Abdullah Qasem, Sami Zhioua*, and Karima Makhlouf

Finding a Needle in a Haystack: The Traffic Analysis Version

Abstract: Traffic analysis is the process of extracting useful/sensitive information from observed network traffic. Typical use cases include malware detection and website fingerprinting attacks. High accuracy traffic analysis techniques use machine learning algorithms (e.g. SVM, kNN) and require to split the traffic into correctly separated blocks. Inspired by digital forensics techniques, we propose a new network traffic analysis approach based on similarity digest. The approach features several advantages compared to existing techniques, namely, fast signature generation, compact signature representation using Bloom filters, efficient similarity detection between packet traces of arbitrary sizes, and in particular dropping the traffic splitting requirement altogether. Experimental results show very promising results on VPN and malware traffic, but low results on Tor traffic due mainly to the single-size cells feature.

Keywords: Traffic Analysis, Website Fingerprinting, Malware Clustering

DOI 10.2478/popets-2019-0030
Received 2018-08-31; revised 2018-12-15; accepted 2018-12-16.

1 Introduction

Network traffic analysis is the process of using metadata such as packet sizes and timings in order to retrieve useful/sensitive information about network communications. As opposed to traditional content-based techniques, traffic analysis does not rely on the content of packets which makes them increasingly relevant given the widespread use of encryption and obfuscation.

Abdullah Qasem: Concordia University, E-mail: a_qas@encs.concordia.ca
***Corresponding Author: Sami Zhioua:** King Fahd University of Petroleum and Minerals, E-mail: zhioua@kfupm.edu.sa
Karima Makhlouf: Imam Abdulrahman Bin Faisal University, E-mail: kmakhluf@iau.edu.sa

A large body of work in traffic analysis focused on applying machine learning techniques to network traffic in order to extract patterns from the traffic traces. These patterns can then be used to recognize similar traces. On the positive side, such techniques could reach very high classification accuracy in closed-world as well as in open-world scenarios. On the downside, the existing techniques are not suitable for telling if a large traffic trace contains the trace of a specific item (finding a needle in a haystack). To illustrate this limitation, consider the following common scenario. An entity through which a huge amount of traffic is flowing (e.g. ISP) is trying to pinpoint a specific artifact (e.g website, malware, etc.) in the traffic and in real-time. Existing traffic analysis techniques are not suitable for such scenario because they work by matching each part of the traffic with each signature. The signature matching is based on comparing "apples to apples". That is, extracting each time a limited sequence of packets and matching them with the signature. For example, if a signature is generated from n packets, at each step, only around n packets should be extracted from the traffic and matched with the signature. In presence of a large number of artifacts (A bank of malware, a bank of websites, etc.) with their signatures, keeping up with the real-time intake becomes quickly infeasible.

Unlike the majority of existing work in the literature which apply machine learning techniques, we apply techniques from digital forensics. The intuition behind our approach is that some algorithms from digital forensics try to solve problems very similar to traffic analysis problems. A common scenario in data forensics is to check if a large search space (e.g. hard disk) contains traces of source data (e.g. a picture file). This scenario has natural resemblance with the problem of checking if a large amount of network traffic contains traces of a specific artifact (e.g. website, malware).

In this paper we use approximate hash based matching (AHBM), a recent and flexible approach for similarity identification which can accommodate various modifications in the source data (insertion, deletion, reordering, etc.). AHBM tries to identify statistically-improbable chunks of bytes and use them to compute the similarity between two sources of data. Unlike sim-

ple hash based techniques, which support yes/no results, AHBM allows requests to be answered approximately, that is, with a value between 0 and 100.

The main contributions of the paper are:

1. Dropping the splitting pre-processing step required in existing machine learning traffic analysis approaches.
2. A generic traffic analysis approach that works for website fingerprinting and for malware detection.
3. A malware traffic clustering algorithm using similarity hashing.

2 Related Work

Several website fingerprinting attacks are reported in the literature [1–7]. While older attacks are relatively slow [3–5], more recent ones are significantly more scalable [6, 7]. Recently, Nasr et al. [8] addressed the scalability issue using a different approach, namely, compression. Traffic features are compressed using linear projection algorithms and compressed sensing and then used without decompression. Overall, the compressive traffic analysis approach achieved a 13 times effective speed up compared to Wang et al.'s k-NN based attack [5] and a 3 times effective speed up compared to Panchenko et al.'s SVM attack [6], both with a negligible accuracy loss. However, all these approaches achieve high accuracy results under the important assumption that a reliable splitting procedure is used. Without such splitting procedure, all these approaches are hardly applicable in real scenarios [9, 10]. To identify a website occurrence within a long sequence of captured packets, there are currently two main approaches: splitting the traffic and using a sliding window. Wang and Goldberg [11] discussed the splitting problem and proposed two ways of implementing it: time-based splitting and classification-based splitting. Time-based splitting is efficient provided that the time-gap separating two website visits is large. If the time-gap is small, classification-based splitting is used which has an additional processing overhead. Either ways, the website fingerprinting attack needs to go through an initial splitting pre-processing step. Moreover, even with an accurate and efficient splitting procedure, each website model has to be matched with each separated session. Our proposed AHBM approach outperforms this process by completely dropping the need for splitting the traffic.

To address the splitting problem, Feghhi and Leith [12] adopt a sliding window approach with a 10 packets moving step. At each step, a sequence of n packets is extracted from the traffic and matched with the website model where n is the size of samples used in the training phase. However, even with a moving step of 10 packets, this approach is clearly not scalable in realistic scenarios.

3 Approach Motivation

Finding similarities between data objects [1] is a typical problem in digital forensics. A typical scenario consists in looking for a similarity between a reference data object (image or office document) and a target data object under investigation (live memory dump, captured traffic dump, etc.). Compared to existing network traffic analysis techniques, the ones used in digital forensics are simpler but more scalable. It is important to note that digital forensics techniques are based on string comparison where each data object is considered as a string of bytes. Hence measuring similarity is based on string matching.

Hashing is a very common forensics tool for string matching. Simple hashing, regardless of its granularity, has two limitations. First, it allows to pinpoint occurrences of exact copies. Any alteration (even 1 bit) yields a hash mismatch. This problem is also known as hash fragility. Second, hashed blocks should be aligned the same way in the reference and target data objects. Any displacement in block boundaries yields a hash mismatch.

To address the hash fragility problem, data fingerprinting [13] is typically used to find similar objects instead of exact object copies. The idea is to select a set of representative features for each object, then the similarity is computed in terms of the level of correlation between the features. Data fingerprinting is known to be resilient to small alterations.

To address the block alignment problem, k-gram and Winnowing [14] are used. That is, each data document is divided into contiguous and overlapping substrings of size k called k-grams. Each position of the data document is a starting point of a different k-gram. A hash is computed for each k-gram and a subset of these hashes is selected as the document's fingerprint. Winnowing is an efficient algorithm for selecting k-gram

1 In this discussion, we refer to the first and second data object as *reference and* target respectively.

hashes which consists in using a sliding window across the data document and keeping only one representative from each window position.

Using data fingerprinting and Winnowing raises three important questions: (1) how features are selected from a data object? (2) how features are stored efficiently? and (3) how the correlation/similarity between two sets of features is measured?

In the context of traffic analysis, we define a data object as a sequence of packets corresponding to a specific network activity (e.g. website visit, malware communication session, etc.). For each sequence, only the following information is kept: packet lengths, packet order, and packet directions. Since the approach is mainly based on string matching, this "meta-data" information is sufficient to characterize similarity. A data feature[2] is a sub-sequence of packets in a data object. Figure 1 shows a snippet of a data object with the first three data features.

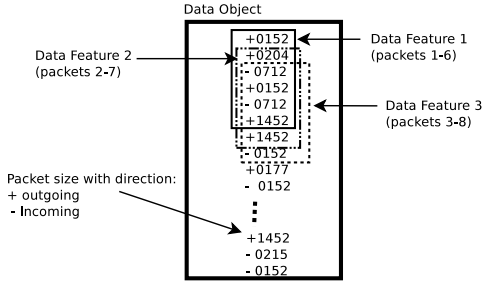

Fig. 1. Data Object and Data Feature.

The proposed traffic analysis technique is based on data fingerprinting and winnowing and is inspired by Roussev's approach of data fingerprinting with similarity digests [15, 16]. Using this approach, the short answers to the above questions are as follows. Given a data object (e.g. samples of a website visit), the technique aims at selecting features (Question 1) that are least likely to occur in other data objects (e.g. samples of a different website visits) by chance. The selected features are then hashed and stored using Bloom filters (Question 2) which allow significantly compact representation and fast membership queries. Hence, each data object will be represented by a Bloom filter. Measuring the correlation/similarity of two data objects (Question 3) consists in comparing their Bloom filter representations.

2 In the next sections we refer to data feature as block. Both terms are used interchangeably

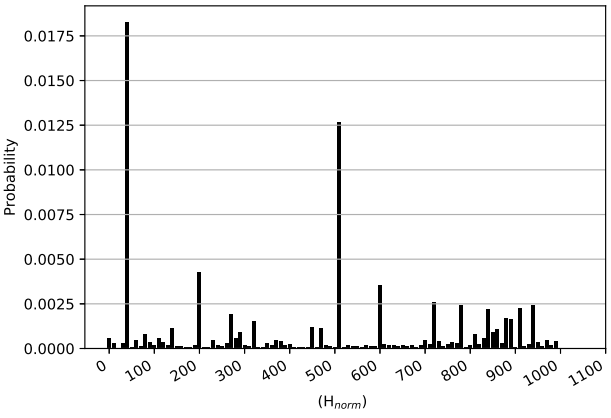

Fig. 2. Probability density function of normalized hash values of features (size 3) for malware dataset. A short bar indicates uncommon features, while a tall bar indicates common ones.

4 Hash-based Signature Generation

The criteria we use for feature selection is the following: choosing uncommon (statistically improbable) features that happen to be part of the available samples.

In order to characterize statistically improbable features, we compute the hash digest (SHA-1) of each feature. The resulting 160-bits digest is then normalized to a value between 0 and 1023 by considering only the 10 rightmost bits of the digest:

$$H_{norm}(F) = int(leftmost10bits(SHA1(F))) \quad (1)$$

4.1 Feature Selection Criteria

Let B be the size (number of packets) of a data feature. For example, in Figure 1, B is 6. A data object of length L packets contains $L - B + 1$ features. The data fingerprinting approach consists in selecting only a subset of these features to uniquely identify the data object. A good feature candidate for selection should satisfy two criteria:

– criteria 1: it should be statistically improbable
– criteria 2: it should be part of most of (or all) data samples

For criteria 1, the statistical probability of a feature depends on the likelihood of its normalized hash value. To this end, we estimate the probability distribution of all possible normalized hash values $[0, 1023]$. For each

feature in the data object, we compute its normalized SHA-1 digest and use the value to identify its statistical probability from the distribution. This is implemented by a rank table of size 1024 mapping normalized hash values to a precedence score (S_{prec}) which is a value between 0 and 1023 proportional to the probability value. It is important to note that a different rank table is generated for each type of traffic. Figure 2 shows a graphical representation of the rank table corresponding to malware dataset.

For every feature, the precedence score indicates how common the corresponding normalized hash value is found in the empirical data. That is, if a feature has a very common H_{norm} value, it gets a high S_{prec} value. Whereas features with uncommon H_{norm} values will get low S_{prec} values. Selecting the statistically-improbable features consists in picking the features with the lowest S_{prec} values. A straightforward approach consists in ordering all the data features across the data object in ascending order of their S_{prec} values and choosing the top ones. The problem is that all selected features will be originating from the same region/cluster of the data object. Winnowing [14] technique uses a sliding window to pick features with local minima. This guarantees that features are selected from various locations in the data object.

Let W be the window size. While the window is sliding across the data, a feature with the lowest S_{prec} is marked[3] at every step. If the same feature is marked a number of times ($1 \leq k \leq W$), it can be considered as a feature with a local S_{prec} minimum and consequently selected as statistically improbable. This is achieved by maintaining a popularity counter score S_{pop} which keeps track of how many times each feature has been marked due to a minimum S_{prec}.

The actual feature selection is based on setting a threshold value t ($1 \leq t \leq W$) such that any feature with a popularity score $S_{pop} \geq t$ is selected as statistically-improbable. In our case, we use a threshold value $t = 2$.

Figure 3 shows how S_{pop} values are updated while a sliding window of size 9 is moving through a data input sample. The upper row shows the H_{norm} values. After the sliding window reaches the end of the data object, the final S_{pop} values (step 12) are used to select the statistically-improbable features. For instance, if the threshold $t = 2$, three features will be selected (positions 4, 8 and 17) whereas for $t = 4$, only two features will be selected (positions 8 and 17).

Once a feature is considered statistically improbable (passed the winnowing process), it passes through the second filter (criteria 2) which checks if the feature occurs in the other samples of the same class. Keeping only the features that occur in all other samples is too restrictive and result in a small number of features. On the other hand, allowing all features that occur in at least half of the samples is too permissive and result in a large number of features. Empirical analysis showed that the best threshold value is to keep only features that appear in at least 75% of the other samples.

4.2 Building Fingerprint Representations

Having selected a set of features to represent a data object, the next step is to build a fingerprint representation out of the features. A simple approach consists in computing the hash of each feature and concatenating the hashes to obtain a fingerprint representation of the data input. This approach is inefficient in two aspects. It is space inefficient and it handles membership queries (checking if an element is part of the set) inefficiently. Note that this simple approach does not incur any false positive rate for membership queries. A more efficient approach for set representation is using Bloom filters [17]. The approach trades space and membership queries efficiency for a small false positive rate in membership queries.

A Bloom filter is an array of bits of a fixed size m initially set to 0. Given a set of elements $\{s_1, s_2, \ldots, s_n\}$, inserting an element s_i in a bloom filter consists in computing hash values of s_i according to k different hash functions h_1, h_2, ... h_k. Each hash function maps the universe of possible element values to an index in the bloom filter (i.e. in the range $1 \ldots m$)[4]. For each of those hash values, the Bloom filter entry is set to 1. Checking the membership of an element s_j goes through the same process, that is, computing the k indices using the k hash values. If all the bits on the k indices are equal to 1, the membership query returns true. Otherwise, it returns false. As mentioned above, Bloom filter approach for set representation trades efficiency for a small false positive rate in membership queries. Indeed, in membership queries, if at least one entry of the computed k

3 If two features happen to have the same S_{prec} value in the current window, only the leftmost one is marked.

4 The hash functions are independent and map the input to the range $1 \ldots m$ uniformly.

Position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
H_{norm}		460	557	580	510	575	805	575	515	575	566	568	566	515	566	568	562	460	566	575	575
Step 1	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 2	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 3	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 4	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 5	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Step 6	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
Step 7	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0
Step 8	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0
Step 9	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	4	0	0	0	0	0	0	0	0	1	0	0	0
Step 10	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	4	0	0	0	0	0	0	0	0	2	0	0	0
Step 11	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	4	0	0	0	0	0	0	0	0	3	0	0	0
Step 12	S_{prec}	520	610	600	592	604	612	604	596	604	614	628	614	596	614	628	638	520	614	604	604
	S_{pop}	1	0	0	3	0	0	0	4	0	0	0	0	0	0	0	0	4	0	0	0

Fig. 3. Computing the popularity score (S_{pop}) on a data sample of size 20 packets and using a sliding window of size 9.

indices is set to 0, we are sure that the element is not in the set. However, if all the k entries are set to 1, we are not 100% sure that the element is in the set. The k entries might be set by chance (while inserting other features). The good news is that, given the number of elements already inserted in the Bloom filter, the false positive rate can be easily estimated [17].

The procedure to insert a feature into a Bloom filter is illustrated in Figure 4. First, the sequence of packets (packet lengths and direction) composing the feature are hashed using SHA-1 algorithm which generates 160 bits. The hash is then split into five 32 bits sub-hashes. Each sub-hash is considered as different hash value. Hence, k value in this case is 5. The 11 least significant bits of each 32 bits sub-hash is used as index in the Bloom filter array of bits. The features are inserted in 256-byte Bloom filters ($m = 256 \times 8 = 2048 = 2^{11}$). The values of k (number of sub-hashes), m (the size of the bloom filter), and n (the maximum number of features to be inserted in a bloom filter) are chosen such as to keep the false positive rate f[5] under an acceptable low threshold. f can be estimated as follows [17]:

$$f = (1 - e^{-\frac{kn}{m}})^k \tag{2}$$

The capacity of a single Bloom filter is fixed to a default value of $n = 128$ features. Based on (2) and the chosen values ($k = 5$, $m = 2048$, and $n = 128$), the expected false positive probability is 0.00139. When the capacity of a bloom filter is reached, a new Bloom filter is created, and so on, until all features are represented.

The fingerprint representation of a data object is the concatenation of all Bloom filters preceded by their total number. The obtained representation is called similarity digest (SD):

$$SD(dob) = s|bf^1|bf^2|bf^3|\dots bf^s \tag{3}$$

where s is the number of Bloom filters and bf^i is the i^{th} Bloom filter of data object *dob*.

4.3 Comparing Similarity Digests

Computing the similarity between two data objects consists in comparing their Bloom filters. Hence, the core of the similarity computation process is the comparison of two Bloom filters which is based on a modified version of Hamming distance. We call it *Hamming Similarity*.

Let bf_1 and bf_2 be two Bloom filters where bf_1 is chosen to be the one with less features and consider the following variables:

- n_1 and n_2: the number of features in bf_1 and bf_2 respectively[6].
- e_1 and e_2: the number of bits set to one in bf_1 and bf_2 respectively.
- e_{12}: the number of bits set to one in both bf_1 and bf_2 (intersection).

The Hamming Similarity (HS) between two Bloom filters is defined as follows:

$$HS(bf_1, bf_2) = \begin{cases} -1, & \text{if } n_1 \leq N_{min} \\ 0, & \text{if } e_{12} \leq t_c \\ 100\ \dfrac{e_{12} - t_c}{E_{max} - t_c}, & \text{otherwise} \end{cases} \tag{4}$$

where,

5 The probability that all bits corresponding to a feature are set to one while the feature is not in the bloom filter

6 Maintained as separated counters.

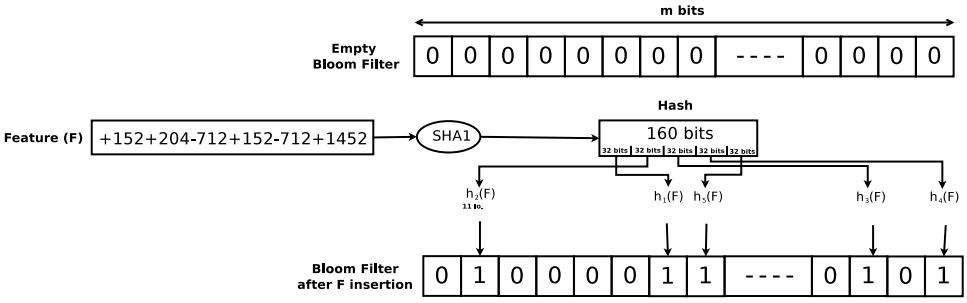

Fig. 4. Process of inserting a feature into a Bloom filter

- N_{min} is the minimum number of features in a Bloom filter that is needed to compute a meaningful HS value[7].
- t_c is a threshold below which any bit matching is assumed to be due to chance:

$$t_c = \alpha(E_{max} - E_{min}) + E_{min} \quad (5)$$

- E_{max} is the maximum number of matching bits due to chance:

$$E_{max} = \min(kn_1, kn_2) \quad (6)$$

- E_{min} is the minimum number of matching bits due to chance:

$$E_{min} = m(1 - p^{ke_1} - p^{ke_2} + p^{k(e_1+e_2)}) \quad (7)$$

- $p = 1 - \frac{1}{m}$ is the probability that a specific bit is still 0 after the setting of a single bit.
- α is a calibrating parameter that is set experimentally.

The two first terms in (4) state that (1) if there is not enough features stored in the first bloom filter, the returned similarity is -1 and (2) if both bloom filters share a small number of 1 bits, any similarity is considered to be due to chance and the returned similarity is 0. The third term estimates the similarity using common 1 bits in both bloom filters while excluding some of those 1 bits due to chance. Three values are used to estimate the "by-chance" factor: t_c, E_{max}, and E_{min}. The only variable set experimentally is α which serves as a calibrating parameter. α is set to 0.3 which consistently yields a zero similarity between bloom filters of unrelated data objects.

We now have all the ingredients to compute the similarity score (SC) between two data objects. Let dob_1 and dob_2 be two data objects. Let $bf_1^1, bf_2^1, bf_3^1, \ldots, bf_s^1$ be the set of Bloom filters from dob_1 and $bf_1^2, bf_2^2, bf_3^2, \ldots, bf_t^2$ be the set of Bloom filters from dob_2. Assuming that $s \leq t$, the similarity score (SC) between the two data objects is defined as follows:

$$SC(dob_1, dob_2) = \frac{1}{s} \sum_{i=1}^{s} \max_{j \in [1 \ldots t]} HS(bf_i^1, bf_j^2) \quad (8)$$

The similarity score as defined in Equation (8) can be used to find the similarity between two data objects with comparable sizes ($s \approx t$). However, it is particularly efficient to compute the similarity between a small data object (e.g. file) and a much larger data object (e.g. hard drive), hence, $s << t$, that is, the number of Bloom filters is much larger in dob_2 than in dob_1. The similarity score computation consists in comparing each Bloom filter of dob_1 with every Bloom filter in dob_2 and keep the highest value. The kept highest values are then averaged to produce the similarity score. The maximum value of the similarity score is the maximum Hamming Similarity between two bloom filters (Equation (4)), that is, 100. That optimal value is obtained not only when all features of dob_1 happen to be in dob_2, but when they occur in the same bloom filter in dob_2. In other words, having all features of one bloom filter from dob_1 occur in dob_2 but dispersed across several bloom filters does not contribute to boost the similarity score. They should be concentrated in a single bloom filter in dob_2.

Hence, calculating the score as the average of the s maximum similarities between the bloom filters of dob_1 and all the bloom filters of dob_2 has two crucial consequences:

1. A large size of dob_2 (i.e. t) does not dilute the similarity score.
2. Having all features of dob_1 occur in dob_2 but dispersed across several bloom filter does not yield a high similarity score.

To be considered similar, the similarity score between two network traces should be larger than a certain

7 We use the same experimentally derived value as Roussev [15] which is 6 for a bloom filter of 128 features

threshold t_{sc}. Threshold selection for all AHBM use cases is discussed in Section 9.

4.4 Finding a Needle in a Haystack

The threat model we are considering is a surveillance entity through which a huge amount of traffic is flowing (Internet Service Provider (ISP), censorship entity, gateway or IDS of a large network, etc.) and which is interested to pinpoint specific sessions in the traffic (website visit, malware communication, etc.). In existing traffic analysis approaches, in particular, website fingerprinting techniques [1–4, 6, 7] learning a website "fingerprint" requires several sample visits. The set of samples have typically comparable sizes (number of packets). This allows to extract features such as the number of incoming packets, the number of outgoing packets, the number of received bytes, etc. which allow to build a model/signature for that website. Consequently, for a website model to be efficient in identifying website occurrence in unknown traffic, it should be matched, each time, with an extracted sequence of packets having a size similar to the samples used in the training phase. Captured traffic, however, comes in the form of a long sequence of packets corresponding to all sort of user activity without clear separation. This issue was not considered in most of previous work on website fingerprinting because k-fold cross-validation is typically used for the experimental evaluation. k-fold cross-validation calculates the accuracy in terms of already "split" samples.

The proposed AHBM approach is efficient to identify website occurrences in a network traffic capture of arbitrary size and hence particularly relevant for the considered threat model. As mentioned previously, this is achieved by comparing bloom filters composing the similarity digest of the website with bloom filters corresponding to different regions of the network traffic capture. By taking the average of the maximum similarities between each bloom filter of the website and all bloom filters of the unknown traffic, the final similarity score will not be diluted by the size of the unknown traffic capture. To reach a high similarity score, features of the website should not only be present in the unknown traffic, but also concentrated in few bloom filters.

Figure 5 illustrates graphically the different phases required to analyze network traffic for website and malware detection using the proposed AHBM approach.

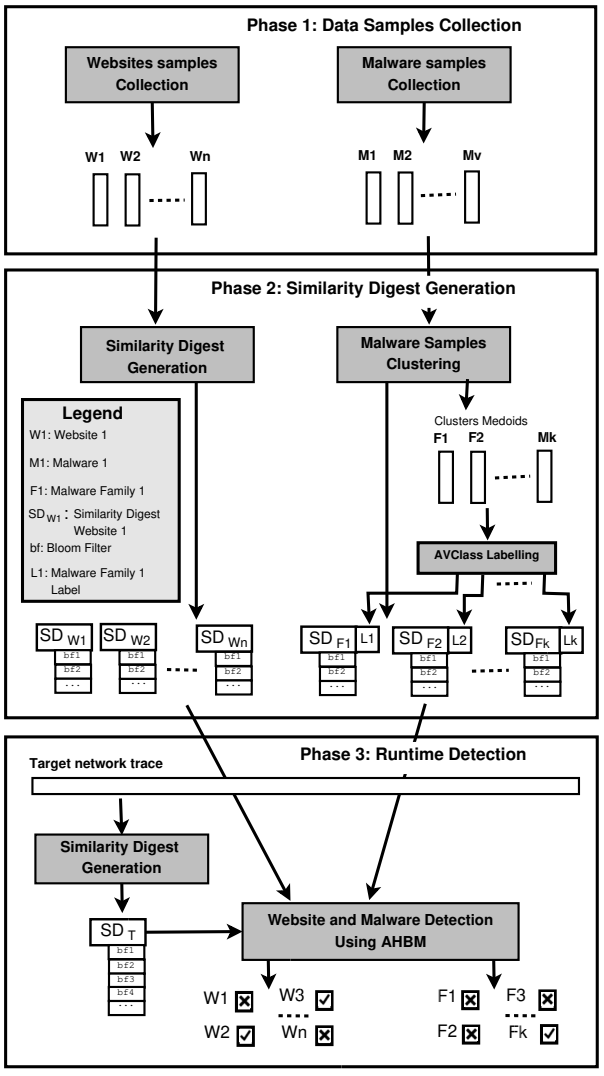

Fig. 5. The different phases of AHBM traffic analysis.

5 Evaluation Models

Most of existing traffic analysis techniques are evaluated through typical cross-validation. The basic form is k-fold cross-validation where the data is first partitioned into k equally sized segments or folds. Subsequently, k iterations (or folds) of training and validation are performed using a different segment each time. Our experiment consists in applying a 10-fold cross-validation on the collected data. That is, in each fold 90% of the samples are used for selecting the features and the remaining 10% samples are used for testing. This process is repeated 10 times (folds) choosing in each fold different samples for testing. It is important to note that class models (e.g. website models in the case of website fingerprinting) are checked with the test data samples, one by one and one

at a time. Therefore, in the remaining sections, we refer to this validation model as one-to-one cross-validation.

One-to-one cross-validation is used in two scenarios: closed-world and open-world. In closed-world scenario, the testing data is composed only of monitored instances[8]. Whereas in open-world scenario, testing data is composed of monitored and non-monitored instances[9].

In addition to one-to-one (open and closed-world) evaluation models we use a modified version of cross-validation where the data is split the same way in each fold (90% samples for training and 10% samples for testing) but matching the class (e.g. website) models with the test data samples is not performed one-by-one. Instead, all test data samples (of all classes) are concatenated to form a single long sequence of packets. Each class model is then matched with the long test data sequence two times: one time with the test data samples from all classes and a second time with the test data samples from all classes, except the class model being evaluated. This is needed to track all cases i.e. true and false positives, true and false negatives. We refer to this validation model as one-to-all cross-validation.

One-to-all cross-validation is used to assess the efficiency of our proposed approach to detect a small network object (website visit, malware communication, etc.) within large amount of traffic. One-to-all cross-validation comes also in two flavors: closed-world (test data composed only of monitored instances) and open-world (test data composed of monitored as well as non-monitored instances). The selection of training and testing samples in both evaluation models is formally defined in the Appendix.

6 Website Fingerprinting on VPN Data

Typically, Virtual Private Networks (VPNs) are used to extend a private network across public networks [18]. Packets are communicated through an encrypted tunnel (SSL, IPSec, SSH, etc.). Since an observer of a VPN traffic can only see encrypted packets to and from the VPN server, VPN is also used to hide user activity and to bypass simple proxy filtering and censorship. Therefore VPN is a typical target of website fingerprinting attacks [1, 12].

VPN data is collected using an OpenVPN server on Amazon Web Services (AWS). A client is running Windows 7 virtual machine whose traffic is tunneled through the OpenVPN server. Website visits are automated using a python script that fetches websites using chrome browser. Three datasets are collected:

1. monitored websites: The top 550 Alexa websites, 40 instances each.
2. non-monitored websites: 10000 websites (top [1000,11000] Alexa) 1 instance each.
3. overlapping websites: 1200 top alexa websites, 1 instance each, but fetched 3 websites at a time.

The first dataset (monitored websites) is used for closed-world scenario. The second dataset (non-monitored websites) is used for the open-world scenario. The third dataset (overlapping websites) is used to assess the impact of visiting websites concurrently. The latter dataset is collected as follows. Using three parallel execution threads (th_1, th_2, and th_3), th_1 starts by requesting the first page p_1, after a delay of 2 seconds, th_2 requests p_2, and then after another 2 seconds delay, th_3 requests p_3 [10]. The pages will keep loading for the next 10 seconds, after which the three pages are stopped altogether and the first round is over. The second round will proceed similarly with the next 3 pages p_4, p_5, p_6, and so on until reaching 1200 pages. For all three datasets, the script clears the browser cache, launches a tcpdump process for packet capturing, and then fetches the URL. For the first dataset, websites are fetched in a round-robin fashion (A first sample of each website is collected, then a second sample, etc.). The data has been collected during October and November 2017.

The proposed AHBM approach to traffic analysis depends on a set of parameters, in particular, the feature size B which represents the number of packets to include in a feature and the window size which specifies the number of features to consider at each step before marking the feature with the lowest S_{prec} score. Choosing a small feature size increases the granularity of the data object representation while losing specificity and yields a larger number of selected features. However, choosing a large feature size decreases the granularity of the data object and reduces the number of features. On

8 Monitored websites are websites that the attacker is interested to identify.

9 Non-monitored websites are background websites which are unknown to the adversary.

10 The delay of 2 seconds is added to simulate real scenarios [19].

	TPR	FPR
1-to-1 (CW)	0.98	0.09
1-to-all (CW)	0.81	0.21
1-to-1 (OW)	0.97	0.18
1-to-all (OW)	0.81	0.03
1-to-all (OW Overlapping)	0.56	0.36

Table 1. TPR and FPR for different scenarios using VPN datasets.

the other hand, a large window size means that a feature will be marked among a large number of considered features. This yields to fewer features marked with higher popularity scores. A small window size yields to more features marked but with lower popularity scores. The combination that yields the best results for the VPN dataset is a feature size of 2 and a window size of 4. Although a feature size of 2 packets is relatively small, but this corroborates previous results [20, 21] where the optimal traffic analysis results were obtained by considering consecutive pairs of packet size values.

Table 1 shows the results of different scenarios using the three datasets. For one-to-one cross-validation, the TPR is 97% in the open-world as well as the closed world. Previous work on VPN traffic reported lower results: Herrman et al. [1] reported a 94% accuracy[11] while Fegghi et al. [12] reported between 90 and 95% TPR. The FPR is less than 20% in both one-to-one scenarios.

For one-to-all scenarios, the TPR is smaller: 80% for closed-world and open-world, and 56% for the overlapping scenario. The FPR, however, is high: 20% for both closed and open-world. This is due to some websites whose threshold is relatively low. When such websites are not part of the testing data, due to some inherent similarity with mainstream website traffic, the similarity score may cross the threshold barrier and leads to false positives. The low TPR (56%) and high FPR (36%) for the overlapping scenario are expected since several distinguishing features will be missed because of the intermixing of packets originating from different websites.

Another way to view the results of one-to-all cross-validation is the average similarity scores for both cases (when the website is part of the test data and when it is not). Figure 6 shows the distribution of the similarity score values in both cases (red and green) using quartiles. The dashed lines represent the range of the 10 observed values in the 10-folds while the closed boxes indicate where 50% of the values are concentrated. The following interesting facts can be observed from the figure:

1. The average similarity scores, when the website is part of the test data (green) and when it is excluded (red), are clearly far apart with the former larger than the latter for all websites.
2. The standard deviation of the similarity scores when the website is not part of the test data is interestingly small. This justifies the selection of the threshold t_{sc} (Section 9) on the basis of this case (the similarity score with arbitrary traffic capture not including the website).
3. The standard deviation of the similarity scores when the website is part of the test data is relatively large. One explanation is that from one fold to another, not all distinguishing features are selected for the website model. Recall that features are selected provided they are part of at least 75% of the training instances.

7 Website Fingerprinting on Tor Traffic

Tor traffic is a common target for website fingerprinting attacks as Tor is an overlay network that aims to provide privacy and anonymity to its users. Previous work reported high accuracy website fingerprinting attacks on Tor [3, 4, 6, 22]. However, several concerns have been raised regarding the practicality of these attacks in realistic scenarios [9–11], in particular how traffic can be efficiently split to produce clearly separated chunks that can be used for classifier training.

Five datasets are used:

– Cai dataset [3]: 100 websites, 40 samples each.
– Wang SVM dataset [4]: 100 websites, 40 samples each.
– Wang kNN dataset [5]: 100 websites, 90 samples each and 9000 samples for open world.
– Panchenko CUMUL dataset [6]: 1123 websites, 40 samples each, and 111884 samples for open world.
– Our dataset: 100 websites, 40 samples each.

Our dataset has been collected by visiting the top 100 Alexa websites, 40 times each, using Tor Browser. The websites are fetched in round-robin fashion and is automated using the Tor-selenium plugin [23]. Every visit lasts for a maximum of 30 seconds and there is a 5 seconds time gap between each successive visits. Packets are stored in pcap files which are then parsed to keep

11 Accuracy is computed as: $(tp + tn)/(tp + fp + fn + tn)$.

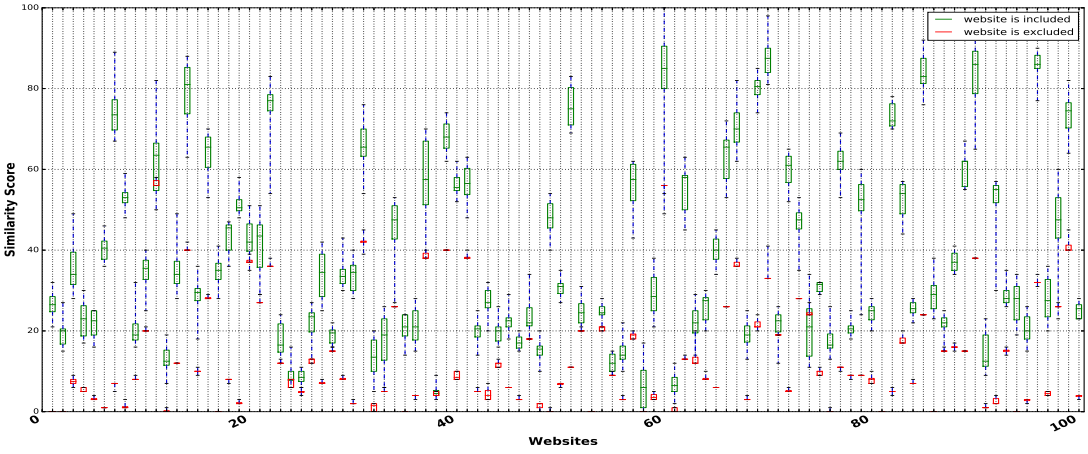

Fig. 6. Distribution of the similarity scores for each website (1) when the website is part of the testing data and (2) when the website is removed from the testing data.

only the direction, order, and size of packets (Figure 1). The data was collected during January 2017.

Tor uses fixed size cells which reduces the feature space in case the feature (block) size B is small. Recall that the block size is the number of packets in a feature. Unlike the VPN datasets where a pair of packets per feature produced the best results, in Tor the best results were obtained with a feature size of at least $B = 12$ packets with a window size $W = 4$.

Table 2 shows the result of 10-fold cross-validation on different datasets with different scenarios. The TPR ranges from 11% for CUMUL dataset to 71% obtained with Cai dataset. The FPR is very high for all datasets. The main reason is that the similarity score when the website is part of the test data is relatively close to the similarity score when the website is not part of the testing data. This is confirmed in Figure 7 which shows the quartile distribution of similarity scores on our dataset. For each website, the quartile figure shows the distribution of similarity score when the website is not part of the data (red) and when the website is part of the data (green). The dashed lines represent the range of all values observed in the 10-folds while the closed boxes indicate where 50% of the values are concentrated. For the majority of websites, the similarity score between the website similarity digest and the consolidated test data is smaller when the website is not part of the test data. However, there is a clear overlap between the green and red boxes. This explains the high FPR in Table 2.

Overall, AHBM approach produces low accuracy results when applied on Tor traffic. We studied carefully the similarity score computation for some dataset samples in order to identify the reasons behind the low accuracy. It turned out that the main reason is the fixed size cell used in Tor. Typically, such uniformity in packet sizes can be compensated by considering longer features (number of packets). However too long features makes them too specific to particular samples and hence will be filtered out by creteria 2 of AHBM feature selection (Section 4.1). Hence, choosing a small feature size increases the granularity of the data object representation while loosing specificity which yields a larger number of selected features (This leads to high FPR in Tor) while choosing a longer feature size increases the specificity of the selected features, reduces their numbers, and for Tor it leads to a low TPR. This finding suggests that AHBM approach will produce similarly low results when applied on uniform-size packets traffic (e.g. BuFLO [24]).

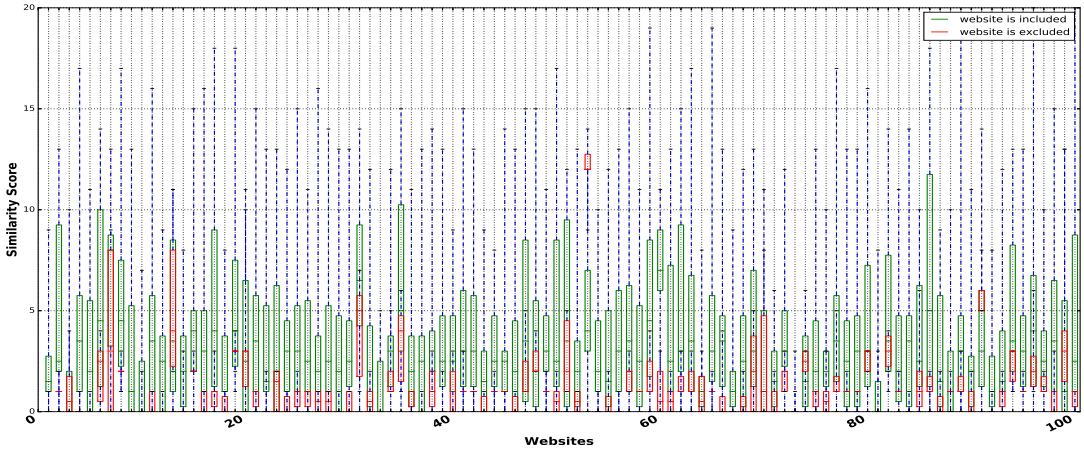

Fig. 7. Distribution of the similarity scores for each website (1) when the website is part of the testing data and (2) when the website is removed from the testing data.

	TPR	FPR	Feature size
Our dataset 1-to-1 (CW)	**0.55**	**0.19**	**10 packets**
Our dataset 1-to-all (CW)	**0.46**	**0.20**	**10 packets**
Cai 1-to-1 (CW)	**0.71**	**0.22**	**12 packets**
Cai 1-to-all (CW)	**0.54**	**0.24**	**12 packets**
Wang SVM 1-to-1 (CW)	**0.52**	**0.46**	**13 packets**
Wang SVM 1-to-all (CW)	**0.36**	**0.19**	**13 packets**
Wang KNN 1-to-1 (CW)	**0.35**	**0.20**	**13 packets**
Wang KNN 1-to-all (CW)	**0.52**	**0.20**	**13 packets**
Wang KNN 1-to-1 (OW)	**0.26**	**0.04**	**13 packets**
Panchenko CUMUL 1-to-1 (CW)	**0.41**	**0.35**	**15 packets**
Panchenko CUMUL 1-to-all (CW)	**0.37**	**0.15**	**13 packets**
Panchenko CUMUL 1-to-1 (OW)	**0.11**	**0.92**	**15 packets**

Table 2. TPR and FPR for different Tor datasets.

8 Malware Traffic Analysis

Detecting malware activity through traffic analysis is attractive since it allows to cover a large number of hosts without requiring any of these hosts to install any software. Deep packet inspection (DPI) is the major approach for network malware detection which consists in checking the packet payloads in search of specific bytes. DPI assumes that malware communicate through plaintext protocols (e.g. HTTP, IRC, etc.). As malware are increasingly using encrypted protocols as well as secure tunneling, recent network malware detection approaches resorted to traffic analysis [25–27].

The list of malware binaries used in the data collection is retrieved from virusshare[12] malware repository. The list consists of 16000 malware binaries posted on the repository on September 2016. To execute the binaries and capture their traffic, we used a sandbox-based isolation program, namely, sandboxie[13] running on a Windows XP SP3 32-bits virtual machine. As an initial filtering step, we wrote a script to automatically check the PE header of the binaries and filter out all non-windows, non-32 bit, and corrupted header binaries. Then, each valid binary is executed once through the sandbox while capturing its network communication. Each execution lasts 2 minutes. As expected, a significant portion of valid malware did not yield network activity for various reasons (malware using anti-VM, malware using anti-sandbox, Command and Control (C2) server is down, etc.).

Malware that exchanged less than 20 packets are filtered out. Among the 16000 intial malware binary retrieved from virusshare, only 1050 passed the two filtering steps. Among these 1050 binaries, we chose the first 1000 for the data collection.

Malware data collection consists in executing the filtered 1000 binaries, 10 times each, in a round-robin fashion through the sandbox on the same Windows XP SP3 virtual machine. Each execution lasts 2 minutes.

12 www.virusshare.com
13 www.sandboxie.com

From the 1000 malware, we kept only the malware that yield 10 valid samples. That is, if a malware has at least one sample (out of 10) with less than 50 packets, it is dropped. After this last filtering, we ended up with 512 malware binaries, with 10 valid samples each.

Applying cross-validation on the malware dataset the same way as VPN and Tor datasets produced very low accuracy results. The main problem is that considering each binary as a separate class is not appropriate for malware traffic analysis because several binaries correspond to the same malware due to obfuscation[14]. To overcome this issue, malware binaries are clustered into families sharing the same features. Malware clustering is a commonly used technique to deal with the redundancy of malware binaries [28, 29].

Typical malware clustering approaches rely either on malware executable bytes [30, 31] malware system call traces [32, 33], or non-encrypted network activity [28, 29, 34]. In the following, we propose a malware clustering technique that relies on the same meta-data information used for VPN and Tor website fingerprinting, namely, ordered sequence of packet sizes. This abstraction of malware binaries and packet payload makes the approach still applicable in case malware is using obfuscation or secure protocols.

Clustering algorithms that require to specify, a-priori, the number of malware clusters (e.g. K-means) are not suitable because that information is not typically available in malware family clustering problems. The most commonly used algorithm for malware clustering is agglomerative hierarchical clustering [28, 29, 33–38]. Hierarchical clustering does not scale well in presence of a large number of samples mainly because it requires to compute a distance matrix which involves computing the distance between every pair of samples [35, 36, 38]. In our case, computing the distance matrix based on the similarity score (Equation (8)) is particularly costly because the similarity score is more computationally intensive than other distance notions used in previous work (string based distance [28, 38], normalized compression distance (NCD) [34], Jaccard distance [29]). An alternative approach to address this scalability issue is to proceed in multiple stages [28, 36, 38] and hence avoid to apply hierarchical clustering on the full set of samples at once. However, this has an impact

Fig. 8. Algorithm 1. Malware clustering using AHBM

INPUT: $\mathcal{S} = \{$All malware traffic samples $mts\}$
INPUT: t_{sc}: a similarity score threshold
OUTPUT: $\mathcal{C}$: a set of malware clusters

$\mathcal{C} \leftarrow$ Initial clustering: each malware with its 10 samples as a cluster with a precomputed medoid
repeat

  //Merging
  $\mathcal{C}_{temp} = \mathcal{C}$
  for each pair of clusters C_i and C_j in $\mathcal{C}_{temp}$ **do**
    if $SC(mdoid_i, mdoid_j) \geq t_{sc}$ **then**
      Merge C_i and C_j into a new cluster and add it to $\mathcal{C}$
      Remove C_i and C_j frm $\mathcal{C}_{temp}$ and frm $\mathcal{C}$
    end if
  end for

  //Splitting
  for each cluster C_k in $\mathcal{C}$ **do**
    $Split\_recursively(C_k)$
  end for

  //Migrating outliers
  for each cluster C_k still in $\mathcal{C}_{temp}$ **do**
    for each instance mts in C_k **do**
      Find the closest cluster to mts in $\mathcal{C}$
      $closestC \leftarrow \arg\max_{i \ldots |\mathcal{C}|} SC(mts, mdoid_i)$
      Migrate mts to $closestC$
    end for
    Remove C_k from $\mathcal{C}_{temp}$
  end for

until (no more splitting and no more merging) or (maximum number of iterations reached)

function $Split\_recursively(C_k)$

  //Select a medoid ($mdoid_k$) for C_k
  $mdoid_k \leftarrow \arg\max_{i=1\ldots|C_k|} \sum_{j=1\ldots|C_k|} SC(mts_i^k, mts_j^k)$

  //Look for outliers in cluster C_k
  $outliers_K \leftarrow \{mts_i^k | SC(mts_i^k, mdoid_k) < t_{sc}\}$

  //Create a new cluster and split recursively
  if $|outliers_k| > 3$ **then**
    Create a new cluster C_n and add it to $\mathcal{C}$
    $Split\_recursively(C_n)$
  end if
end function

14 Malware obfuscation consists of a set of techniques to change the shape of binaries in order to bypass antivirus detection. This includes encryption, packing, polymorphism, metamorphism, etc.

on the effectiveness of the clustering. Perdisci et al. [36] used BIRCH clustering [39] for an initial coarse-grained stage and then used hierarchical clustering inside each coarse-grained cluster. A single hierarchical clustering stage on the full dataset, although significantly slower, produces more accurate clustering.

To cluster malware samples into families, we have chosen to propose a variant of hierarchical clustering (Algorithm 1) for two reasons. First, it does not require to compute a distance matrix on all samples. Each cluster is characterized by representative sample called a medoid. The distances (based on the similarity score) are computed only between the samples of a cluster and the corresponding medoid. Second, agglomerative hierarchical clustering starts by defining one cluster per sample and then proceeds by only merging clusters together. This produces poor clustering in our case because once a sample is merged into a cluster it cannot migrate to another one if the medoid changes. Our dataset is particularly sensitive to this issue because every malware binary is used to generate several (10) samples. The proposed algorithm combines both agglomerative (merging steps) and devisive (splitting steps) ideas[15] to address this issue [40].

Algorithm 1 takes as input a set of malware traffic samples $\mathcal{S}$ and a similarity score threshold t_{sc} and returns as output a set of malware clusters corresponding to different malware families along with their medoids. The set $\mathcal{S}$ contains several samples of each malware binary in the dataset (10 samples each in our dataset). The threshold t_{sc} is the similarity score value beyond which two instances are considered part of the same family. The clustering goes through three stages: merging (lines 5-11), splitting (lines 14-16) and migrating outliers (lines 19-26).

The set $\mathcal{S}$ is first split into an initial set of clusters $\mathcal{C}$ by putting each malware with its 10 samples into a separate cluster. The algorithm starts then looping on three types of transformations: merging, splitting, and migrating outliers. Splitting a cluster (Lines 30-43) consists in keeping in that cluster only the samples that are close to each others and the remaining samples move to a new cluster. This is achieved by choosing a representative sample from the cluster called medoid. A medoid ($mdoid_k$) for cluster C_k is a sample that has a maximum average similarity score with all remaining samples of the cluster (Line 33). Having the medoid as reference, all samples whose similarity score with the medoid is less than the threshold t_{sc} are considered outliers. If the number of the outliers is significant (more than 3), a new cluster C_n is created with the outlier samples and added to the set of clusters $\mathcal{C}$. A small number of outliers (3 or less) is tolerated to stay in the same cluster. The splitting step is recursive, that is, the newly created cluster C_n is in turn split. The merging step (Lines 5-9) consists in grouping together clusters who have similar medoids (a similarity score of more than t_{sc}). Unlike the splitting step, the merging step is not recursive: a cluster is merged a maximum of one time in an iteration. Clusters left out in the merging step (they are more than t_{sc} away from any other cluster) are considered outliers. The last step consists in migrating every sample of the outlier clusters to the closest cluster in $\mathcal{C}$. The cluster is then removed. The algorithm loops over these steps until the set of clusters is stabilized (no splitting nor merging is performed in the loop iteration).

Accuracy. To evaluate the accuracy of Algorithm 1, we use a reference clustering as ground truth. Most of reference clusterings used in the literature are built using AV (Antivirus) labels [28, 31, 34, 41]. While existing work rely on the labels of few AV engines (e.g. 5 AV engines in [34], 3 AV engines in [28]), the reference clustering used in our evaluation is based on AVClass [42], a tool for malware labeling that leverages labels from all 99 AV engines used in VirusTotal. AVClass resolves known inconsistencies resulting from the use of labels coming from different AV engines (e.g. lack of standard naming convention, the use of generic tokens, etc.).

Given a malware reference clustering, the most commonly used accuracy measures are precision and recall [31, 37, 42–44] defined as follows. Let n be the total number of malware samples ($|mts|$ in Algorithm 1). Let $C = \{C_1, \ldots, C_k\}$ be the set of k clusters returned by Algorithm 1 and $R = \{R_1, \ldots, R_t\}$ be the set of t reference clusters.

$$Prec = \frac{1}{n} \sum_{i=1}^{k} \max_{u=1,\ldots,t} (|C_i \cap R_u|)$$

$$Rec = \frac{1}{n} \sum_{i=1}^{t} \max_{u=1,\ldots,k} (|C_u \cap R_i|)$$

Precision and recall for Algorithm 1 are 82% and 55% respectively. However, while analyzing manually the results of the clustering algorithm, we noted that some malware fall entirely (all 10 samples) into a wrong

15 There are two types of hierarchical clustering: devisive which proceeds by splitting steps only and agglomerative which proceeds by merging steps only.

family (according to AVClass). By checking the AV labels of those malware, it turned out that AVClass assigned them to the wrong family. This corroborates the reported AVClass accuracy (F1 measure) of 93.9% [42]. Not considering those samples improved the precision and recall to 93% and 60% respectively. The accuracy results are summarized in Table 3.

	Precision	Recall
Using AVClass as is	**82%**	**55%**
Considering AVClass error rate	**93%**	**60%**

Table 3. Accuracy measures of Algorithm 1.

Termination. Typical agglomerative hierarchical clustering proceeds by merging clusters until one single cluster is left (containing all samples). A common approach is to terminate agglomerative hierarchical clustering when an a-priori specified number of clusters is reached. Devisive hierarchical clustering does the opposite (i.e. keeps splitting until there are as many clusters as samples) [40]. Algorithm 1, unlike agglomerative and devisive hierarchical clustering, alternates between splitting and merging steps and hence does not have a natural termination guarantee. Therefore we added a second termination condition (similar to $k-means$ implementations) which forces the algorithm to stop if a number of iterations threshold is reached. It is important to mention that in all experiments we performed using different t_{sc} values, the first termination condition (i.e. no more splitting and no more merging) has always been satisfied.

Efficiency. Executing our implementation of Algorithm 1 on the malware dataset (5120 samples) took 9.19 minutes on a laptop setup (Ubuntu 12.04, Intel Core i7-6700HQ CPU 8 cores 2.60GHz with 16GB RAM). Running the python sklearn implementation of agglomerative hierarchical clustering [45] on the same dataset, using AHBM similarity score to compute the distance matrix, and using the same setup took between 35 and 36 minutes depending on the type of linkage (complete, average, or single) and at which number of clusters threshold to terminate (we tried values between 5 and 30). As expected, most of the runtime (34.89 minutes) is spent on the distance matrix computation. The best parameters combination (complete linkage and 24 samples) for agglomerative hierarchical clustering achieved a precision of 70% and a recall of 49.3%.

Finding a needle in a haystack capability. To assess the capability of the proposed AHBM approach to identify malicious traffic in a large network traffic sequence, we carried out the following experiment. We used 5-fold cross-validation to split the collected malware samples into different training (8 samples from each binary) and testing (the 2 remaining samples) sets. For each fold, the training samples are clustered into malware families using the malware clustering algorithm. Then, we check the similarity between each malware cluster and the following three large traffic sequences: all testing samples from our dataset concatenated into a single traffic sequence, and two clean network traffic sequences (without malware infections) [46] with 4GB and 16GB data sizes respectively. This is achieved by computing the similarity score between each cluster's medoid and the three traffic sequences. Figure 9 shows the average similarity score for each fold and for each traffic sequence. The average similarity scores are very high for the first traffic sequence while relatively low for the two clean traffic sequences. This is expected since the first traffic sequence contains testing samples of each malware family while the two other traffic sequences, although very large, do not contain any malware traffic. The other interesting fact which is confirmed in the figure is that the similarity score does not get diluted by the size of the traffic sequence (4GB and 16GB) and the similarity score stays relatively low as long as the sequence does not contain malware activity traffic.

When detecting malware infections using family (cluster) medoids, there are three sources of error: 1) the AHBM clustering error, 2) the AVClass error, and 3) the malware family detection error when deployed. The first source of error means that the clustering process (Algorithm 1) either splits a malware family into several clusters or groups different families in the same cluster. The second source of error means that AVClass mislabels a malware binary. As a consequence, the label of the family (represented by its medoid) may be wrong. These two sources of errors are "labelling" errors. So a detected malware infection is wrongly labelled. Most users of malware detection systems can live with that type of error because what is important is the infection detection in the first place. Now if the correct label is needed with high accuracy, security professionals can further investigate the origin (e.g. IP address) of the infection and get more details allowing to correct the label of the infection. For the third source of error: AHBM fails to detect an infection or indicates a wrong malware family. This is the regular traffic analysis detection error.

To assess the malware infection detection error from the more specific malware labelling error, the following two experiments are carried out using the same 5-fold cross-validation as above (Figure 9). For the first experiment, for each fold, we compute the similarity score between each malware family medoid and two versions of testing data: a 5GB clean network trace and the same 5GB clean network trace with the testing samples. The results of this experiment are shown in the two first columns of Table 4. For the second experiment, for each fold, we compute the similarity score between each malware family medoid 1) with the infected trace (clean trace + malware testing samples) and if at least one of the scores is above the threshold, the trace is declared infected and 2) with the clean trace and if all the scores are below the average, the trace is declared clean. The results of this experiment are shown in the last two columns of Table 4. For malware family identification, considering the AVClass error in the clustering procedure, when the trace is infected with a specific malware, the approach can identify its presence 95% of the time. However, it mislabels other infections with this specific malware family on average 24% of the time. For malware infection detection, in all experiments carried out, the infection is always detected (at least one malware family has above-threshold similarity with the infected trace). In 18% of the time, however, a clean trace is declared as infected because one or more similarity scores between a malware family and the clean trace are above-threshold.

	Malware Family Identification		Malware Infection Detection	
	TPR	**FPR**	**TPR**	**FPR**
AVClass as is	**0.91**	**0.26**	**1.00**	**0.18**
Considering AVClass err	**0.95**	**0.24**	**1.00**	**0.18**

Table 4. Malware infection detection vs malware family identifcation accuracy.

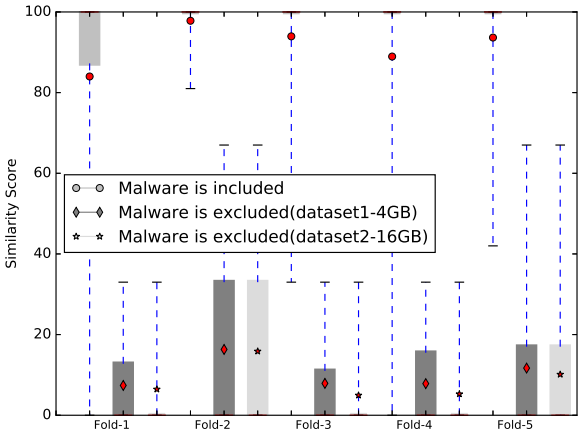

Fig. 9. Average similarity score between each malware cluster medoid and the three traffic sequences, displayed in quartiles. The segments represent the range of all similarity score values. The closed bars represent where 50% of the values are concentrated. The dots represent the average values.

9 Threshold Selection

The similarity score as defined in Equation (8) returns a value between 0 and 100 indicating the degree of similarity between two network traces. The similarity score value is not enough to declare two network traces as orginating from the same network event (same website or malware). There is a need for a reference threshold, $0 < t_{sc} < 100$ beyond which two network traces are considered similar. According to Roussev and Quates [47], a similarity score of more than 21 indicates a strong similarity between objects. However, this should not be considered as an absolute guarantee because traces of various network events (different website visits, different malware communication, etc.) may have variable inherent similarity with general mainstream network traffic. That is, the returned score might be due to similarities which are not of interest and consequently the threshold should be adapted accordingly. For example, if a network event has inherently a large (small) number of common features with mainstream traffic, the corresponding threshold should be proportionally high (low).

When generating the similarity digest (bloom filters) of a specific network event for which a set of samples is available, an appropriate threshold t_{sc} can also be selected empirically as follows. A benchmark traffic trace is collected without involving the website in hand. The benchmark is split into n (typically 10) chunks. A similarity score is calculated between the similarity digest and each of the chunks. The highest similarity score is selected as threshold. This procedure is used to select specific website thresholds for all VPN and Tor experiments (Sections 6 and 7). Figure 10 shows the ROC curve for three scenarios, namely, one-to-one closed-world, one-to-all closed-world, and one-to-all open-world on overlapping data. Each plot is obtained by computing the TPR and FPR for a different threshold value from 0 to 100. So if a higher TPR is more

important than lower FPR, the threshold should be selected towards the right of the plots (higher values). If lower FPR is preferred, the threshold should be chosen from the left side of the plots (small value).

For malware clustering (Algorithm 1), however, a fixed threshold t_{sc} is used since the clustering procedure manipulates network traces from different malware binaries/families at the same time. The threshold value is selected empirically by examining the precision and recall for decreasing values of t_{sc} and selecting the value that produces the desired trade-off between precision and recall. Table 3 results are obtained by considering a threshold value of 21 as recommended by Roussev and Quates [47]. For smaller threshold values, Algorithm 1 tends to return fewer clusters grouping together traces from different families (AVClass). For larger threshold values, Algorithm 1 tends to split traces from the same family (AVClass) into more than one cluster. This empirical selection of t_{sc} is based on a representative set of 5120 malware samples using typical malware communication protocols (telnet, ssh, IRC, P2P, etc.). In presence of newer malware families using different communication protocols and tactics, the fixed threshold value of 21 may not generalize well. For the generalizability of the proposed clustering algorithm and to maintain a good trade-off between precision and recall, it is recommended to update the fixed threshold value regularly by considering a representative set of recent and commonly seen malware binaries[16]. To further improve the detection accuracy of a specific malware family, it is possible to use the corresponding cluster output by Algorithm 1 and generate a more customized threshold value as described above for website fingerprinting.

Updating the threshold is also needed for website fingerprinting. However, it is highly likely that regular updating will be required for the malware use case since malware authors are actively trying to evade detection and there are few obstacles to changing their binaries, whereas website fingerprints are more likely to not require regular updates since websites authors are not actively trying to evade detection (indeed fingerprint evasion is not their primary goal) and there are more practical restrictions on website network traffic being morphed (through padding which adds overhead or server side mechanisms which requires cooperation from website developers).

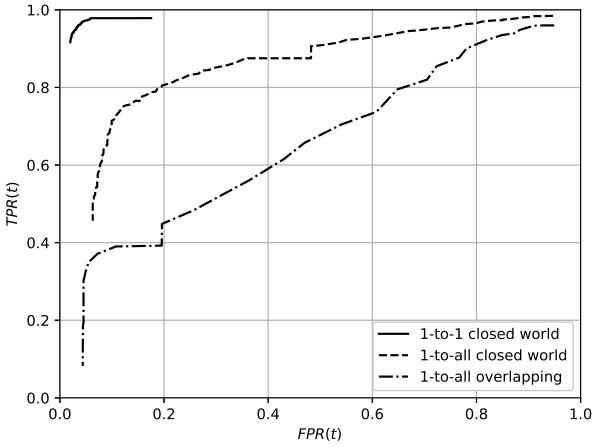

Fig. 10. ROC (Receiver Operating Characteristic) curve for the threshold t_{sc} using different VPN dataset scenarios.

10 Scalability

10.1 Complexity Analysis

The computational cost of learning a class model depends on variables $\boldsymbol{n}$, the number of packets in one instance and $\boldsymbol{m}$ the number of instances per class. Let $\boldsymbol{n_f}$ be the total number of features per instance. Since the size of a feature is $\boldsymbol{B}$ packets, $\boldsymbol{n_f = n - B + 1}$. Let $\boldsymbol{C_h}$ be the cost of a single hash computation. A hash value needs to be computed for every feature which requires a total cost of $\boldsymbol{C_h \times n_f}$. The hash computation cost can be further optimized by using a "rolling" hash function that allows the hash of the $i + 1^{th}$ feature to be computed efficiently based on the hash of the i^{th} feature [14]. Then, the precedence score $\boldsymbol{S_{prec}}$ for each feature is determined by consulting the ranktable. The ranktable access is in $O(1)$ because the normalized hash value (Equation (1)) is the actual ranktable index. The cost of finding the local minimum $\boldsymbol{S_{prec}}$ in every window is $\boldsymbol{W}$ (the window size), resulting in a total cost of $\boldsymbol{W \times n_f}$. When a feature is selected according to criteria 1 (statistical improbability), it is added in another hashtable data structure that maintains a counter of how many instances (among the $\boldsymbol{m}$ instances) the feature appears. This is useful to quickly identify the selected features (criteria 1) that are present in at least 75% of the $\boldsymbol{m}$ instances (criteria 2). The remaining $\boldsymbol{m}$ instances are processed the same way. Finally, only the features from the hashtable that have a counter larger than $\boldsymbol{m \times 0.75}$ are inserted in the class bloom filters. Let $\boldsymbol{s_f}$ be the number of selected features. Inserting each

16 This can be automated by regularly acquiring newer malware samples from online malware databases (e.g. VirusTotal [48]).

feature in the bloom filter typically requires the computation of k hash functions. However, in the current implementation, only one hash function used (SHA-1) and its digest is split into $k = 5$ sub-hashes. Inserting a feature consists in setting the k indexed bits to 1. The detailed cost of generating a similarity digest of a class is:

$$m\ n_f\ (C_h + 1 + W) + s_f\ (C_h + k) \approx O(n\ m) \tag{9}$$

Typically, the number of features selected for a website class is between 100 and 250, therefore, most of website similarity digests are composed of one or two bloom filters with a maximum capacity of 128 features.

Having a large network traffic sequence, the testing phase consists in computing the similarity score between the similarity digests of a class and the large traffic sequence. Generating a similarity digest of large traffic sequence is linear in the number of packets ($O(n)$). The similarity score computation (Equation (8) consists mainly of bloom filter comparisons (Equation (4)). Bloom filters are compared by applying a logical AND bit-wise. Let $|b_f|$ be the number of bits in bloom filter b_f. The cost of computing the Hamming similarity between two bloom filters (of the same size) is $2 \times |b_f|$. Let s be the number of bloom filters in the class similarity digest and t the number of bloom filters in the large traffic sequence similarity digest (typically $s \ll t$). Computing the similarity score requires the computation of the Hamming similarity between every bloom filter of the class similarity digest and all bloom filters of the large sequence similarity digest and taking the maximum. Hence, the computation cost is:

$$s\ t\ (2\ |b_f|) \approx O(s\ t) \tag{10}$$

10.2 Time Efficiency

Based on the complexity analysis of the previous section, the proposed AHBM approach does not involve heavy computations. In addition, it introduces the possibility to analyze large chunks of network traffic without splitting it into small sequences which improves further the time efficiency of the attack. To benchmark AHBM approach with existing traffic analysis attacks, Table 5 shows the runtime (in minutes) of AHBM, CUMUL [6], and k-fingerprinting [7] approaches on different datasets. The reported runtime corresponds to 10-fold cross-validation[17] involving both training and testing. The part of the runtime spent on training is specified between parentheses. All executions were performed on the same setup (Ubuntu 12.04, Intel Core i7-6700HQ CPU 8 cores 2.60GHz with 16GB RAM). Compared to CUMUL, AHBM is an order of magnitude faster in the closed as well as the open world scenarios. Compared to k-fingerprinting, AHBM is slighly slower in the 1-to-1 scenario because a different evaluation model (i.e. out-of-bag [7]) is used, but faster in the 1-to-all scenario. The last 4 rows of Table 5 indicate the runtime for the VPN dataset.

	websites		Runtime (min)	
Evaluation	**Mon**	**Unmon**	**AHBM(Train)**	**CUMUL**
1-to-1 (CW)	100	0	3.25 (1.58)	12.05
1-to-1 (OW)	100	5000	22.5 (1.58)	67.38
1-to-1 (CW)	300	0	10.86 (4.19)	140.73
1-to-1 (OW)	300	5000	38.14 (4.19)	184.01
1-to-1 (CW)	1125	0	85.41 (16.73)	3677.4
1-to-1 (OW)	1125	111824	1928.05(16.73)	-
1-to-all	1125	0	17.74 (16.73)	-
1-to-all	1125	111824	81.28 (16.73)	-
Evaluation	**Mon**	**Unmon**	**AHBM(Train)**	**k-fing**
1-to-1 (CW)	55	0	1.59 (0.21)	0.73
1-to-all	55	0	0.51 (0.21)	-
1-to-1 (OW)	55	100207	284.49 (0.21)	146.75
1-to-all	55	100207	3.97 (0.21)	-
VPN Dataset	**Mon**	**Unmon**	**AHBM (Train)**	
1-to-1 (CW)	548	0	15.78 (7.44)	
1-to-all (CW)	548	0	9.32 (7.44)	
1-to-1 (OW)	548	9645	73.47 (7.44)	
1-to-all (OW)	548	9645	22.87 (7.44)	

Table 5. Runtime (in minutes) of AHBM with respect to CUMUL [6] and k-fingerprinting [7] using Wang and Goldberg [5] dataset (rows 1 and 2), Panchenko et al. [6] dataset (rows 3 to 8), and Hayes et al. [7] dataset (rows 9 to 12). The last four rows report the runtime of AHBM on VPN dataset (Section 6). The values between parentheses indicate the training time. The testing time is the difference with the total runtime.

10.3 Real-time Scenario

Using the "one-to-all" mode, AHBM approach can be used in real-time scenario, that is, detecting the occurrence of certain artifacts (website or malware) in real-time network traffic. Consider an entity through which

17 The only exception is K-fingerprinting approach (rows 9 and 11) which is evaluated using out-of-bag score which does not require k-fold cross-validation [7].

Target Trace Size	Bloom Filters Generation Time	Checking Time (One by One)	Checking Time (Bulk)
5 GB	2.48s	57.57s	0.56s
11 GB	4.63s	77.83s	1.04s
16 GB	6.73s	86.50s	1.57s

Table 6. Runtime (in seconds) of all steps required to check if a target traffic trace (5, 11, and 16GB) contains each one of 550 monitored websites. The checking can be done one website at a time (Column 3) or in bulk (Column 4).

#Inst	4	8	12	16	20	24	28	32
#Feat	248	198	183	177	172	169	167	165

Table 7. Number of selected features.

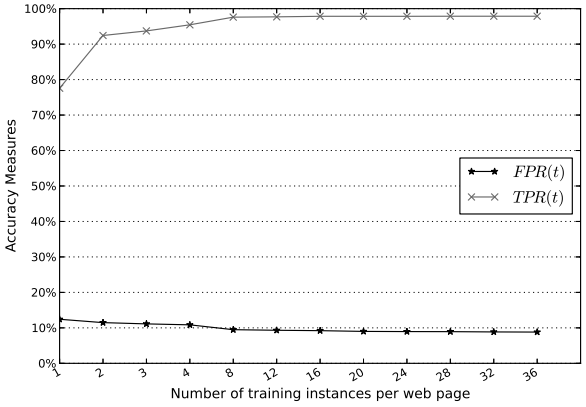

Fig. 11. TPR and FPR for one-to-one closed-world scenario on VPN dataset using different number of training instances.

a large amount of traffic is flowing (several GB/s) and trying to identify specific websites (for which the trained models (i.e. similarity digests) are available) in real-time. The entity can choose a checking frequency (e.g. every 5 seconds). Using AHBM, the following needs to be done every interval. First, a similarity digest (bloom filters) is generated for the recorded traffic since the last check (Sections 4 and 4.1). Second, similarity scores are computed between each trained website similarity digest and the target traffic. This second step can be performed in two different modes: either one website similarity digest at a time (one by one) or by concatenating all similarity digests of the trained websites together and do the checking in a single pass [49]. The bulk mode uses several threads to compute the Hamming similarities (Equation 4) between the websites and the target traffic bloom filters in parallel. Notice that both modes (one by one and bulk) can be further optimized by computing the similarity score of each website in parallel. Table 6 shows the runtime (in seconds) of each step for different size traffic traces and using the same setup (Ubuntu 12.04, Intel Core i7-6700HQ CPU 8 cores 2.60GHz with 16GB RAM). The total running time, considering the bulk mode delay (Columns 2 and 4), is within the real-time requirements. These figures can be further improved in presence of appropriate hardware and software that are typically available in surveillance entities.

10.4 Effect of the Number of Training Instances

Using fewer number of instances in the training of classifiers is a desired feature. Figure 11 shows the performance of AHBM approach when the number of instances is small. The values correspond to VPN dataset in the one-to-one closed-world scenario. Training the classifiers using less than 90% (36) of the instances involves selecting a subset of the instances (e.g. 4 out of 36, 8 out of 36, etc.) which may lead to biased results. To remove the bias, we used cross-validation in the selection of the subset of instances. Figure 11 shows the average values obtained by choosing each time a different subset of instances.

As expected, more training instances produce better TPR and FPR values. Interestingly, the TPR is still very high (94%) even when only two instances are used to generate the similarity digest. The slight change in the TPR and FPR values as the number of training instances increases can be explained by the average number of selected features (Table 7): as the number of training instances increases, fewer and better quality features are selected.

11 Conclusion

To the best of our knowledge, this is the first attack that addresses the "finding a needle in a haystack" problem in encrypted and anonymized network traffic. The attack drops several assumptions made by previous work which makes it easy to apply in realistic scenarios. In particular, it needs few samples for training, it does not require splitting the traffic, and the models (similarity digest) can be efficiently generated and updated.

Since the proposed approach is based on string similarity of packet size sequences, any manipulation of such sequences (padding, morphing, decoy, a malware that keeps changing its network communication tactics, etc.) will have an impact on the attack accuracy. Most of traffic analysis attacks suffer from this traffic shape fragility problem, but the proposed approach is more sensitive than the rest. The low accuracy results on Tor dataset confirms this finding.

As future work, the proposed AHBM approach can be used in a two stage attack where the first stage consists in identifying the region of a large traffic sequence exhibiting high similarity with a given model and the second stage consists in using existing heavy and high accuracy approaches to compute a more precise matching score.

Acknowledgement

We would like to thank the anonymous reviewers and in particular our shepherd Tariq Elahi for their very helpful comments and feedback. This work is supported by the Deanship of Scientific Research at King Fahd University of Petroleum and Minerals (KFUPM) under Research Grant FT131021.

References

[1] D. Herrmann, R. Wendolsky, and H. Federrath, "Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naive-bayes classifier," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, ser. CCSW '09. New York, NY, USA: ACM, 2009, pp. 31–42.

[2] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, ser. WPES '11. New York, NY, USA: ACM, 2011, pp. 103–114.

[3] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson, "Touching from a distance: website fingerprinting attacks and defenses," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 605–616.

[4] T. Wang and I. Goldberg, "Improved website fingerprinting on tor," in *12th ACM Workshop on Privacy in the Electronic Society*, ser. WPES'13. ACM, 2013.

[5] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting." in *USENIX Security*, 2014, pp. 143–157.

[6] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel, "Website fingerprinting at internet scale," in *Network & Distributed System Security Symposium (NDSS). IEEE Computer Society*, 2016.

[7] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique." in *USENIX Security Symposium*, 2016, pp. 1187–1203.

[8] M. Nasr, A. Houmansadr, and A. Mazumdar, "Compressive traffic analysis: A new paradigm for scalable traffic analysis," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017, pp. 2053–2069.

[9] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt, "A critical evaluation of website fingerprinting attacks," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 263–274.

[10] M. Perry, "A critique of website traffic fingerprinting attacks," "https://blog.torproject.org/blog/critique-website-traffic-fingerprinting-attacks", The Tor Blog, 2013.

[11] T. Wang and I. Goldberg, "On realistically attacking tor with website fingerprinting," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 21–36, 2016.

[12] S. Feghhi and D. J. Leith, "A web traffic analysis attack using only timing information," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 8, pp. 1747–1759, Aug 2016.

[13] M. O. Rabin *et al.*, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[14] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 76–85.

[15] V. Roussev, "Data fingerprinting with similarity digests," in *Advances in digital forensics vi*. Springer, 2010, pp. 207–226.

[16] ——, "Building a better similarity trap with statistically improbable features," in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*. IEEE, 2009, pp. 1–10.

[17] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[18] R. Stanton, "Securing vpns: Comparing ssl and ipsec," *Computer Fraud & Security*, vol. 2005, no. 9, pp. 17–19, 2005.

[19] F. D. Smith, F. H. Campos, K. Jeffay, and D. Ott, "What tcp/ip protocol headers can tell us about the web," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, no. 1. ACM, 2001, pp. 245–256.

[20] Y. Shi and S. Biswas, "Detecting tunneled video streams using traffic analysis," in *Communication Systems and Networks (COMSNETS), 2015 7th International Conference on*. IEEE, 2015, pp. 1–8.

[21] C. V. Wright, L. Ballard, F. Monrose, and G. M. Masson, "Language identification of encrypted voip traffic: Alejandra y roberto or alice and bob?" in *USENIX Security*, vol. 3, no. 3.6, 2007, p. 3.

[22] J. Hayes and G. Danezis, "Website fingerprinting at scale," *University College of London (UCL), number: Technical*

report, 2015.

[23] *Tor-Browser-Selenium*. [Online]. Available: https://github.com/webfp/tor-browser-selenium

[24] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 332–346.

[25] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer, "Using machine learning techniques to identify botnet traffic," in *In 2nd IEEE LCN Workshop on Network Security*, 2006, pp. 967–974.

[26] F. Tegeler, X. Fu, G. Vigna, and C. Kruegel, "Botfinder: Finding bots in network traffic without deep packet inspection," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 349–360.

[27] G. Gu, R. Perdisci, J. Zhang, W. Lee *et al.*, "Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection." in *USENIX Security Symposium*, vol. 5, no. 2, 2008, pp. 139–154.

[28] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of http-based malware and signature generation using malicious network traces." in *NSDI*, 2010, pp. 391–404.

[29] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 309–320.

[30] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features." in *USENIX Annual Technical Conference*, 2013, pp. 187–198.

[31] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9, 2009, pp. 8–11.

[32] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2013, pp. 144–163.

[33] I. Gurrutxaga, O. Arbelaitz, J. M. Perez, J. Muguerza, J. I. Martin, and I. Perona, "Evaluation of malware clustering based on its dynamic behaviour," in *Proceedings of the 7th Australasian Data Mining Conference-Volume 87*. Australian Computer Society, Inc., 2008, pp. 163–170.

[34] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 178–197.

[35] L. Rokach and O. Maimon, ""clustering methods"," in *"Data mining and knowledge discovery handbook"*. Springer US, 2005, pp. 321–352.

[36] R. Perdisci, D. Ariu, and G. Giacinto, "Scalable fine-grained behavioral clustering of http-based malware," *Computer Networks*, vol. 57, no. 2, pp. 487–500, 2013.

[37] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang, "Experimental study of fuzzy hashing in malware clustering analysis," in *8th workshop on cyber security experimentation and test (cset 15)*, vol. 5, no. 1. USENIX Association Washington, DC, 2015, p. 52.

[38] N. Kheir, "Behavioral classification and detection of malware through http user agent anomalies," *Journal of Information Security and Applications*, vol. 18, no. 1, pp. 2–13, 2013.

[39] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *ACM Sigmod Record*, vol. 25, no. 2. ACM, 1996, pp. 103–114.

[40] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009, vol. 344.

[41] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 108–125.

[42] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 230–253.

[43] A. Nappa, M. Z. Rafique, and J. Caballero, "The malicia dataset: identification and analysis of drive-by download operations," *International Journal of Information Security*, vol. 14, no. 1, pp. 15–33, 2015.

[44] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[45] "Agglomerative hierarchical clustering." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html

[46] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *computers & security*, vol. 31, no. 3, pp. 357–374, 2012.

[47] V. Roussev and C. Quates, "sdhash result interpretation," "http://roussev.net/sdhash/tutorial/03-quick.html#result-interpretation", 2013.

[48] "Virustotal," "http://www.virustotal.com".

[49] V. Roussev, "Managing terabyte-scale investigations with similarity digests." in *IFIP Int. Conf. Digital Forensics*. Springer, 2012, pp. 19–34.

12 Appendix

12.1 Training and Testing Samples for One-to-one Evaluation Model

Let Mon be the set of monitored websites. For each fold of the 10-fold cross validation, let $train(W)$ be the set of training samples of website W composed of 90% of all samples and let $test(W)$ be the set of testing samples of website W composed of the remaining 10% of samples. Let $Mon_{\neg W} = Mon - W$ (the set of monitored websites excluding W). Let $NonMon$ be the set of Non-monitored websites samples. Then, $\forall W \in Mon$, $\forall S_W \in test(W)$, $\forall U \in Mon_{\neg W}$, $\forall S_{\neg W} \in test(U) \cup NonMon$:

Condition	Case
$SC(SD_W, S_W) \geq t_{sc}$	True Positive
$SC(SD_W, S_W) < t_{sc}$	False Negative
$SC(SD_W, S_{\neg W} \geq t_{sc}$	False Positive
$SC(SD_W, S_{\neg W} < t_{sc}$	True Negative

Where

- SD_W is the similarity digest for website W generated using $train(W)$.
- $S_{\neg W}$ is a testing sample of a different website.
- t_{sc} is the similarity threshold as defined in Section 9.

12.2 Training and Testing Samples for One-to-all Evaluation Model

Let $Mon = \{W_1, W_2, \ldots\}$. Given a $T = \{t_1, t_2 \ldots\}$ a set of sample traces, let $seq(T) = t_1 \mid t_2 \mid \ldots$ be the sequence trace obtained by concatenating t_1, $t_2, \ldots$ together. Let $Seq(Mon) = seq(test(W_1)) \mid seq(test(W_2)) \mid \ldots$ obtained by concatenating all test samples of all websites in Mon. Let $AS = Seq(Mon) \mid seq(NonMon)$ and $AS_{\neg W} = Seq(Mon - W) \mid seq(NonMon)$. $\forall W \in Mon$:

Condition	Case
$SC(SD_W, AS) \geq t_{sc}$	True Positive
$SC(SD_W, AS) < t_{sc}$	False Negative
$SC(SD_W, AS_{\neg W)} \geq t_{sc}$	False Positive
$SC(SD_W, AS_{\neg W)} < t_{sc}$	True Negative

Where

- SD_W is the similarity digest for website W generated using $train(W)$.
- AS all testing samples of all websites (monitored and non-monitored) concatenated together in a single sequence.
- $AS_{\neg W}$ all testing samples of all websites (monitored and non-monitored) concatenated together in a single sequence except the testing samples of website W.
- t_{sc} is the similarity threshold as defined in Section 9.

	TP	FN	FP	TN
1-to-1 (CW)	2145	46	104532	1094491
1-to-all (CW)	444	104	115	433
1-to-all (OW)	4460	1020	190	5290
1-to-1 (OW)	2136	55	1735	7910
1-to-all (OW Overlap)	306	241	198	349

Table 8. Underlying TP, FN, FP, and TN values for Table 1.