, Marco Caselli, Maarten H Everts, Pieter Hartel, Rick Hofstede, Willem Jonker, and Andreas Peter. 2017. DECANTeR: DEteCtion of Anomalous outbouNd HTTP TRaffic by Passive Application Fingerprinting. In *Proc. of the ACM Annual Computer Security Applications Conference*.

[6] Riccardo Bortolameotti, Thijs van Ede, Andrea Continella, Maarten Everts, Willem Jonker, Pieter Hartel, and Andreas Peter. 2019. Victim-Aware Adaptive Covert Channels. In *Proc. of the Conference on Security and Privacy in Communication Networks (SecureComm)*. Orlando, FL.

[7] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *Annual Network and Distributed System Security Symposium (NDSS)*.

[8] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[9] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. 2007. Traffic classification through simple statistical fingerprinting. *Computer Communication Review* 37, 1 (2007), 5–16.

[10] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. 2013. Networkprofiler: Towards automatic fingerprinting of android apps. In *Proc. of the IEEE INFOCOM Conference*.

[11] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. 2017. The Security Impact of HTTPS Interception. In *Proc. of the Annual Network and Distributed System Security Symposium (NDSS)*.

[12] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.

[13] Electronic Frontier Foundation. [n. d.]. Kaspersky User-Agent Strings - NSA. ([n. d.]). https://www.eff.org/it/node/86529

[14] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1388–1401.

[15] David Fifield and Serge Egelman. 2015. Fingerprinting Web Users Through Font Metrics. In *Proc. of the Financial Cryptography and Data Security International Conference (FC)*.

[16] Martin Roesch. [n. d.]. Cisco Announces OpenAppID, the Next Open Source Game Changer in Cybersecurity. ([n. d.]). https://blogs.cisco.com/security/cisco-announces-openappid-the-next-open-source-game-changer-in-cybersecurity

[17] Martin Roesch. [n. d.]. Firepower Management Center Configuration Guide. ([n. d.]). https://www.cisco.com/c/en/us/td/docs/security/firepower/610/configuration/guide/fpmc-config-guide-v61/application_detection.html?bookSearch=true

[18] Stanislav Miskovic, Gene Moo Lee, Yong Liao, and Mario Baldi. 2015. AppPrint: automatic fingerprinting of mobile applications in network traffic. In *International Conference on Passive and Active Network Measurement*. Springer, 57–69.

[19] MITRE. [n. d.]. Commonly Used Ports, MITRE. ([n. d.]). https://attack.mitre.org/techniques/T1043/

[20] Jose Nazario and Thorsten Holz. 2008. As the net churns: Fast-flux botnet observations. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*. IEEE, 24–31.

[21] Terry Nelms, Roberto Perdisci, and Mustaque Ahamad. 2013. ExecScent: Mining for New C&C Domains in Live Networks with Adaptive Control Protocol Templates. In *Proc. of the USENIX Security Symposium*.

[22] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23-24 (1999), 2435–2463.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[24] Roberto Perdisci, Wenke Lee, and Nick Feamster. 2010. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*. 391–404.

[25] Ponemon Institute. [n. d.]. 2018 Cost of a Data Breach Study by Ponemon. ([n. d.]). https://www.ibm.com/security/data-breach

[26] M Zubair Rafique and Juan Caballero. 2013. Firma: Malware clustering and network signature generation with mixed network behaviors. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 144–163.

[27] John W Ratcliff and David E Metzener. 1988. Pattern-matching-the gestalt approach. *Dr Dobbs Journal* 13, 7 (1988), 46.

[28] Martin Roesch. 1999. Snort: Lightweight Intrusion Detection for Networks. In *Proc. of the Conference on Systems Administration (LISA-99), Seattle, WA, USA, November 7-12, 1999*. 229–238.

[29] Guido Schwenk and Konrad Rieck. 2011. Adaptive detection of covert communication in http requests. In *Computer Network Defense (EC2ND), 2011 Seventh European Conference on*. IEEE, 25–32.

[30] Robin Sommer and Vern Paxson. 2010. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proc. fo the IEEE Symposium on Security and Privacy (S&P)*.

[31] Aditya K. Sood, Sherali Zeadally, and Richard J. Enbody. 2016. An Empirical Study of HTTP-based Financial Botnets. *IEEE Trans. Dependable Sec. Comput.* 13, 2 (2016), 236–251.

[32] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. 2016. Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic. In *Proc. of the IEEE European Symposium on Security and Privacy*.

[33] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. FP-STALKER: Tracking Browser Fingerprint Evolutions. In *IEEE S&P 2018-39th IEEE Symposium on Security and Privacy*. IEEE, 1–14.

[34] Qiang Xu, Yong Liao, Stanislav Miskovic, Z Morley Mao, Mario Baldi, Antonio Nucci, and Thomas Andrews. 2015. Automatic generation of mobile app signatures from traffic observations. In *Proc. of the IEEE INFOCOM Conference*.

[35] Hongyi Yao, Gyan Ranjan, Alok Tongaonkar, Yong Liao, and Zhuoqing Morley Mao. 2015. Samples: Self adaptive mining of persistent lexical snippets for classifying mobile application traffic. In *Proc. of the Annual International Conference on Mobile Computing and Networking*. ACM, 439–451.

[36] Ali Zand, Giovanni Vigna, Xifeng Yan, and Christopher Kruegel. 2014. Extracting probable command and control signatures for detecting botnets. In *Symposium on Applied Computing, SAC*.

[37] Sebastian Zander, Grenville J. Armitage, and Philip Branch. 2007. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys and Tutorials* 9, 1-4 (2007), 44–57.

[38] Apostolis Zarras, Antonis Papadogiannakis, Robert Gawlik, and Thorsten Holz. 2014. Automated generation of models for fast and precise detection of HTTP-based malware. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*. IEEE, 249–256.