UNIVERSITY OF CALIFORNIA
RIVERSIDE

Reverse Engineering User Behaviors From Network Traffic

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Guowu Xie

June 2013

Dissertation Committee:

Professor Michalis Faloutsos, Co-Chairperson
Professor Harsha V. Madhyastha, Co-Chairperson
Professor Mart Molle
Professor Iulian Neamtiu

Acknowledgments

I thank my committee, without whose help, I would not have been here.

To my parents.

ABSTRACT OF THE DISSERTATION

Reverse Engineering User Behaviors From Network Traffic

by

Guowu Xie

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2013
Professor Michalis Faloutsos, Co-Chairperson
Professor Harsha V. Madhyastha, Co-Chairperson

In today's world, more and more people are managing every aspect of their lives over the Internet. As a result, the study of Internet traffic, which is undergoing constant evolution as new technologies emerge, has attracted much attention from the research community. In this dissertation, we present a three-pronged approach to help ISPs and network administrators: a) gain insight about the applications that generate traffic in their networks, b) understand the Web browsing behaviors of their users, and c) detect in a timely fashion when external malicious entities seek to compromise their websites.

The first component of our approach is SubFlow, a Machine Learning-based tool that classifies traffic flows into classes of applications that generate them, for example P2P or Web. The key novelty of SubFlow is its ability to learn the characteristics of the traffic from each application class in isolation while traditional approaches simply try to assign flows to predefined categories. This allows SubFlow to exhibit very high classification accuracy even when new applications emerge.

The second component is ReSurf, a tool to reconstruct users' web-surfing activities from Web traffic. ReSurf enables the separation of users' intentional web-browsing (such as the click user makes) from the traffic automatically generated when the website is rendered. ReSurf, then, can be an effective method to study the browsing behaviors of users and gain insights into the evolution of modern Web traffic, which accounts for about 80% of Internet traffic.

The last component of our approach is Scanner Hunter, an algorithm to detect HTTP Scanners, external entities that selectively probe websites for vulnerabilities that may be exploited in subsequent intrusion attempts. Our algorithm is developed in response to the fact that HTTP scanners have not received much attention despite the high risk and danger they pose. Scanner Hunter utilizes a novel combination of graph-mining approaches to expose the community structure of scanners. Using Scanner Hunter, we conduct the first extensive study of scanners in the wild during a half-year period, which we also provide novel insight on this little-studied emerging phenomenon.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet has become an essential part of everyone's daily life and been widely used for online entertainment, shopping, socializing, content sharing and many other purposes. As a result, the global Internet traffic volume averaged at 32,990 Petabytes per month in 2012 and is expected to grow at a rate of 28% every year [7]. Because solving the challenge of understanding network traffic at this immense scale is important to network operators and administrators, this topic has attracted much effort from both industrial and academic communities.

1.1 Scope and Motivation of Study

Identifying the applications behind Internet traffic is one of the topics that generates much interest from network operators. For example, accurate identification of network traffic allows Internet Service Providers (ISPs) to determine separate QoS for different types of applications such as voice/video and peer-to-peer (P2P) while it is important for administrators of enterprise networks to know what applications their users are using or which types of applications are dominating the network traffic. Furthermore, understanding the origin of the traffic in a network enables its protection and constitutes the very first step in building network intelligence.

In recent years, web browsers have been used increasingly as the interface to more and more online services. As a direct result, HTTP has become the most widely used protocol and contributes up to 80% of the total traffic on some networks [70]. Given that "HTTP is the new IP in the Web 2.0 world", traffic analysis methods need to adapt to this new reality. One implication of these trends is the limited applicability of traditional traffic analysis and characterization tools [62, 111] in morden network traffic. Given the dominance of HTTP traffic, knowing that a network flow is generated by an HTTP-based application conveys little information with regards to the usage of websites/services and web users behaviors. It is therefore important for network operators to have a tool that goes beyond the HTTP class label and provides additional information about the HTTP traffic in their network.

The popularity of online web services, unfortunately, has also drawn the attention of miscreants and thus it is vital that network operators deploy tools that can detect malicious behaviors. In this particular area, we have identified through our research a new kind of threat, HTTP scanning, that has not received enough attention despite the risk it poses. HTTP scanning is a not very well-known activity that aims to discover the security weaknesses of websites and can function as a first and exploratory step that may enable subsequent website infiltrations and ultimately compromise.

1.2 Overview of our research

Motivated by the reasons disscussed in the previous section, we develop three network traffic analysis approaches, SubFlow, ReSurf and Scanner Hunter to help ISPs and network administrators to better understand Internet traffic. Each approach infers network users' activities from the network traffic they generate from different perspectives and at granularities.

SubFlow is a Transport Layer Machine Learning-based traffic classifier. It infers what application generates network flows from their Transport Layer statistics, e.g. average packet size and inter-arrival time between packets. It can be useful in answering the questions like what applications are running in the network and how much traffic is generated by a given application. As opposed to the way traditional traffic classifiers work, SubFlow learns to identify application in isolation by leveraging subspace clustering algorithms. For each application, SubFlow uses a small number of features, which are best for the application of interest, to classify flows instead of a large number of features, which may be overall good for all applications.

An HTTP category label provides very limited information about network flows because the HTTP protocol is responsible for 80% of total Internet traffic. To better understand HTTP trafic, we develop an approach called ReSurf that can reverse-engineer web users' browsing activities from HTTP traffic. Using only the information extracted from HTTP headers, ReSurf can accurately identify the head HTTP requests and associate automatically triggered HTTP requests with their head HTTP request. ReSurf is arguably the first HTTP traffic analysis tool that can answer the following important question just from observing network traffic: a) which web pages the user intentionally visited, and b) how much traffic is generated by the visits. Moreover, ReSurf can tell how web users navigate from one web page to another and how much time they stay on those pages.

In our study, we identify a new security threat, HTTP scanners, which is not well documented or studied before. In response to the fact that HTTP scanners pose high risk but have not received attiontions, we develop Scanner Hunter to uncover these malicious entities by monitoring HTTP traffic. Scanner Hunter first constructs a bipartite graph to capture the relationship between hosts and the targets they requested but failed, then exposes the community of HTTP scanners in the bipartite graph by exploiting a graph-mining algorithm. Our approach accurately detects more than 4000 HTTP scanners every week from a universiy campus network. With the help of Scanner Hunter, ISPs and network administrators can learn which hosts are HTTP scanners and what they are looking for.

1.3 Contributions

The thesis makes the following contributions:

(1) We develop three novel traffic analysis tools to better understand network traffic and network users' activities by reverse-engineering network traffic. Our approaches, which combine machine learning and graph mining algrithims, achieve very high accuracy in our evaluations.

(2) Each tool can profile network traffic from different perspectives and at various granularities. SubFlow identifies which applications are running and how much traffic they generate. ReSurf distinguishes the websites users intentionally visited from unintentional

ones and how much traffic is generated by the former. Scanner Hunter separates HTTP scanners from legitimate users and what types of resource the malicious entities are looking for.

(3) The three approaches allow us to conduct an extensive study of real-world traffic traces in various time spans, from several hours to half a year. The results we obtain can provide insights on how network traffic have been evolving in recent history.

The thesis with the three techniques that it develops make significant contributions in understanding, managing and securing Internet traffic and its users.

Chapter 2

SubFlow: Towards Practical Flow-Level Traffic Classification

2.1 Introduction

Identifying the flows generated by different applications is of major interest for network operators. For Internet Service Providers (ISPs), understanding the origins of the traffic allows them to differentiate the QoS for different types of applications such as voice and video. Moreover, the knowledge enables them to assert more control on high-bandwidth and non-interactive applications such as peer-to-peer (P2P). For enterprise networks, it is very important for administrators to know what is happening on their network, what services the users are running, or which application is dominating their traffic. Traffic classification is also important for securing the network. In fact, many traditional protocols are often used in new attacks, such as the use of IRC as Command and control (C&C) channel for botnets. Overall, traffic classification is the first step in building network intelligence.

In this paper, we define an application class (referred to as *application* throughout the paper) as an application-level protocol with a distinct documented behavior in terms of communication exchanges, control packets, etc. With this definition, our application classes include: SMTP, MSN, BitTorrent, Gnutella, POP3, etc. Interestingly, the definition of application in traffic classification research is often defined rather loosely and varies between studies. Some studies resort to higher level classes such as client-server versus peer-to-peer [120], while others attempt to detect subclasses within a class [100]. What is the appropriate granularity for defining application classes is a key question, but it does not have a single answer. The right granularity depends on the intent of the network administrator or the goal of the study. We believe that our application class granularity provides a sufficiently detailed starting point for many research questions and practical needs.

Our goal is to develop a practical traffic classification approach, which should be: (a) easy to use (ideally plug-n-play) and (b) effective in terms of accuracy. To the best of our knowledge, no such solution exists today. First, most deployed solutions rely heavily on payload-based or deep packet inspection (DPI) techniques. However, these techniques have several limitations in that they fail to classify encrypted traffic and raise privacy concerns. In addition, it is often desirable to classify traffic that is summarized in the form of flow records or packet headers. Second, port-based classification cannot function when applications randomize their ports, as they do to day, and cannot detect new applications. Third, flow-level machine learning (ML) approaches have been proposed as an alternative to the expensive and cumbersome packet-based methods. These methods use flow-level properties, such as packet sizes and timing information, to classify traffic. However, despite significant research efforts [92, 19, 89], these methods have not made the anticipated impact in the real world. This motivates us to ask two questions: **(Q1)** Why are existing flow-level methods not widely deployed? and **(Q2)** Can we design a flow-level classifier that is easier to use in practice?

The contribution of our work is twofold, as it addresses the two questions above: (a) we study and document the limitations of existing ML solutions, and (b) we propose a new approach, **SubFlow**, that operates unlike any previous ML approaches and enables profiling each application in isolation.

First, we answer the question Q1 by identifying the key limitations of ML approaches. ML methods require a set of known flows to train/bootstrap the classifier. Typically, the training data are provided by a DPI classifier. In a nutshell, supervised methods train a ML algorithm to distinguish flows among a predefined set of applications. This has two key disadvantages: (a) it is often very hard to know all applications in the network a priori and train for them; and (b) the classification performance degrades significantly when new applications emerge. Intuitively, if an algorithm learns to distinguish A from B, it is very hard to deal with a new class C. Moreover, there is no single feature set which works well for all applications. Actually, the proper features for classifying applications vary from one to another. To classify flows from all application, we'd better include all these proper features but suffers from the "curse of dimensionality". As we show in §2.2 in more detail, these challenges can significantly affect the accuracy of existing solutions, both supervised and semi-supervised.

In the second part of this chapter, we capitalize on our knowledge from answering question Q1, and we propose a conceptually different approach, SubFlow. The key novelty is that **our approach learns the intrinsic statistical fingerprint of each application in isolation**. In other words, our method learns to identify class A and then class B in isolation instead of trying to distinguish A from B. Second, in order to address the curse of dimensionality, we utilize subspace clustering, which has never been used in the context of traffic classification before. Using our technique, our classifier extracts the key features of each application and ignores the features that are not useful. This is a very attractive property given the fact that one feature can be great for identifying application A, but be useless in identifying application B. Our approach has the following key advantages, which effectively address the limitations of previous methods: (a) bootstrapping is easier and practical, as we demonstrate in the rest of this paper; (b) our approach is robust and adaptable to *change*, such as the appearance of new protocols, or the evolving behavior of existing applications.

We also show the promise of our approach using five traces from different ISPs captured between 2005 and 2011. These traces are from a geographically and functionally diverse group of Internet backbone links, which is the setting that causes most difficulties for application classification solutions [65, 54]. Our key findings and contributions can be summarized in the following points:

- We investigate, document, and explain the key limitations of previous machine learning based traffic classification approaches. For example, we show that, by intentionally excluding BitTorrent (BT) during training, state of the art classifiers tend to report

BT flows as MSN by mistake, reducing the accuracy for MSN by more than 30% (see §2.2).

- SubFlow learns to distinguish the traffic of an application with higher than 95% accuracy. Furthermore, the majority of the flows of an application that are missed in the training phase, are not misclassified to another application, instead reported to be "Unknown" (see §2.4).
- To highlight the advantages of SubFlow, we apply it in a "plug-n-play" fashion, where training data are directly derived from popular port numbers. Our classifier was able to make signatures and classify traffic successfully over a range of applications, both encrypted and un-encrypted. In fact, on a 2011 trace from a mobile provider, we identify and create signatures for the Android Marketplace protocol, without having a known payload signature for this application during training (see §2.4.5).

The rest of this chapter is organized as follows. In §2.2, we give a formal definition of our problem, highlight the limitations of previous ML methods, and identify the requirements for a practical solution. In §2.3, we provide the details of SubFlow. In §2.4, we evaluate SubFlow over different traffic traces and configurations. We review the related work in §4.5 and conclude our work in §2.6.

2.2 Definition and motivation

The task of a flow-level traffic classifier is to identify the applications (e.g., BitTorrent, RTSP, HTTP) that generate flows according to statisctical flow-level features, without relying on payload content and port number. The application name is defined as the label or class of the flow. In the paper, we consider flows are uni-directional. One TCP/UDP session has two uni-directional flows, forward and backward, repsectively. An uni-directional flow is defined as a five-tuple: {*source IP, source port, destination IP, destination port, and protocol*}. It is not rare that observing only one uni-directional flow of TCP/UDP sessions in the backbone links due to routing asymmetry. SubFlow classifies traffic only based on uni-directional flows that makes it more general than other approaches which requeire the features in both directions. Therefore, SubFlow can be deployed in both backbone and access Internet links.

Statistical flow-level features: Flow-level classifiers utilize statistical features of a flow to predict the flow's class. For this work, we have collected a number of key flow-level features that are also used by others [92, 88]. Specifically, we used the exact size and the inter-arrival time (IAT) of the first 6 packets of the flow. In addition, we use the average, max, min, and standard deviation values of packet sizes and IATs over the entire flow. Intuitively, the exact size of the first packets captures the protocol behaviors during the initiation of the protocol interaction. The statistics of IAT over entire flow are good features for classifying the traffic generated by real-time applications such as VoIP or Video.

Many ML algorithms are introduced to traffic classification area but have not been widely deployed and made the anticipated impact in practice. We attribute this to two main reasons. First, a traditional multi-class classifier trained without flows from all possible applications produces unexpected results. Second, the dilema in feature selection. We explain two reason in detail in what follows.

In a network, new applications appear from time to time. It is very hard to collect a complete and representive set of flows for training classifiers all the time. Unfortunately, **a traditional multi-class classifier trained without flows from all possible applications produces unexpected results.** We verified this using a large set of real-world data and different supervised classifiers proposed in the literature [65], such as K-nearest neighbors, logistic regression, decision trees, support vector machines (SVM), and Bayesian Networks. We used the WEKA implementation [116] of the algorithms with their default parameters as in [65]. Here, we show an example using data from Asia-3[1], where we try to classify only five protocols, namely: HTTP, SMTP, EDONKEY, BitTorrent (BT), and MSN. From the supervised algorithms we used, the Bayesian Networks gave the best results. For brevity, here we only present results from this algorithm. Our experimental methodology is the same as in all the experiments in the paper and is explained in detail in §4.2.

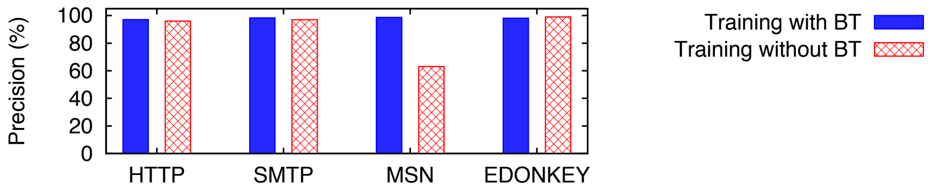


Figure 2.1: The per-protocol precision for a traditional multi-class classifier when BT flows are included or not in training

Our goal is to compare the performance of the Bayesian Network classifier when: (a) the flows from all five protocols are included during training, and (b) when the flows form one or more of the protocols are not used for training. While evaluating the classifier, we always use all the flows from all five protocols. In Figure 2.1, we compare the percentage

[1]More details on the specific trace is given in §2.4.

of correctly classified flows (a.k.a. precision) of HTTP, SMTP, eDonkey, and MSN when (a) BT is included in training and (b) when BT is not used for training. From the figure, we see that when the training set consists of all five protocols, the precision approximates 100%. Unfortunately, when the classifier does not take BT into consideration, flows from other protocols are mistakenly labeled as BT. In fact, 35% of flows labeled as MSN are now erroneous. We repeated the same experiment excluding protocols other than BT in training and achieved qualitatively similar results. Moreover, we observed that the more protocols we exclude in training, the worse the overall precision is.

A pictorial explanation for the limitation is presented in Figure 2.2. During training phase, a machine learning based classifier, such as logistic regression or support vector machine, would approach the problem of separating flows from protocol A from B. The classification decision boundaries are shown with a dotted line. It is clear that the boundaries perform well and can distinguish the two protocols from each other. But when the flows from a "new protocol" (not included in training phase) appear, the classification performance degrades because these flows are randomly classified into protocal A or B. We highlight this observation in the classfication phase.

The other reason is **the feature selection dilema in traffic classification**. The good feature set for classifying different protocols varies from one to another. A typical example is the average packet inter-arrival time (IAT), which is a great feature for identifying time sensitive applications, such as streaming, but not good in HTTP flows. In steaming, IAT and its variance will be small for the vast majority of flows, whereas for HTTP IAT varies from very small to very large, depending on the flow. On one hand, the good features for

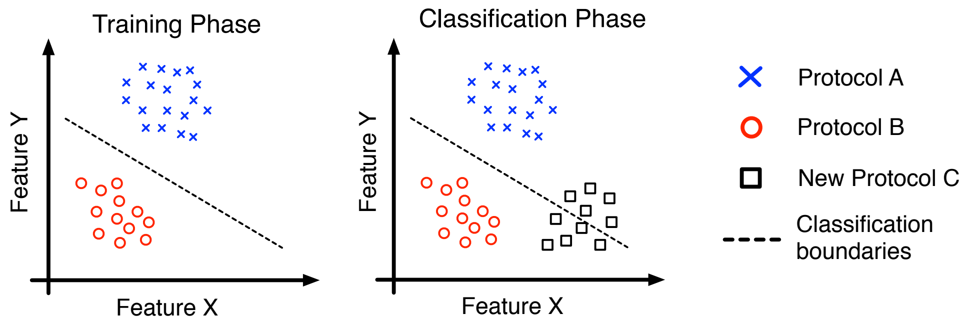


Figure 2.2: Training and testing phases for traditional classification methodologies

each application should be included to accurately classify the flows from these applications. On the other hand, the more number of features, the less effective ML algorithms will be due to the curse of dimensionality. That is, the distance functions lose their usefulness in high dimensional feature space. Simply put, the dilemma is that each application is captured best by different features, but if we simply use the union of all these features together, the classification performance degrades along with the number of applications we consider.

To overcome these limitations of traditional approaches, we propose a conceptually different solution, SubFlow. Here, we highlight the key difference of SubFlow with other methods. We consider the problem of building a classifier to identify BitTorrent flows. In Figure 2.3, we show the functional steps of (a) our SubFlow classifier, (b) a two-class (or multi-class in general) supervised classifier, and (c) a semi-supervised clustering algorithm. Most important, the input information to SubFlow is significantly different from the traditional supervised approaches. Our classifier only requires training flows form the target application in order to build a classifier. In contrast, a two-class classifier (Figure 2.3(b)) requires a significant effort to provide training data from all possible applications in the

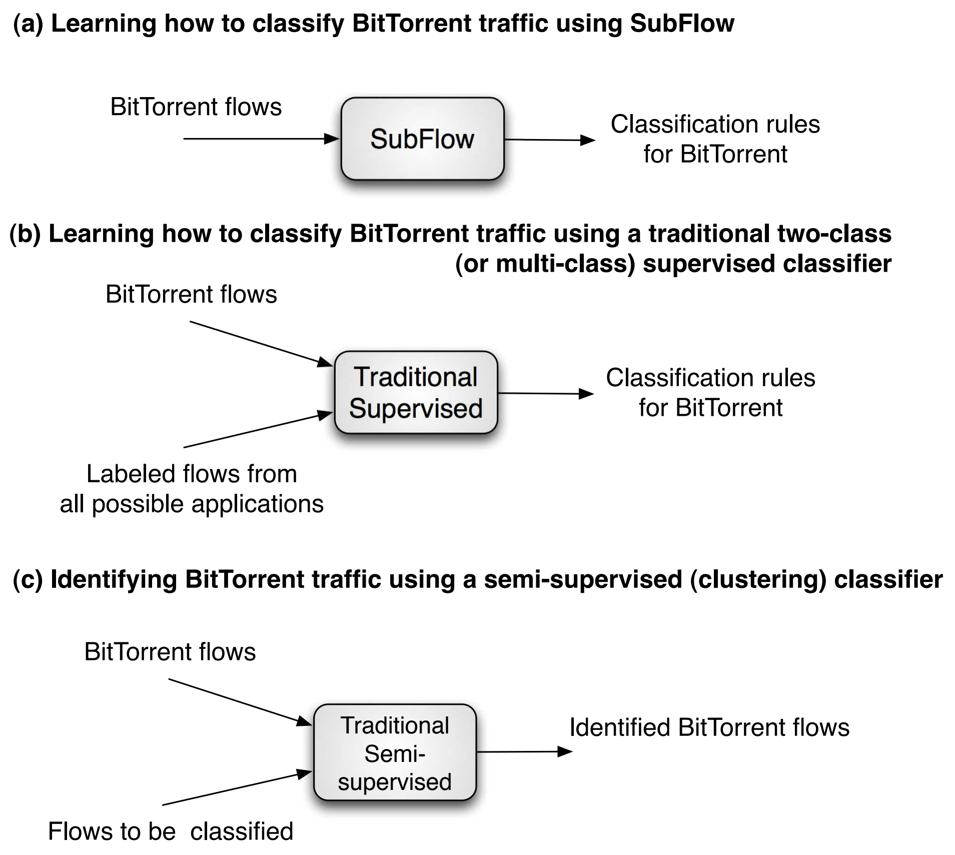

(a) Learning how to classify BitTorrent traffic using SubFlow
BitTorrent flows
SubFlow
Classification rules for BitTorrent
(b) Learning how to classify BitTorrent traffic using a traditional two-class (or multi-class) supervised classifier
BitTorrent flows
Traditional Supervised
Classification rules for BitTorrent
Labeled flows from all possible applications
(c) Identifying BitTorrent traffic using a semi-supervised (clustering) classifier
BitTorrent flows
Traditional Semi-supervised
Identified BitTorrent flows
Flows to be classified

Figure 2.3: The difference between SubFlow and traditional methods

network. As we show in Figure 2.1, if the labeled data are incomplete, due to some missing applications, the accuracy of the classifier degrades. Using the semi-supervised solution of Figure 2.3(c) avoids the tedious process of providing labeled flows from all other applications. However, as we explain next in this section, clustering algorithms suffer from the curse of dimensionality.

2.3 Traffic classification using subspace clustering algorithms

In this section, we present our SubFlow classifier. First, we give an overview of how SubFlow operates in §2.3.1. Then, we explain the details of signature generation module in §2.3.2 and flow classifier module in §2.3.3. The subspace clustering algorithm and our implementation are presented in §2.3.4.

2.3.1 Overview of SubFlow

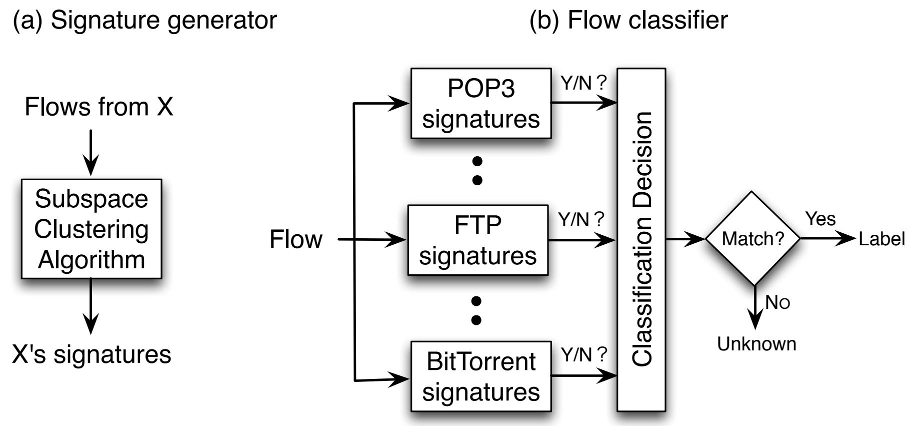


Figure 2.4: The methodology of SubFlow

Before delving into details, we first present an overview of SubFlow. SubFlow has two important modules as described in Figure 2.4: (a) signature generation, and (b) flow classifier. Signature generation module can be seen as a training phase of SubFlow. The module takes the flows from one application as input and ouput signature(s) which can characterize the flows generated by the application. To achieve this functionality, subspace clustering algorithms are employed. Subspace clustering algorithms outputs signatures in term of (feature subset, flow subset) pairs. Each application's signatures will be forwarded to flow classifier module. Based on these signatures, flow classifier module predicts the application names (labels) of incoming flows.

2.3.2 Signature generation module

Here we explain the basic operations of signature generation module using subspace clustering algorithm. The algorithmic details are described in §2.3.4.

Input: A set of flows F that belong to the same application and a full set of features S. Each flow $f \in F$ is represented as an $|S|$-dimensional vector of numerical values.

Output: One or more pairs of flows and subspaces (F_i, S_i), such that $F_i \subset F$ and $S_i \subset S$. S_i reports relevant features for the flows in F_i. This essentially projects the initial $|S|$-dimensional flows, to an $|S_i|$-dimensional subspace with $|S_i| \in (1...|S|)$. A feature subspace may contain more than one cluster comprising of different flows. That is, when $S_x = S_y$, $F_x \cap F_y = \emptyset$. Also, a flow f can belong to clusters in different subspaces. That is, $f \in F_x$, $f \in F_y$, but $S_x \neq S_y$.

A signature is in fact a set of flows F_i and a corresponding feature space S_i returned by the subspace clustering algorithm. For each signature (F_i, S_i), the flows in F_i meet a given cluster criterion (i.e., they are very close to each other) when projected into the feature subspace S_i.

2.3.3 Flow classifier module

During classification as described in Figure 2.4, an incoming flow is tested for a match over all application signatures. Essentially, each signature is a binary classifier that reports either match or not. Flow classifier module consolidates each binary classifier's result and report a final label. We next present the process in more detail.

Testing a signature for a match: When a new flow is tested over a specific signature (F_i, S_i), it is first project to S_i and then compared with the flows in F_i. The distance between flows is calculated using the standard Euclidean distance. The distance to the closest flow is set to be the distance of the test flow to signature (F_i, S_i). If the test flow is within a predefined radius (r) is reported as a match. Essentially, each signature is a fixed-radius nearest neighbor classifier with $|S_i|$ dimensions and $|F_i|$ points. The fixed radius guarantees the signatures are specific enough to match the flows of the application without matching flows of other applications. It is not fair to use the same fixed radius for all subspaces, since from the Euclidean distance formula $d = \sqrt{\sum_{k=1}^{k=|S_i|} (x_k - y_k)^2}$ we see that the larger $|S_i|$ is, the larger the distance will become, even if the distance of each individual dimension remains small. We use the basic scaling factor of $\sqrt{|S_i|}$ to remedy this. Therefore, if the one dimensional radius we use in our classifier is r it means that the value becomes $r_i = \sqrt{|S_i|} \cdot r$ for signature i. We refer to the region covered by the radius of all the points of the signature as its *region of interest.* We evaluate our algorithm over different radius (r) values in the next section.

Classifying a flow: Our SubFlow classifier contains a number of binary classifiers. Each binary classifier corresponds to one application signature (F_i, S_i). Assume that at a specific point in time we have n binary classifiers, $X = \{x_1, x_2, ..., x_n\}$. Any new flow that reaches the SubFlow classifier is processed by each of the n binary classifiers. Each binary classifier replies with a `true` or `false` and the distance (d). Therefore, the outcome is: (a) an n-dimensional boolean vector $L = \{l_1, l_2, ..., l_n\}$ where the variable l_i captures the label given by the binary classifier i, i.e., where $l_i = 1$ iff x_i labels the flow `true`, otherwise $l_i = 0$; and (b) an n-dimensional vector $D = \{d_1, d_2, ..., d_n\}$, where the variable d_i captures the distance of the test flow to signature i. Now, since an application can be associated to multiple signatures, it may be mapped to more than one binary classifier. To keep track of the mapping between binary classifiers and applications, we introduce a new vector called M, where $M(i) = App$ if the binary classifier i is from the application *App*.

The final decision on whether the flow should be labelled as *App* or *Unknown* is made running the following algorithm that use vectors L, D, and M:

1. Add all i, where $L(i) =$ `true`, the response set R.

2. If $|R| = 0$, reply as "Unknown" since no binary classifier gave a label for the flow. Else if $|R| = 1$, reply as $M(k)$, where $R = \{k\}$ since only one binary classifier labeled the flow. Else if $|R| > 1$, reply as $M(k)$, where $k \in R$ and $\forall i \in R, i \neq k, |S_j| < |S_k|$. That is, if more than one classifier labels the flows, the classifier from higher dimensional signature is chosen because it is more specific.

2.3.4 Details of subspace clustering algorithms

In this subsection, we first give an introduction of subspace clustering algorithms. Then, we explain what are the requirements for our algorithm. Finally, we go into the basic steps of the subspace clustering algorithm we use and present its subtleties.

The problem of subspace clustering is the following: given a set of data points, find a set of subspaces where data points can be grouped into clusters of high quality. A naive approach might be to search through all possible subspaces and use cluster validation techniques to determine whether there are clusters in subspaces. The exhaustive search for all subspace clusters is intractable [22, 66]. Its complexity is $O(2^d)$, where d is the data dimensionality. Most existing subspace clustering algorithms are based on different heuristics to identify subspace clusters, while keeping the computational complexity acceptable.

Requirements for the subspace clustering algorithm: Given the challenges of the subspace clustering problem, we wanted to find an algorithm that: (a) gives results that are easy to understand, (b) produces meaningful results in a reasonable time given the high dimensionality of our data. Our subspace algorithm is inspired by the bottom-up heuristic used by the FIRES algorithm [67], which is a very good match for our task at hand. Due to space limitations, we present here how our algorithm works and do not report the exact similarities and differences with FIRES. At a conceptual level, the algorithms are similar but many procedures in FIRES have been simplified in order to facilitate the interpretation of output subspace clusters. We refer the interested reader to [67] for more information about FIRES, and to [93, 91] for a survey and comparison of different subspace clustering algorithms.

As we mentioned before, the input to our algorithm is a set of flows F that belong to a single application, over a feature set S. Our subspace clustering algorithm takes the following basic steps:

1. A standard clustering algorithm is used to find the so called base clusters in each dimension $s \in S$. Essentially, we cluster all the flows in F in each 1D-space $s \in S$.

2. Base clusters from different dimensions are then merged together to form higher dimensional subspaces. In order for base clusters to be merged, we require them to have a large ratio of common flows. Merging base clusters results in subspaces of dimensionality $d \in (2, ..., |S|)$.

3. To find the clusters in each subspace, we project all the flow in F to each subspace S_i. Then, we use a standard clustering algorithm to find the clusters in each subspace.

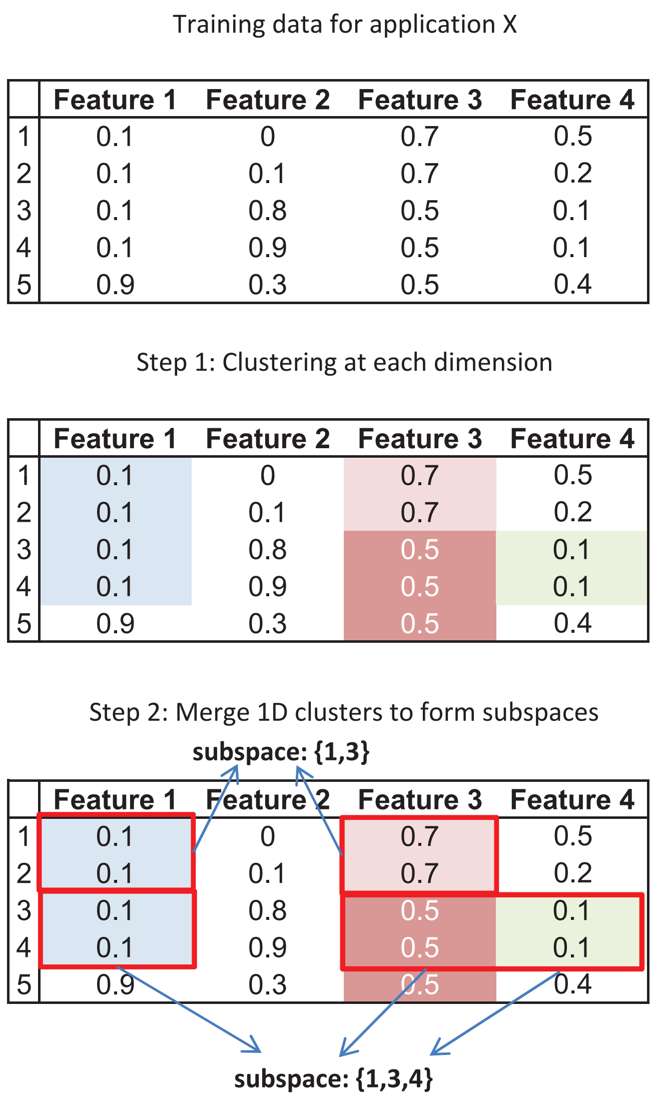
Walking through a toy example: The first two steps of our algorithm are graphically illustrated in the toy example of Figure 2.5. In the example, we have five flows and four features. At the first step, we apply clustering at each individual feature. For the standard clustering algorithm, we can use any algorithm of choice, such as DBSCAN and k-means. Here we use density based clustering algorithm DBSCAN because it fits our problem. In Step 2, we highlight the base clusters identified by using a DBSCAN. There are 2 base clusters in Features 1 and 4, two base clusters in Feature 3, and no base clusters in Feature 2. In our approach, the features that do not have any base clusters as well as that do not merge their base clusters to form higher dimensional subspaces are not used in the signatures. This is one of the great benefits of our approach, where we remove features that do not capture any dominant trend of the application at hand. More details on the example are explained next.

Additional implementation details: All the features are normalized using the standard z-score algorithm, defined as $x_{norm} = (x - \mu)/\sigma$, where μ and σ are the mean and standard deviation of the feature calculated over the training data. During the cluster merging phase, in order for two bass clusters to be merged at step 2, we required that the number of common flows between them is at least 50% of the number flows of the smaller of the two base clusters (we try different ratios and find the results are similar). Finally, it is often the case that a large cluster, say, C_1 has high overlap to two (or more) other clusters, say, C_2 and C_3, but the overlap of $C_2 \cap C_3$ is very small. The design choice of FIRES is to always split the larger base cluster into two smaller disjoint base clusters (C_{11}, C_{12}). In our implementation, we split a large cluster if and only if the new base clusters are larger than the average size of all base clusters. We see an example of splitting a base cluster in the toy example of Figure 2.5. In the example, the overlap is between the base cluster of Feature 1 and two base clusters of Feature 3. The compeering base clusters are the same feature (Feature 3), but in general, the clusters can belong in different features. As we see in Step 3 of the toy example, the base cluster of Feature 1 is split into two smaller base clusters to form the subspace of Features 1, 3 and the subspace of Features 1, 3, 4.

In our current implementation, we identify base clusters using DBSCAN [45] as FIRES. In the evaluation section, we show detailed experiments about how we choose the parameters of DBSCAN.

2.4 Evaluation

In this section, we evaluate the performance of our approach: (a) we study and finetune its parameters, (b) we test it across different network traces, and (c) we test its


Training data for application X

	Feature 1	Feature 2	Feature 3	Feature 4
1	0.1	0	0.7	0.5
2	0.1	0.1	0.7	0.2
3	0.1	0.8	0.5	0.1
4	0.1	0.9	0.5	0.1
5	0.9	0.3	0.5	0.4

Step 1: Clustering at each dimension

	Feature 1	Feature 2	Feature 3	Feature 4
1	0.1	0	0.7	0.5
2	0.1	0.1	0.7	0.2
3	0.1	0.8	0.5	0.1
4	0.1	0.9	0.5	0.1
5	0.9	0.3	0.5	0.4

Step 2: Merge 1D clusters to form subspaces

	Feature 1	Feature 2	Feature 3	Feature 4
1	0.1	0	0.7	0.5
2	0.1	0.1	0.7	0.2
3	0.1	0.8	0.5	0.1
4	0.1	0.9	0.5	0.1
5	0.9	0.3	0.5	0.4

Figure 2.5: Illustration of the extraction of subspaces using the subspace clustering algorithm

ability to detect new applications. To evaluate (c), we emulate the appearance of a new application' by intentionally excluding a known application from training (e.g., eDonkey), and then report what percentage of its flows are labelled as "Unknown", as it should be.

2.4.1 Datasets

We use five full packet traces (headers and payload) from different ISPs collected between 2005 to 2011. The ISPs are distributed across different geographic locations: three are in Asia, one in South America, and one in North America. We will refer to them them as Asia-1, Asia-2, Asia-3, SouthA, and NorthA in the paper. Some basic traffic statistics are presented in Table 4.1. These traces are collected from backbone links connecting client ISPs to their providers as well as peering links between ISPs. The traces Asia-1, Asia-2, Asia-3, and SouthA capture residential traffic from the customers of the ISPs as well as transient traffic. The trace from NorthA is from a cellular service provider and contains traffic from only mobile devices, such as laptops and smart phones with high speed data plans. Overall, our data traces are collected from a diverse set of network links, over different time periods, with different users, applications, and characteristics.

Our classifier works with both TCP and UDP flows. Because the features of the two layer-4 protocols are different (e.g., TCP flags do not have any meaning in UDP), we train TCP and UDP applications using different classifiers. In this paper, for brevity, we report the results using our TCP classifier, which covers a larger set of protocols and significantly larger portion of flows ($> 85\%$). Using UDP provides qualitatively similar results and observations. From this study we exclude flows that do not carry any payload, such as scanning traffic and failed TCP connections.

Extracting the ground truth labels of flows: The ground truth of a flow refers to its generating application. We extract the ground truth for our experiments, using a DPI classification techniques similar to those used in [63, 112]. Our traces contain traffic

Name	Collection Date	Duration	Raw Size	Total Flows
SouthA	26/08/2005	10 mins	25G	176k
Asia-1	17/08/2006	40 mins	188G	111k
Asia-2	26/07/2009	6 hours	358G	64k
Asia-3	21/09/2010	30 mins	90G	688k
NorthA	18/03/2011	3 hours	186G	130k

Table 2.1: The details of the five data traces we used

from the following applications: HTTP, SMTP, POP3, MSN, BitTorrent, eDonkey, Gnutella, Telnet, Samba (SMB), IMAP, XMPP, Yahoo IM, SSH, and FTP. The total number of unlabeled flows corresponds to 20% of the traffic. Typically, DPI labels a large number of flows as unknown because of either encrypted payload or incomplete signatures. This highlights the benefit of our algorithm, where its performance is not affected by the presence of unknown applications.

2.4.2 Classification evaluation metrics

True positives for an application i (TP_i), is the number of flows correctly classified as i. False positives for i (FP_i), is the number of flows from other applications misclassified as i. False negatives for i (FN_i), is the number of flows belonging to i but not classified to be of application i. Therefore, $Total_i = TP_i + FN_i$.

Precision for an application i, precision(i) $= \frac{TP_i}{TP_i+FP_i}$. Intuitively, this metric depicts how much confidence we have in the label i given by a traffic classifier.

Recall for an application i, recall(i) $= \frac{TP_i}{TP_i+FN_i}$. Recall is also known as detection rate. It tells the percentage of flows from application i that are detected by a traffic classifier.

New application (NewApp) detection rate. This metric measures the ability of a classifier to correctly identify new traffic and report it as being "unknown." For example, if the classifier does not have a signature for, say, BitTorrent (BT) the classifier should report all BT traffic as "unknown." Since BT was not known to the classifier before, it effectively represents a new application. Therefore, $NewApp_i = \frac{Unknown_i}{Total_i}$.

Coverage is the number of flows that were actually given a prediction (i.e., not labeled as "unknown"), divided by the total number of flows in the trace. It is also known as "completeness."

Accuracy on covered set is the number of correctly classified flows divided by the total number of predicted flows. That is, accuracy on covered = $\frac{\sum_i TP_i}{\sum_i (TP_i + FP_i)}$.

2.4.3 Experimental methodology

For evaluating our classifier, we split each trace into two parts. we use a fraction of the flows for training and the other disjoint fraction for testing. For training the classifier, we consider applications with more than 1,000 flows. We believe the number is small enough for collecting training data and large enough for extracting good signatures. The applications which have no enough flows (<1,000) are not included in training data, but included for testing the classifier. In what follows, we repeat all experiments ten times and just report the average values over all runs since the variations in different runs is very small ($< 1\%$). In all our experiments, we generate signatures for uni-directional flows. This allows us to extract different signatures from the client to server interaction, as well as from the server to the client. When we compare our classification predictions with the ground truth, we look at the labels we gave for both directions of the flow. If one direction was found to be unknown, we give it the label of its reverse direction if it exists. If the labels from two directions are different, we report the flow as unknown.

Evaluating the new application (NewApp) detection rate: With these experiments we aim to evaluate the accuracy of our classifier in detecting novel traffic. That is, when we classify the traffic of an application that we did not include in training, we want its traffic to be reported as unknown. We use the following methodology. We exclude from training one application at a time. Then, we observe how the classifier reports the flows of the "hidden" application (i.e., the $NewApp$ rate for that application). We summarize this behavior by averaging the $NewApp_i$ over all the applications i in the trace.

2.4.4 Evaluating SubFlow

We evaluate the performance of our algorithm under different configurations. This allows us to understand the trade-offs and how to select the proper configuration given the task at hand. We used one trace to explore the different configurations and then finalize these parameters for all the other traces in all our experiments. In the following experiments we show that our basic configuration gives high classification performance on all five traces.

Using different configurations

Our subspace algorithm uses DBSCAN to generate the base clusters (see §2.3). The DBSCAN algorithm has two parameters, the distance (ε) and minimum points (*minPts*) [45]. Since we already have ε to denote the maximum allowed distance between flows, we also assign the radius r of the region of interest to have the same value. Therefore, in all our experiments $r = \varepsilon$. In addition, our classifier allows us to control its precisions by excluding signature of low dimensionality. Intuitively, using only one or two features to make a signature, increasing the probability of matching flows from other applications. We refer to the minimum allowed dimension of a signature as *minDim*. In the following experiments we study the overall accuracy of our classifier by varying ε, *minPts*, and *minDim*. We use the Asia-3 trace to study these parameters and to select a good configuration for our classifier. We then apply the same configuration to the remaining four traces.

In Figure 2.6 we show the accuracy, coverage, and average NewApp detection rate when we vary the minimum dimension of used signatures (minDim) for the Asia-3 trace. For this experiment we used $minPts = 10$ and $\varepsilon = 10^{-4}$. We study the sensitivity to these two parameters next. In Figure 2.6, we see that SubFlow achieves high accuracy on the covered set, irrespective of the minDim value. As we expect, the coverage decreases the stricter we want our signatures to be, but it gives good results, with above 80% predictions, even when we use minDim of four. In this figure, we also observe a very interesting dynamic between coverage, accuracy, and NewApp rate. The more specific the signatures (i.e., the larger $minDims$ is), the higher the performance will be in classifying known applications (Accuracy) as well as identifying previously unknown traffic (NewApp). We use $minDim =$ 4 in the remaining of the paper that gives great Accuracy and NewApp rate, and good Coverage.

In Figure 2.7 we show the accuracy, coverage and NewApp rate when varying the parameter ε from $1 \cdot 10^{-6}$ to $5 \cdot 10^{-4}$. As explained in §2.3.4, all features are Z-score

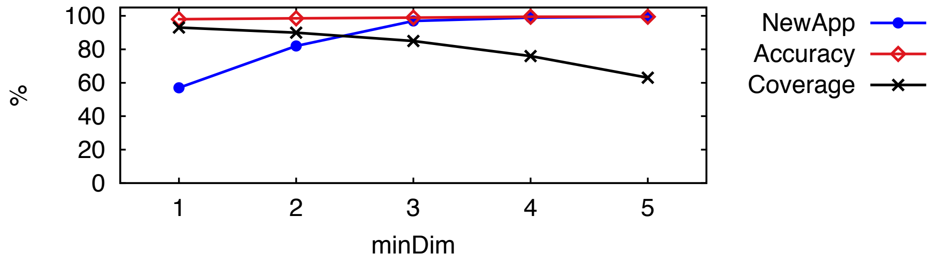


Figure 2.6: The accuracy, coverage and average NewApp rate using different *minDims* thresholds

normalized, which typically brings them in a range between -1 and 1. As we see from Figure 2.7, SubFlow achieves very high accuracy, coverage, and NewApp detection rate using very small distances between flows (ε). We attribute this to the good features being selected by the subspace clustering algorithm. When all features are relevant for an application, it allows the euclidian distance between flows to be very small. The coverage starts to decrease if we go below $5 \cdot 10^{-5}$, which is a remarkably tight bound. We chose our default value to be $\varepsilon = 1 \cdot 10^{-4}$. As we show next, this value gives good results in all our traces. Finally, as we expected, for high values of $\varepsilon > 10 \cdot 10^{-4}$, the region of interest (see r in §2.3.1) becomes very large. This means that flows that are not very close to each other will start to be mixed together, which results in less accurate signatures.

We observed qualitatively similar results and observation with varying the $minPts$ parameter as with ε so we exclude the figure for brevity. We observed values in the range of 5 to 20 $minPts$ to give high Coverage, Accuracy, and NewApp rate. For the remaining of the paper, we use the following as our **default configuration**: $minPts = 10$, $\varepsilon = 10^{-4}$, and $minDim = 4$.

We show the overall performance of SubFlow over all five traces using the default

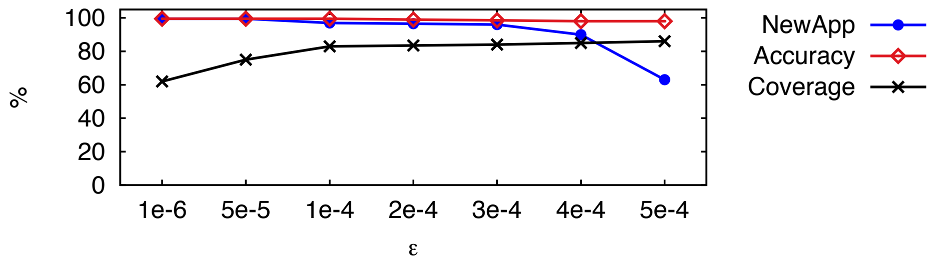


Figure 2.7: The accuracy, coverage and NewApp detection rate using different ε in Asia-3 trace

configuration in Figure 2.8. Event though we choose our parameters using the Asia-3 trace, we see from the figure that the performance is very good over all five traces. Specifically, we see that SubFlow achieves high accuracy ($> 98\%$), high NewApp rate ($> 97\%$), and good coverage ($> 75\%$) over traces with a diverse traffic composition. The high NewApp rate shows that our method can successfully report traffic from applications not included in training as unknown in all traces.

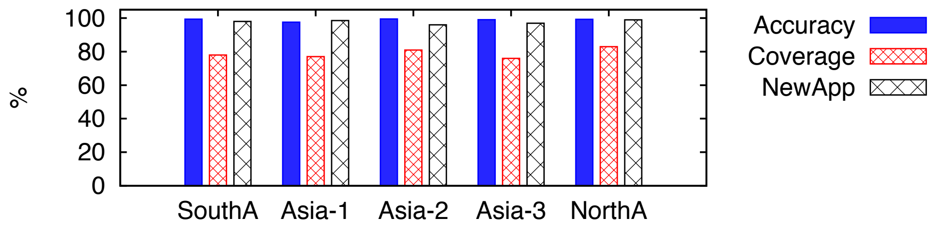


Figure 2.8: The overall accuracy and coverage and NewApp rate for all five traces using the default configuration

2.4.5 Applying SubFlow in practice

The main advantage of SubFlow. With SubFlow, classifier training becomes an easier process. Training a classifier to learn a target application, say, BitTorrent (BT), only requires flows from BT. We no longer need to have training data from the other applications in the network. This is a great advantage of SubFlow and this is what we highlight in this section.

As we show next, it is often easy to identify a dominant port of an application. We can therefore collect traffic from those ports and generate a signature for different applications. In fact, this allows us to extract flow-level signatures for applications that we did not have payload information to begin with. Such examples, include encrypted traffic on ports: 995 (POP3S), 443 (HTTPS), and 993 (IMAP4S). Here we make the basic assumption that many applications run on a set of default (well-known) ports. We observed this to be true for traditional applications, such as HTTP, SMTP, etc. We acknowledge that for applications that do not have a dominant port, we will not be able to extract a signature in this way. However, even for P2P applications, it is common for some clients to use default port numbers. In fact, using this simple approach we were able to identify a large number of traffic from EDONKEY and BT. It is important to note here that using ports to extract flow level signatures, is not the same as using port numbers to classify traffic. In fact, if we only use ports to classify BT and EDONKEY traffic in the SouthA trace, we would only get $\sim$10% recall. Using SubFlow in the SouthA trace, if we train on the dominant port of EDONKEY and apply on the traffic we can actually achieve 75% recall for EDONKEY!

Deploying SubFlow in a plug-n-play fashion. When we learn signatures for a port, we include all the traffic on that port. This includes the flows from which we have a label given by our DPI classifier, as well as those reported as unknown. This makes our evaluation similar to what the classifier will have to deal in practice. For all the flows for which we have DPI labels, we use them as ground truth to evaluate our plug-n-play approach.

For all the flows when we do not have a DPI label (i.e., encrypted traffic) we use the port numbers as ground truth (see Figure 2.10). In more detail, we use 2,000 flows for training and then apply the signature on the entire trace, which also includes the remaining flows from the dominant port. We then look closely to see if our signatures match the remaining flows on the port. In what follows, we will refer to the port numbers using the well-known application names on those ports. We believe the process of mapping well-known ports to applications is not hard. Moreover, network administrators can automate this process by using information from the Web, similar with what is described in [112].

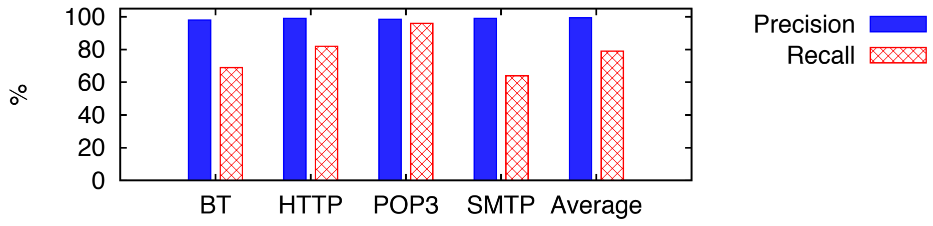


Figure 2.9: DPI-based ground truth in the NorthA trace

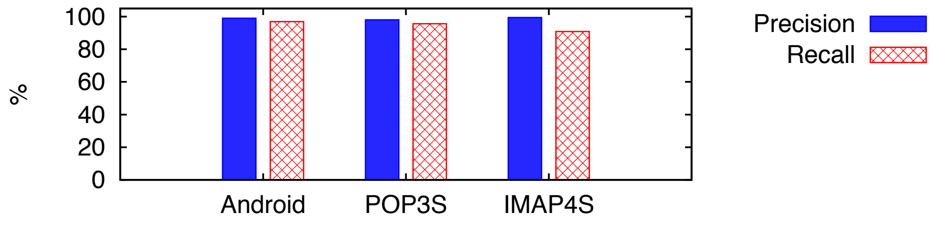


Figure 2.10: Port-based ground truth in the NorthA trace

In Figures 2.9 and 2.10, we summarize the results for the most popular ports of the NorthA trace. We observe that the precision of these signatures is very high for all applications. As we see, **SubFlow successfully classifies BitTorrent, even though**

more than 80% of its traffic uses a random port number. With the help of SubFlow by simply looking at the BT traffic on port 6881, we were able to identify 70% of its traffic, with 99% precision. The fact that some legacy applications, such SMTP, have lower recall in this trace is due to the fact that some SMTP flows use an alternative port, other than its default at number 25. Those flows appear to have some differences compared to the SMTP flows on port 25, which explains why our classifier was not able to successfully create a signature for them. More interesting is the fact that our method extracts signatures for encrypted traffic on these two popular e-mail ports for IMAPS and POP3S. As we see from Figure 2.10 the signatures extracted from these ports have high precision and recall ($> 95\%$).

With SubFlow we identify new applications and automatically extract flow-level signatures. It is worth mentioning that our method with plug-n-play operation successfully detect a new application in the NorthA trace, which our DPI classifier cannot label. After looking into these flows, we found they are generated by smart-phones accessing the Android marketplace. Our method was able to extract a signature for this applications and classify its traffic with 98.1% precision and 95.8% recall, as we see in Figure 2.10.

2.4.6 Discussion and comparison

The advantage of only needing positive samples. By design, SubFlow can profile an application relying only on its positive samples. By contrast, a two-class classifier (see Figure 2.3) will also need negative samples to extract a signature. For example, it can start observing positive samples for BitTorrent from the well know BitTorrent port (6881), as we do in SubFlow. However, unlike SubFlow , the two-class classifier will also need negative samples from *each and every* of the remaining applications. Recalling the example in Figure 2.1, even if one application is left out (not included in the negatives), it degrades the accuracy for the classes we want to detect. Moreover, we cannot use all the flows that are not on port 6881 as the negatives for two reasons: (a) most of the BitTorrent traffic will not be on port 6881, which means that the “negative" samples will erroneously include BitTorrent, which will confuse the classifier, and (b) new applications may appear. This example illustrates that using SubFlow provides significant advantages.

Comparison with other methods. In our preliminary comparisons, multi-class or two-class classifiers perform well (even better than SubFlow), when the training data for positive and negative samples are well defined and known a priory. However, as we see in Figure 2.1, these classifiers do not perform well in the presence of unknown traffic. Given that in all the traces, up to 30% of the flows are unknown, it is a challenge to precisely evaluate accuracy of these methods in a practical setting, where we cannot ignore the unknown traffic.

2.5 Related work

Over the last years, there have been many paper presenting different machine learning algorithms for solving the traffic classification problem. We refer the interested reader to a survey [92] that covers the majority of poplar techniques. In what follows, we present the most representative supervised and semi-unsupervised ML methods proposed in the literature.

Supervised ML algorithms include naive Bayes classifier [89, 23], nearest neighbours [115], decision trees [94], logistic regression [35], neural network and support vector machines [65, 75]. Features are selected manually [23] or using supervised feature reduction algorithms like Correlation-Based Filter in [89], and regression techniques [35]. Such supervised feature selection techniques inherit the same limitations as supervised learning algorithms in handling new applications. Bonfiglio et al. [23] propose the combination use of naive Bayes and Chi-square classifiers to detect encrypted and obfuscated Skype flows. The method in [23] operates at the payload and not at the flow-level.

Semi-supervised algorithms explored in traffic classification include AutoClass, Expectation Maximization (EM), K-means, DBSCAN and more [39, 122, 84, 19]. Semi-supervised algorithms group flows into different clusters according to statistical information without a priori information about traffic mix. As we explained, uncovering clusters in high dimensional data is challenging because of the curse of dimensionality.

Besides machine learning based solutions, there are some approaches based on the behavior of endhosts. BLINC [63, 120] classifies flows based on the behavior of their hosts. The label granularity from some of these approaches [120] are coarser than ours. In addition, host-based approaches, such as BLINC [63] do not perform well at the backbone [65]. All the traces we used here are from backbone networks and SubFlow performed very well. In [54], the proposed algorithm identifies traffic from known applications that try to evade detection. The algorithm uses a graph-based solutions and does not report unknown traffic. It therefore suffers from the same limitation as the other supervised approaches. Finally, the novel solution proposed in [112] utilizes information from Internet to profile IP addresses. As reported in [112], this approach gives very small recall for P2P applications. We believe that the small number or reported P2P flows from [112] can be used for training SubFlow and ultimately perform better by working together and we will pursue this in the future.

Subspace clustering and one-class classification: The FIRES [67] algorithm is one of the many existing subspace clustering techniques. such as, SUBCLU [57], FIRES [67], INSCY [13]. At the same time, the problem of building a classifier using only positive samples is not new in the data mining community. The problem is often refer to as one-class classification, or novelty detection [49, 99, 109, 74]. We believe that how we define the traffic classification problem here, will open the way for new research efforts to investigate the effectiveness of different, subspace clustering and one-class classifiers in addressing the problem.

2.6 Conclusion

The goal of this work is to develop an application classifier that would be relevant in practice. As our first contribution, we identify the factors that limit the practicality of the majority of the existing methods, especially focusing on Machine Learning techniques. For example, we see that the introduction of BitTorrent without prior training, decreases the accuracy of detecting MSN traffic by more than 30%. The second limiting factor is the curse of dimensionality. Simply put, the dilemma is that each application is captured best by different features, but if we use all these features together in a single feature space, they are too many, and the classification performance degrades.

As our second contribution, we propose SubFlow, a different approach to application classification leveraging the powerful subspace clustering technique. The key novelty is that our approach learns the intrinsic statistical fingerprint of each application in isolation, which deviates from typical classification approaches. We show that SubFlow has a lot of promise. We stress-test the capabilities of our approach in a series of experiments with five different backbone traces. SubFlow performs very well with minimal manual intervention: it identifies traffic of an application with very high accuracy, on average higher than 95%, and can detect new applications successfully.

Chapter 3

ReSurf: Reconstructing Web-Surfing Activity From Network Traffic

3.1 Introduction

HTTP is the new IP in the Web 2.0 world, and traffic analysis methods need to adapt to this new reality. First, web browsers are being widely used as the interface to a large number of services and applications, such as Email, gaming, file sharing, and video streaming. Second, today HTTP is the most widely used protocol, contributing up to 80% of the traffic on some networks [70]. One implication of these trends is the limited relevance and applicability of traditional traffic analysis and characterization tools [62, 111]. Assigning flows to an HTTP category today conveys very limited information with regard to the usage of websites/services and web users behaviors.

Given the above trends, it is increasingly important for network administrators to monitor and characterize web traffic for operational and security purposes. First, understanding traffic is important for managing and provisioning one's network. Second, such capabilities are important for security, since more and more modern malware spreads via websites and botnet command & control channels utilize HTTP. Overall, the more information administrators have about the traffic, the more effectively they can manage the network, identify anomalies and prevent attacks. At the same time, extracting information from network traffic is needed by regulators who aim to protect the rights of the consumers and allow a healthy competition between content providers and between ISPs. In addition, analyzing web traffic is important for researchers who want to study modern websites and their evolution [53, 25].

The overarching problem we address in this paper is the following. Given web traffic collected at a network link, we want to be able to look "under the hood" and reconstruct the user behaviors. Here is a list of motivating questions: (a) What websites (e.g., `google.com`, `cnn.com`) are *explicitly* requested by a user as opposed to being accessed automatically by his browser in the background? (b) How much traffic is generated by each request? and (c) What are the typical web surfing user patterns and the typical referral relationships across websites? We want to answer these questions starting from raw network traffic, such as a *tcpdump* trace, or web-proxy records.

Making the problem more specific, we can identify two sub-tasks: (a) group together HTTP requests generated by a single **user request**, such as a click, and associate them with the primary website requested by the user; and (b) reconstruct the **click-through stream**, i.e., the referral relationship between user requests, to identify whether a user's re-

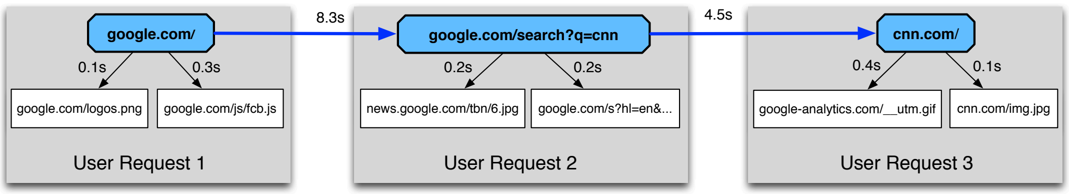


Figure 3.1: An example of a click-through stream with three user requests

quest to a website is from a hyperlink clicked on an earlier website or from within the same website. Figure 3.1 illustrates the above concepts. Understanding web traffic at both the user request and click-through levels provides insights into the user's web-surfing activity.

The problem of understanding web-surfing activity from network traffic has been studied before in [102, 53, 18], but not to the extent we do here. Schneider et al. [102] focus on reconstructing the browsing activity in social networking websites by using specialized features. The primary goal of those studies is to understand user behaviors in social networking websites and not to provide a generic methodology for reconstructing web-surfing activity from different sites. Closest to our work is the StreamStructure methodology proposed in [53], which utilizes the web analytics beacons generated by tracking services. By relying on tracking services, their approach can identify websites that do send beacons, which on average decreases the coverage by 40% (see Figure 3.5). We extensively compare ReSurf with StreamStructure in Section 3.3 and discuss related work in more detail in Section 3.5.

In this paper, we make two main contributions:

(A) ReSurf: reconstructing web-surfing activity. We develop a systematic method to reconstruct user requests and their relationship in click-through streams. First, we create a graph that represents the referral relationships between HTTP requests. Second, we identify the HTTP requests that are generated explicitly by the user, such as clicking on a link or typing a URL in a browser's address bar. Finally, we reconstruct the user request by grouping “subordinate" HTTP requests that are generated due to other requests, such as the acquisition of a video, an image or a web advertisement.

(B) Extensive measurements and validation. Our experiments with real data traces and our validation with both real and synthesized data provide the following highlights: (i) *ReSurf can reconstruct web-surfing accurately.* We show that our approach can identify and reconstruct user requests with more than 95% precision and 91% recall on all our traces, while we highlights the limitations of the state-of-the-art methods that rely on web analytics beacons [53]; (ii) *Web users are continuously being exposed to advertising and tracking services.* We observed in our traces that 50-60% of user requests (i.e., clicks) trigger an interaction to a tracking or advertising service. In fact, we observed that the chance of a user triggering such as service after just three user requests is close to 90%; (iii) *Click-through streams are surprisingly “shallow"*, as the median number of websites in a click-through stream is one or two, and only 5% of the click-through streams have more than three websites.

This chapter is structured as follows: In Section 3.2, we present the problem, provide the necessary background, and describe our data sets. In Section 3.3, we explain ReSurf, and evaluate its classification performance. The observations extracted from data traces are presented in Section 3.4. Finally, we discuss related work in Section 3.5.

3.2 Terminology, problem definition, and datasets

We present the necessary background, previous work, and the web traffic traces used in our study.

Terminology: We use the term **user request (UR)** to describe a single user action, such as clicking on a hyperlink, visiting a page from one's bookmarks, or submitting a web-form, which includes a search engine query. Figure 3.1 depicts three user requests represented by big gray boxes: one to Google homepage, a Google query, and one to CNN, via clicking a hyperlink returned by the query. A user request generates one main HTTP request, which we refer to as the **head HTTP request** indicated by the colored octagons in the figure, and it usually accesses an HTML or XML file. In practice, the head HTTP request will often generate more HTTP requests that acquire different **web-objects**, such as images, videos, or javascripts. We call these subsequent requests **embedded HTTP requests**, which happen without the user explicitly requesting them. In the example, we indicate these requests as white boxes (i.e. `google.com/logos.png`). Note that embedded HTTP requests can obtain objects from different websites, such as ad servers and content distribution networks (CDNs). We use the term **primary website** to indicate the website requested by the head HTTP request, which also represents the website that the user intends to visit.

A **click-through stream** (CTS) is a series of consecutive user requests, where the later user requests in the stream are follow-ups of the earlier user requests ones. Figure 3.1 shows an example of a click-through stream with three user requests, as we discussed above. The Google query, which is the second user request, would not have been possible without the user having visited Google first, in the first user request. Subsequently, the third user request is generated by the user's clicking on the returned results of the Google query.

Problem definition: Given web traffic collected at a network link (think HTTP packet headers), we want to reconstruct the web-surfing behavior. We want to identify head HTTP requests, and correctly group their associated embedded requests. Simply put, in our example in Figure 3.1, we want to identify the “grey boxes". Once we do this correctly, we can identify which websites users visit explicitly and how much traffic is generated by each visit.

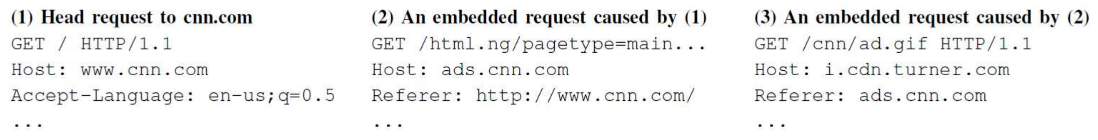
When analyzing web traffic, we have to work with information available in the HTTP requests and responses. In Figure 3.2 we give an example of only three HTTP requests generated by a visit to `cnn.com`. Following the above terminology, the first HTTP request is the *head request* to the *primary webpage* (i.e., `cnn.com`) and the other two are *embedded HTTP requests.* For each HTTP request, the domain name of the web server is located in the `Host` field of the HTTP header. Even though all three requests are triggered by the visit to `cnn.com`, in this example, only one has `www.cnn.com` as the host name. From this example, we see that by looking at HTTP headers in isolation, it is hard to track which visits they originated from.

An important piece of information is the **referrer** field in an HTTP header (visible in Figure 3.2). This field shows which previous HTTP request triggered the current HTTP request. Moving back to the graph of Figure 3.1, in “user request 1” the request for `google.com/logo.png` has `google.com` in its referrer field. In the same example, when a user visits `cnn.com` by clicking a link in the Google search results, the referrer field of the resulting

HTTP request indicates the Google search result page as the origin. In Section 3.3, we show how we can carefully combine the HTTP requests as a graph and use it to identify head HTTP requests.

Challenges: Correctly attributing individual HTTP requests to a user request and to the primary website is quite complex. First, when looking at HTTP transactions in isolation, it is hard to know which website the user intentionally visits. In our traces, if we use the `host` field in HTTP headers to identify the primary website, it results in only 20-40% accuracy. We can see this from Figure 3.2 where only one of the three requests has the intended website (`www.cnn.com`) as the host name. Second, users often browse multiple websites at the same time, which causes flows and HTTP requests to intermingle. Modern web pages are fairly complex [25]; rendering a single page may generate tens of HTTP requests towards different web servers. In Figure 3.1, for clarity, we keep only a subset of web objects. In reality, even within few minutes the size of the referrer graph reaches several hundreds of nodes. In our traces, the median size of the referrer graph over a ten minute interval is 200 nodes for an IP address. Third, many websites, such as CDNs, web-ad servers, and web analytics services are used by many websites and shared across several services.

Data sets: The web traffic traces used in our study are summarized in Table 4.1. Our traces cover several thousands of millions HTTP requests over long periods of time. They include an ISP link trace, two university traces of different sizes, a mobile traffic trace and a synthesized trace in a control enviroment. We collected traces in both controlled and uncontrolled environments, which allows us to both examine user browsing activities in the wild as well as verify the correctness of our methodology. The users in our traces are also



(1) Head request to cnn.com

```
GET / HTTP/1.1
Host: www.cnn.com
Accept-Language: en-us;q=0.5
...
```

(2) An embedded request caused by (1)

```
GET /html.ng/pagetype=main...
Host: ads.cnn.com
Referer: http://www.cnn.com/
...
```

(3) An embedded request caused by (2)

```
GET /cnn/ad.gif HTTP/1.1
Host: i.cdn.turner.com
Referer: ads.cnn.com
...
```

Figure 3.2: A simplified example of HTTP request headers issued by a browser during a visit to `cnn.com`.

diverse: researchers in a university lab, residential ADSL users, students and academic staff from a large university campus, as well as mobile device (smartphone and tablet) users. This allows us to compare the browsing patterns between different users. Details regarding the exact locations and the names of the providers for all our traces are intentionally kept anonymized due to privacy concerns.

We use the same traffic collection methodology for all the traces and capture all the IP packets on TCP ports 80, 8000 and 8080 in both directions. More details can be found below.

LAB: We collected this traffic trace from a research lab in a university in the US. In the lab, there are about 15 graduate students and 20 laptops/desktops. The collection duration spanned six non-consecutive months over the period of December 2010 until September 2011.

ISP-1: The trace was collected from an edge link of a European residential ISP. We were given access to only the first five packets of each unidirectional TCP flow.

MOB: We collected this trace from from a 3G/4G mobile service provider in the US. The vast majority of the traffic is generated by the mobile devices, such as smart-phones and tablets.

CAM: The CAM trace was collected from a university campus in China, containing the browsing activities of about 28.2K users. Our monitor device sits on the edge gateway connecting the campus to the public Internet. All download and upload traffic from the whole campus goes through the monitor point. Due to the amount of traffic, we did not store raw IP packets, instead logged all important HTTP header fields for all HTTP transactions. Specifically, the fields includes: timestamp of each request, client/server IPs, URL, referrer, content-type, content-length, HTTP response code and user-agents. To preserve privacy, client IPs are anonymized. We applied our method to several different days of traffic. The trends extracted from different days of traffic are very similar. So we only show the results for one weekday in the paper.

SYN: The trace was generated in a controlled environment for the purpose of evaluating ReSurf. We generated the traffic by replaying nine volunteers' Google Chrome browsing history. We first extracted the timestamp, referrer and URL field of each visit from their browsing history. Then, to establish the ground truth, we replayed each visit using the following procedure. We instructed Google Chrome to open each URL in isolation. At the same time, we collected all the traffic on TCP port 80, 443, 8000 and 8080 using a packet capturing software (tcpdump). After 60 seconds, we closed the browser and saved the captured HTTP traffic to an individual file. To emulate how the traffic would be if it came directly from the user's surfing activity, we carefully adjusted the time stamps and referrer fields of HTTP traffic according to Chrome's browsing history. After replaying all visits, we merge all these individual files to form a complete traffic trace. Since each visit was collected and stored separately, we effectivity have the ground truth for each HTTP request in the trace.

Name	CAM	LAB	ISP-1	MOB	SYN
Starting date	Mar 9 2012	Oct 3 2010	Aug 25 2011	Jan 7 2011	Aug 11 2011
Duration	2 mon	6 mon	24 h	3 h	1 mon
# of HTTP transactions	19B	1.2 M	1.7 M	22.9 M	186K
# of Clients (IPs)	28.2K	21	359	3,521	9
Ground truth available	No	No	No	No	Yes
Payload	HTTP header	Full	Full	Full	Full
Users	Students & staff in an university	Graduate students in a CS lab	Residential users	Smartphone/tablet users in 3G networks	-

Table 3.1: An overview of the web traffic traces used in our study.

3.3 The ReSurf approach

Here, we present the ReSurf methodology, evaluate and compare it with existing solutions. Finally, we discuss the practical issues and limitations of our method.

A. The ReSurf Methodology. The goal of ReSurf is to group HTTP requests into user requests (see definition in Section 3.2). Our approach works in two steps. First, we identify the *head HTTP requests* by using different features from each HTTP request. These features include: the size of the web-object, the type of the object, the timing between successive requests, and others. Second, we use the referral relationships (see definition in Section 3.2) to assign all the embedded HTTP requests to their corresponding head request. We explain the methodology step by step below.

Step 1. We form the HTTP referrer graph. We represent the HTTP requests from the same client IP address as a `referrer graph`. ReSurf builds such a graph for each IP address over a period of time (e.g., every ten minutes). An example of such a graph is shown in Figure 3.1, with the exception that we don't know which HTTP requests are in which user requests (i.e. shown as grey boxes), until we use our method. We provide a very high level description of this graph creation. For generating the graph, we use the `referer` field from each HTTP request to identify the previous request that triggered the request to the current website (as indicated by the `host` field). We generate a directed graph that captures these referral relationships and we enrich it with edge weights that represent the time difference between the two requests. Figure 3.1 shows an example of the HTTP referrer graph of a user accessing `google.com` and `cnn.com`. The nodes in the HTTP referrer graph are web-objects annotated with their complete URL. The directed edges capture the referral relationship between nodes where a directed edge from A to B means A is B's referrer.

In practice, the construction of the referrer graph hides many subtleties. First, character case, URL encoding and the presence or absence of trailing slashes are very common in HTTP headers. For the ease of string matching between referrer and URL, we unify character case, unquote URL encoding and strip all trailing slashes. Second, HTTP redirection (HTTP status code 302) complicates the construction of the referrer graph. We remove 302 HTTP redirections by combining the endpoints of the HTTP redirection in referrer graph into a new super node. Third, some web objects may have empty referrers. From our experiments, we learn there are two major reasons for such cases. (a) A Flash player plugin in Firefox has a well-known bug that does not append the referrer while requesting flash objects. (b) For some objects whose URL are dynamically generated by javascripts, the referrer fields in their HTTP requests are empty. Parsing the javascript file can prove useful for identifying the referrer, but only if payload is available. Without using payload, there are two possible solutions for handling these HTTP requests with empty referrer. The conservative one is to simply label them as unknown. The aggressive one is to attach these to their closest HTTP requests. In this paper, we use the conservative strategy and opt for precision.

Step 2. We identify all the head HTTP request candidates. ReSurf selects head request candidates according to the following rules:

(a) The candidate should be an HTML/XML object.

(b) Since most modern web-pages are fairly complex, the size of candidates should be larger than V bytes and candidates also should have at least K embedded objects.

(c) The time gap between candidates and their referrers should be larger than a threshold T. The reason is that head requests are usually further away from their referrers in time since they are initiated by users. By contrast, embedded requests are very close to their referrers because they are automatically initiated by browsers.

Step 3. We finalize the identification of the head requests. We utilize the referral relationship between the head request candidates. Specifically, a candidate is classified as a head request if its referrer is also a head request or if it has no referrer. In the referrer graph, nodes with no referrers have no incoming edges. In Figure 3.1, the `google.com/` in "User Request 1" is an example of such a node with no referrer. Such nodes are formed when a user, say, opens a web pages from a browser bookmark or by directly typing the URL in the browser. If the referrer is not empty, it means the user navigated to a web page by following the links from a previous web page. This implies that the referrer of a head request should also be a head request.

Step 4. We assign embedded HTTP requests to head requests. ReSurf associates embedded HTTP requests to head requests by utilizing the timing information and referral relationship in the referrer graph. In fact, once we know the head request of a user requests, it is easy to attribute the rest of HTTP requests to user requests. For each HTTP transaction (node), we traverse the referrer graph backwards until we reach a head request. If an HTTP transaction (node) has more than one incoming edges, we follow the edge with the smallest time difference (i.e., smaller weight on the edge). In this way, the path will eventually lead back to the head request that was triggered by the user request. If a node has no referrer and is not a head request, it is labeled "unknown."

Below we will show that ReSurf outperforms the current state-of-the-art [53], and provides high classification precision and recall.

B. Evaluation. We use the standard classification metrics of precision and recall. Precision is the number of true positives (TP) divided by number of TP and false positives (FP), $P{=}TP/(TP{+}FP)$. Recall is the number of TP divided by the number of TP and false negatives, $R{=}TP/(TP{+}FN)$. We also use the F1 score which is the harmonic mean of P and R, specifically, $F1 = 2\times\frac{P\times R}{P+R}$.

To evaluate the performance of ReSurf, we ask the following complementary but slightly different questions.

Q.1: How accurately can ReSurf identify head HTTP requests? We want to quantify how effectively ReSurf identifies the head requests from a large set of requests. Given that the number of head requests is usually much less than the total number of requests, this question allows us to focus only on head requests. For example, if out of 100 requests one is a head and the others are embedded, if a classifiers reports all the requests as embedded its precision is 99%, but would offer limited utility in solving our problem. For this reason, we report the P and R on head requests separately.

Q.2: How accurately can ReSurf classify head and embedded requests? We want to quantify how effectively our approach classifies each HTTP requests as a head or an embedded HTTP request. Unlike Q.1, we report results over all HTTP requests and not only over the head requests. That is, precision represents the number of correctly classified HTTP requests compared to the total number of HTTP requests classified by our algorithm. Note that ReSurf may leave some requests unlabeled (a.k.a unknown). Recall expresses the total number of classified HTTP requests compared to the total number of existing HTTP requests in the trace.

Q.3: How accurately can ReSurf associate HTTP requests to their corresponding user request? This is a more demanding question than the classification for Q.1 and Q.2: we want to associate each HTTP request with the generating user request. This is a **multi-class classification** problem, where each user request is a separate class. For example, if an embedded HTTP request R is correctly identified as embedded, but it is associated with the wrong user request, we will consider it a misclassification. The precision captures the number of correctly classified HTTP requests compared to the total number of HTTP requests classified. The recall reports the correctly classified HTTP requests compared by the total number of HTTP requests in the trace.

We use the following values for the parameters in ReSurf: T=0.5 seconds, V=3000 bytes and K=2 embedded objects. We justify this selection later in this section.

A key issue in evaluating any classifier is how to determine the ground truth in the datasets. To address this challenge, we use two different approaches: (a) using a synthesized trace SYN, and (b) using the labels from a classifier that is based on web analytics beacons.

(a) Validation using ground truth from the SYN trace. In SYN trace, at each point in time we knew exactly which website was being visited, and what requests were generated by the visits to those websites. Details regarding the generation of the SYN trace are given in Section 3.2. Figure 3.3 shows the precision, recall and F1 score when we apply ReSurf on the SYN trace, for all three questions, Q1-Q3. As we see, all metrics are above 90%, showing that ReSurf can successfully identify the originating website for the vast majority of HTTP requests. Moreover, we see that the precision of ReSurf is very high, 96% and above, implying high confidence in our classification of requests.

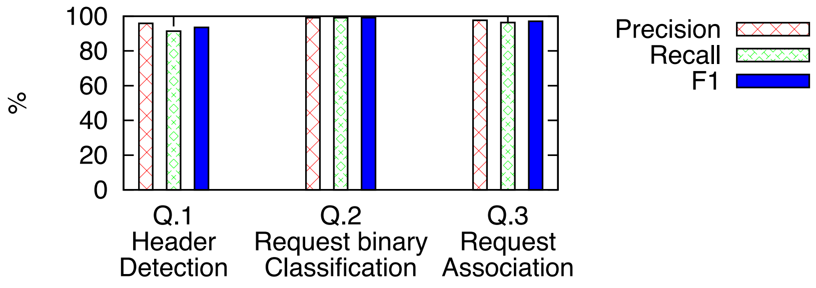


Figure 3.3: The precision, recall and F1 score in the SYN trace.

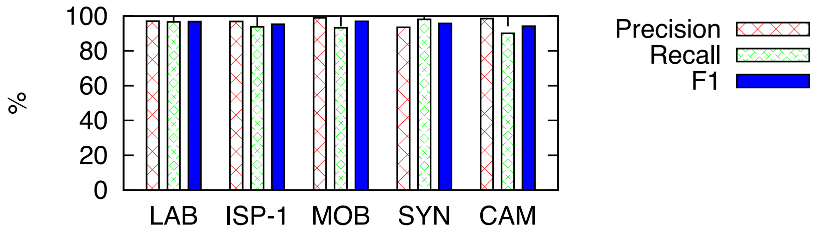


Figure 3.4: The precision and recall for detecting head requests (Q.1)

(b) Validation using web analytics beacons as ground truth. For the CAM, LAB, ISP-1 and MOB traces, we do not have the ground truth. Therefore, we evaluate ReSurf based on the predictions given by the *StreamStructure* [53] method. This method is based on the observation that many websites use web analytics beacons to track their web pages and objects. Intuitively, the web analytics beacons report to the analytics server which page is visited. And this helps us detect which is the head request and towards which primary website. We consider web analytics beacons from three major services: `google-analytics.com`, `pixel.quantserve.com` and `yieldmanager.com`.

Here, we give a more detailed explanation of the beacon method (i.e., StreamStructure), using the google-analytics beacon as an example. Once the object or web page tracked by google-analytics is requested, a beacon is generated based on the requested object's URI and sent to a google-analytics server in the form of "special" HTTP GET request. Unlike regular HTTP GET requests, beacons' URIs encode the URI of the tracked object/page. Therefore, after some careful parsing of beacon's URI, we can identify the primary website of the user request. We refer the reader to [53] for more details about *StreamStructure*.

As we will discuss later in this section, *StreamStructure* can be used for only a fraction of the requests, since only a small percentage of requests use beacons. However, this set of requests can help us determine the effectiveness of ReSurf providing an additional ground truth set. To achieve this, we first use beacons to identify as many head requests as possible. We refer to this set of identified head requests as S. Then, we compare how well ReSurf performs over the known set S. Figure 3.4 shows the precision, recall and F1 for head detection (Q.1) using beacons as ground truth. We observe that ReSurf achieves above 96% precision in all traces and 91-98% recall. The results show that our approach performs consistently well across all the datasets, which are collected in different continents and during different time periods. Note that we only use web analytics beacons here to establish the ground truth, but **ReSurf does not use beacon information** during its classification process.

Using web-analytic beacons is not enough. A natural question is why we don't just use web analytics beacons exclusively for user request reconstruction. Even though the use of beacons gives good results for those websites that use them, we discuss one of identified limitations here. **The majority of user requests (~80%) do not have a beacon in our data traces.** In all our traces, we find that less than 21% of the user requests that were found by ReSurf have beacons. Given the precision and recall of ReSurf in the controlled dataset SYN, we are confident that this percentage is reasonably accurate estimate of requests in the other traces. To further verify this, we used the SYN trace, for which we have the ground truth, and observe that only 23.9% of them carry a beacon. In the ALE trace, the ratio of user requests with beacons is roughly 60%. However, note that ALE does not represent surfing patterns from real-users, and only covers the popular website homepages as reported by Alexa. To summarize, we observed that beacons can only successfully identify approximately 21% of the user requests, compared to above 91% we achieve with ReSurf.

Figure 3.5 shows the recall for detecting head requests in the SYN and ALE traces using *StreamStructure* and ReSurf. As we see, with *StreamStructure* the recall is 22% and 60% for the SYN and ALE traces, respectively. The higher recall in the ALE trace is due to the higher popularity of web analytics by very popular websites. By contrast, ReSurf works consistently well in both traces with recall above 92%. Unfortunately, for the CAM, LAB, ISP-1 and MOB traces, we cannot repeat the same experiment since we do not have ground truth. Overall, we observed that ReSurf identifies double the number of head requests in these traces compared to *StreamStructure*.

Using *StreamStructure* and ReSurf on the same trace results in different result. In Figure 3.6 we plot the distribution of user requests to the top websites for the CAM trace. We see that the reduced number of identified requests by *StreamStructure* leads to different results. For instance, the top website with ReSurf corresponds to 18% of all user requests, whereas the same value for *StreamStructure* is 8%.

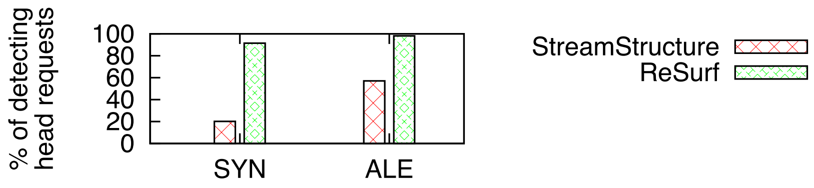


Figure 3.5: The recall for detecting head HTTP requests (Q.1) using StreamStructure and ReSurf.

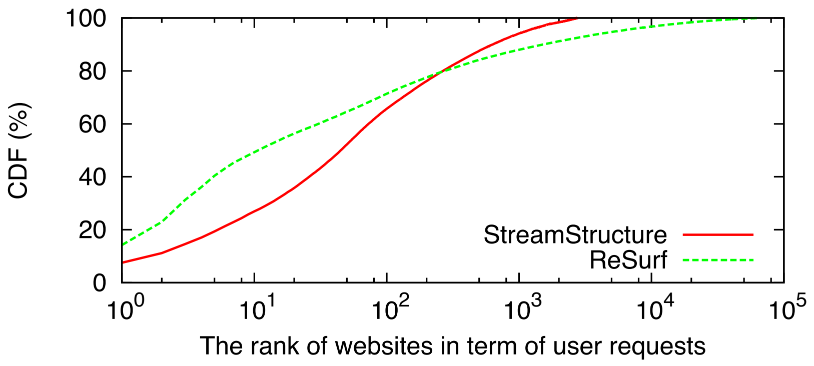


Figure 3.6: The user requests to top websites using StreamStructure and ReSurf in the CAM trace.

Evaluating ReSurf over a different range of parameters. We examine the effect of different parameters on the performance of ReSurf. We only show the plots for Q.1

for brevity; the performance for all questions is qualitatively the same. We use the SYN trace to set our parameters and then apply them to the rest of the traces.

Figure 3.7 shows the F1 metric for detecting head requests (Q.1) using different values for the volume V and the out-degree K over the SYN trace. We observe that the precision increases and the recall decreases as we increase the value of V. Intuitively, large `html/xml` files are more likely to be the primary web-site of an actual user request compared to shorter ones. Short `html/xml` files typically carry advertising related content and are triggered by embedded requests. At the same time, by further increasing V, we start considering only very large `html/xml` files as head requests, which results to lower recall. As we see from Figure 3.7, the combined behavior or P and R captured by the F1 score, exhibits good performance for V in the range of 3000 to 5000 bytes. To achieve both good precision and recall, we choose V=3000. In the same figure, different lines show how the F1 score changes when the out-degree K varies from 1 to 5. We find that the values 2 and 3 gave the best results, with K=2 performing slightly better in the range of parameter V.

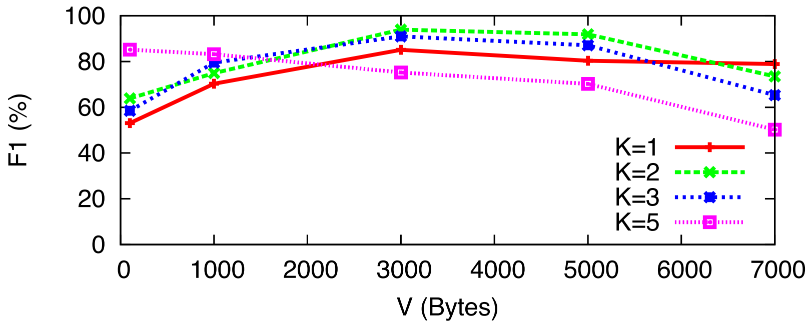


Figure 3.7: The F1 score of detecting head requests (Q.1) with different parameters.

Regarding T, we found that our approach exhibits good performance as long as T is less than 1 second and more than 0.1 second. The results are not shown due to space limitations. In the rest of paper, we use these setting: T=0.5 seconds, V=3000 bytes and K=2 embedded objects. Finally, our observation interval for generating the referrer graphs and performing our classification is 10 minutes. We experimented with different intervals in the range of few seconds up to 30 minutes and we observe better results and faster computations with windows in the range of 5 to 15 minutes.

C. Discussion. *What about encrypted web traffic?* ReSurf uses information from the HTTP header, therefore, if the web traffic is encrypted (e.g., using HTTPS) our approach cannot classify those flows. However, by analyzing our real-word traces (see Table 4.1), we observed that the encrypted traffic only amounts for 2% to 8% of the total web traffic. Unencrypted web traffic is the norm today and we believe it will continue to amount for a significant portion of the traffic in the future. The analysis of encrypted web traffic remains an interesting, open problem.

How is ReSurf affected by users behind network address translation (NAT)? Having users behind NATs is very similar to having users with very high activity. Since referrer graphs are built per IP, NAT users will appear as one "heavy user" with a complex referrer graph, for user accesses within the same time windows. There are two cases here: If different NAT users browse completely different websites, their referrer graphs will not be connected and ReSurf will distinguish different requests. On the contrary, in the worst case where two users request the same web page at the same time, ReSurf will combine them as one large request. However, it will still be able to attribute their traffic to the originating website. Finally, there may be cases where some embedded requests are "multiplexed" between more than one user request and disambiguating is hard; however, we have not observed that to be a problem in our study. Note that our goal is twofold: i) Group HTTP requests to identify the initial user requested page, and ii) identify the user click-through stream. Hence, having users behind NATs does not affect the first goal, while the second is impacted if users follow the same stream of pages at the same time.

Can ReSurf classify traffic in real-time? We have not applied and tested ReSurf in real-time classification. From the offline experiments with our current implementation, ReSurf can classify ten minutes of one thousand users' HTTP traffic in 5-8 minutes. However, Step 1 requires the collection of traffic for several minutes before ReSurf can analyze the referrer graph and classify the requests. Therefore, ReSurf can only classify requests a few minutes after their creation. As mentioned earlier, off-line analysis of web traffic is useful to operators that want to understand how their network is being used, as well as for researches that want to study modern trends and changes in web activity. Real-time classification can be important to network operators that want to enforce different policies and achieving this requirement is left as future work.

3.4 Using ReSurf on web traffic

We now use ReSurf to group HTTP requests into user requests and analyze how users behave in our four web traffic traces. We focus on three main directions:

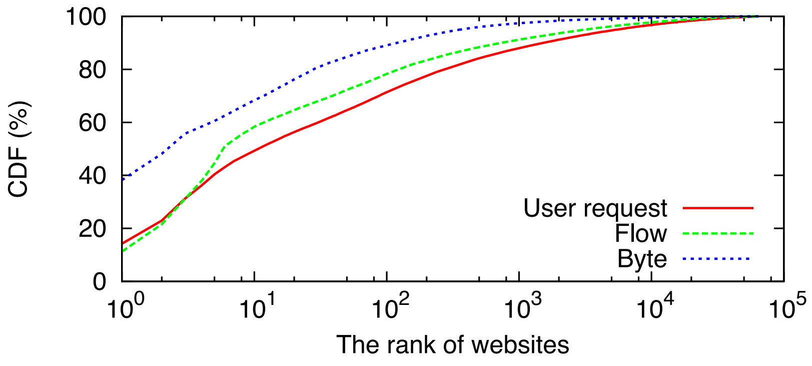


Figure 3.8: The top websites in the CAM trace in terms of user requests, flows, and bytes. The x-axis (rank) is in logarithmic scale.

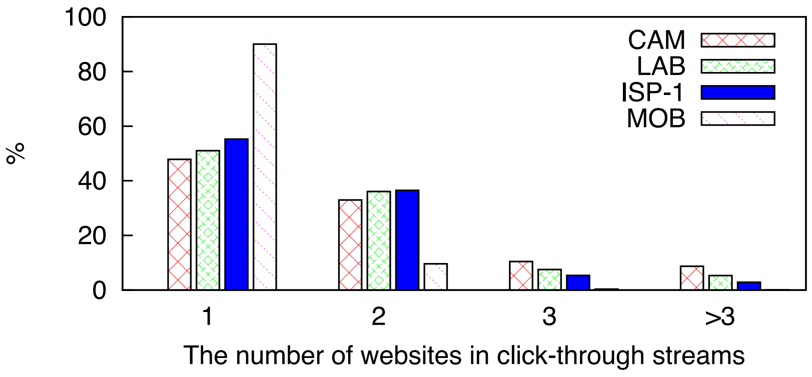


Figure 3.9: The number of user requests in click-through streams (CTSs).

3.4.1 The popularity of websites in terms of user requests

As expected, in all our traffic traces we observed that some websites are more popular than others. Figure 3.8 shows the cumulative share of user requests, bytes, and flows for the top websites in the CAM trace. We observe that the byte-curve is more skewed compared to the others, with the traffic to the top website (`115.com, an online file sharing website`) accounting for 40% of the total traffic in bytes. The corresponding cumulative share of flows and user requests for this top site is lower at 16% and 17% respec-

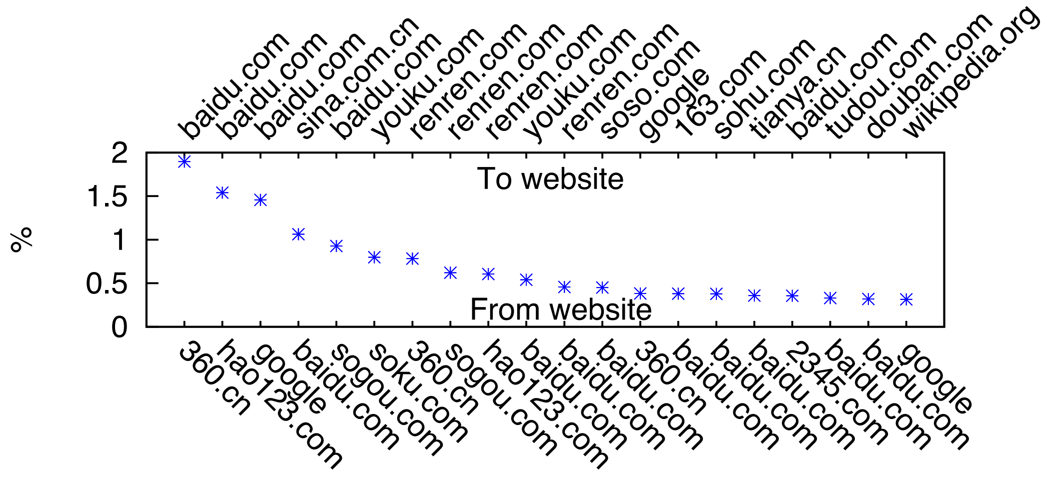


Figure 3.10: The top website transitions for the CAM trace.

tively. In our traces, we found the flow and user request rankings to be similar; which is also supported by how close the two curves are together in Figure 3.8. That is, the top websites in terms of flows are usually the websites that have the most user requests and vice versa. Intuitively, the more user requests a website receives, the more flows will be generated.

We also observed the traffic volume of a website depends on the content it distributes. That is, the top websites in terms of bytes usually are multimedia streaming websites, such as `youtube.com`, and sites featuring adult-content video. On the other hand, the top websites in terms of user requests and flows correspond to social networking and search sites, such as `facebook.com`, `baidu.com`, and `google.com`. To give an example of how different the byte ranking and user request ranking for different website are, in the CAM trace `115.com` ranks the first in terms of bytes and 16th in terms of user requests. It covers a remarkable 40% of the byte volume and only 2% of the number of user requests. Our observations suggest the existence of two categories of websites in terms of the traffic they generate: (a) high traffic volume sites, and (b) high flow/user request sites.

3.4.2 The click-through streams and web browsing patterns

In Figure 3.9 we show the distribution of the number of websites traversed in a single click-through stream (CTS). The definition of CTS is given in Section 3.2. Here we assume that a CTS ends after an inactivity period of 30 minutes [1]. Surprisingly, the vast majority of CTSs span only up to two different websites. In fact, the percentage of CTSs with more than three websites is less than 5% in all our traces. These observations suggest a browsing behavior that is "focused" on a particular task. For example, the user starts with a particular goal in mind, searches for something on a popular search engine, and stops when he clicks on the correct link that takes them to that website they are looking for. This behavior is further supported by the most popular external referrals shown in Figure 3.10. Our measurements show that "referrer" websites are usually web portals, search engines, and social networking, while "referred-to" websites are content providers, like news sites, online video, and information sites (e.g., wikipedia).

Mobile traffic: Finally, we want to highlight that the "focused" browsing is more prevalent in mobile traffic where we see in Figure 3.9 that 98% of CTSs have a maximum length of two. We believe this is due the fact that the lower bandwidth available to these users deters them from initiating long browsing sessions that span over multiple sites.

[1]We experimented with inactivity periods of 5-60 minutes and we observe similar results.

3.4.3 The exposure of users to advertising and tracking services

Our goal is to determine the percentage of user requests in our traces that involve: (i) advertising (ad) services, (ii) tracking services (i.e., web analytics beacons), or (iii) either an ad or a beacon. We summarize the results of this study in Figure 3.11, where we show the percentage of user requests to web pages that have at least one ad or trigger at least one analytic beacon to a tracking site. In order to identify popular ad services we use the popular keywords and patterns compiled by open source Ad-blocking software [3]. To identify beacons, we use the approach described earlier in Section 4.3. From Figure 3.11, we see that 18-36% of all user requests, depending on the trace, involve tracking services, 40-50% of them directly access at least one ad, and 50-60% access either a beacon or an ad. This suggests that more than half of the user requests (e.g., clicks) in the web traffic involve some form of advertisement or tracking!

We take this study one step further and try to understand the probability of a user encountering ads, beacons and either as a function of the number of user requests (e.g., clicks) that he makes. For this experiment, we treat each distinct IP address as a single user and use ReSurf to create a sequence of consecutive user requests for each IP. In the CAM trace, IT regulations enforce that no NAT is being used, which increases our confidence about IPs used by a single user at any time. Then, we randomly select X consecutive user requests from all IPs and caculate the probability of them containing at least one ad, beacon, and either. We summarize these results for the CAM trace in Figure 3.12. We see that, the probability of encountering an online ad or beacon after three clicks is close to 90%! We repeated the same experiment for the other traces and observed very similar trends even for our mobile trace. We also want to stress the fact that Figure 3.11 represents user requests from all users (IPs) grouped together, whereas Figure 3.12 focuses on what happens to different users.

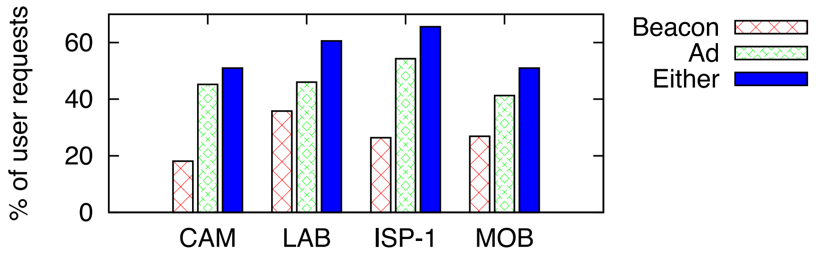


Figure 3.11: The percentage of user requests to web pages with beacon, ad, and either in our four traces.

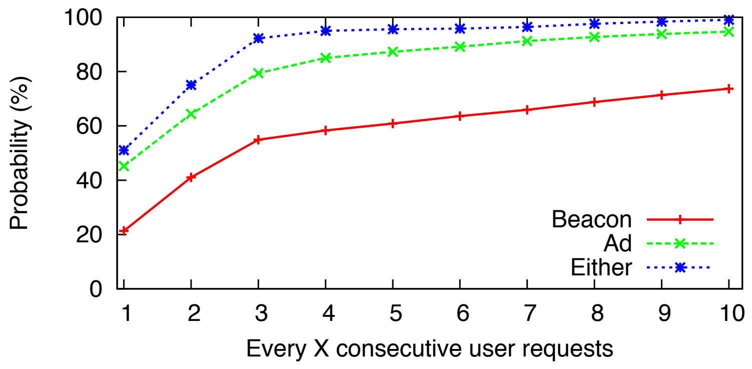


Figure 3.12: The probability of encountering at least one beacon, ad, and either in every X consecutive user requests in the CAM trace.

3.4.4 Inter-page time

Web users in the mobile trace spend 3X more time on pages compared to the ones on the wireline traces. Click-through streams can be used to estimate how much time people "stay" on a web page. We define inter-page time as the time difference between two consecutive user requests in click-through streams. Therefore, the last user request of click-through stream does not have inter-page time. Inter-page roughly approximates the time a user spends on a page. In Figure 3.13, we present the CDF of inter-page time in different traces. The median of page staying time for wireline traces (CAM, LAB and ISP-1) and MOB mobile trace are about 14s and 45s, respectively. One possible reason for this difference is that surfing on smartphones takes much longer because of the more limited bandwidth in 3G networks. Another contributing reason could be the smaller screen size on smartphones which necessitates scrolling.

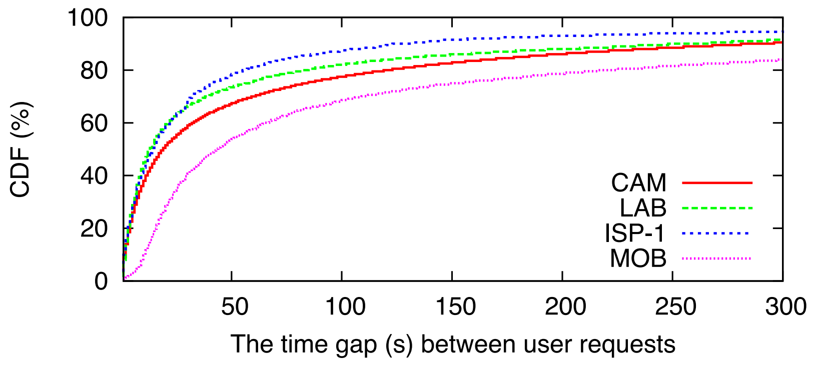


Figure 3.13: The inter-page time in different traces

3.5 Related work

The recent growth of network services provided over HTTP has been attracting the interest of the research community. Labovitz et al. [70] brought to light the fact that most inter-domain traffic is HTTP. Schatzman et al. [100] present a methodology to identify web-based mail servers, and distinguishing between services, such as Gmail and Yahoo mail. Erman et al. [42] analyze traffic from residential users and find that a significant part of

HTTP traffic is generated by hand-held devices and home appliances, while a large fraction is machine generated (e.g., OS/Anti-virus updates, ads). Li et al. [73] present methods to identify the type of the object transferred over HTTP (e.g., video, xml, jpeg). The recent work from Schneider et al. [101] characterize the inconsistencies between observed HTTP traffic and what is advertised in its HTTP header. Bermudez et al. [18] understand the tangle between the content, content providers, and content hosts based on DNS traffic. All this previous work try to understand HTTP traffic from different perspectives, but they do not focus on the reconstruction of web-surfing at the user request level from HTTP traffic as we do here.

Regarding the problem of reconstructing web-surfing activity, existing work usually fall into one of four categories. In the first category, people assumes that any HTTP request for an HTML object is the head HTTP request of a user request [14]. The second category of methodologies are based on the timing information of HTTP requests: if the idle time between two HTTP requests is smaller than a predefined threshold, they belong to the same user request [81, 108]. Both methods were effective in the early days of the web, but are no longer due to the complexity of modern WWW. The third category is based on web analytics beacons. A representative example is `StreamStructure` in [53]. To understand the evolution of modern web traffic and web-pages, Ihm et al. proposed a web analytics beacon based method to detect “primary" web pages requested by users from web proxy server logs. A key limitation of `StreamStructure` is its dependence on web analytics beacons, which seem to form the basis of their reconstruction algorithm without which accuracy drops significantly. Recall that as high as 80% of user requests may not contain any analytics beacons (see

Figure 3.5). The last category relies on the patterns in the requested URI. E.g., Schneider et al. [102] proprosed to reconstruct user requests (user actions) in several online social networking sites by matching previously compiled patterns with URI. This kind of method is customized for target websites and hard to be generalized because of various website architectures.

The click-through streams are also studied in the previous work. To understand the behavior of searching the Web, Kammenhuber et al. in [59] extract issued search term, search results returned from Google and the subsequent clicks on results from network traffic traces. The reconstructed sequences of clicks after the query is used for profiling users' prevalent search patterns. Schneider et al. [102] extract click streams within online social networks from HTTP traffic to characterize the interaction between users and social network websites. However, both studies characterize user browsing behaviors on a specific kind of websites. Instead, our study shows user browsing behaviors in general.

Web analytics beacon is pervasive in modern web pages [83]. Krishnamurthy et al. [69, 68] study the privacy issues arising from the wide use of web analytics beacons in web pages. Ihm et al. in [53] show web analytics beacon can be used to detect pages and understand modern traffic.

3.6 Conclusion

Network traffic has been increasingly dominated by web traffic and HTTP protocol has become the most prevalent means for applications to provide their services. In this paper, we framed and addressed a relatively novel problem: reconstructing web-surfing behavior from web traffic. The problem is far from trivial given the complex and interconnected websites of today. We made two key contributions. The first is developping a systematic approach ReSurf which can reconstruct user requests with more than 95% precision and 91% recall. As our second contribution, we showcase interesting results that one can obtain from raw network traffic using ReSurf. We observed that web users are continuously being exposed to advertising and tracking services and that in our traces 50-60% of user requests (think clicks) interacts with tracking or advertising services. Another surprising result is the “shallowness" of the click-through stream of users accessing websites. The the majority of streams have the maximum length of two. This behavior suggests a more focused usage of the web, where users have a specific goal in mind and are less likely to click on links that take them to irrelevant websites. In conclusion, we believe that ReSurf represents an enabling capability for ISPs, network administrators, and researchers that want to model and understand how users surf the web.

Chapter 4

Scanner Hunter: Understanding HTTP Scanning Traffic

4.1 Introduction

HTTP scanning is a fascinating and lesser known activity that aims to discover the security weaknesses of websites and can be seen as a first and exploratory step that may enable subsequent website infiltration. In HTTP scanning, malicious entities probe a website for particular resources that are promising for exploits and may be found in very specific places on the websites. These resources can be files with security-sensitive information and web interfaces for authorized users such as *phpmyadmin.php* as shown in Figure 4.1. The amount of requests sent by scanners is small compared to the traffic the websites received because they only probe for very specific files, therefore detecting them is a needle-in-the-haystack problem. At the same time, the risk that scanners pose is high. For an example, an intrusion attempt that seems to have been enabled by scanning is carried out by the hacktivist group NullCrew, which broke into a Department of Homeland Security website after identifying files that should have been invisible to the general public [32].

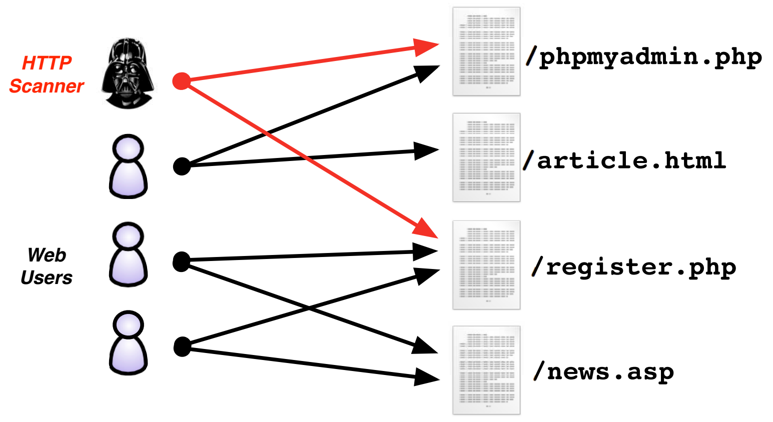


Figure 4.1: Visualization of an HTTP scanner among legitimate users.

The motivation for our work is the little attention that the problem has received so far and the ever-increasing importance of website security. First, there is currently no existing solution for this problem. Second, scanning is a first exploratory step towards compromising a website. Third, having a clear understanding of scanning behaviors could help administrators secure their websites and provide assistance during forensics analysis. Fourth, understanding what scanners are looking for can reveal what are the preferred vulnerabilities. Finally, understanding the spatiotemporal dynamics of these scanners can provide insights on ecosystem that enables them.

The main focus of this work is detecting and understanding the behavior of HTTP scanners. Given web traffic logs, the desired output is a list of IPs that potentially engaged in HTTP scanning activities. Naturally, the goal is detect all scanners with minimal false alarms. This is a challenging problem for the following reasons: (a) scanners usually generate few requests, (b) it is not easy to distinguish between a scanner and a normal web user, and (c) the number of scanners is significantly smaller than the number of legitimate users, leading to a *needle in the haystack* problem. We substantiate all these claims in later sections.

Surprisingly, this problem has attracted very little attention from the research community. In fact, ours is arguably the first work focusing on this problem. At the same time, we want to make clear that we what we do is different from identifying: (a) web crawlers, whose goal is to crawl and map out the entire website, (b) network-level scanners such as IP or port scanners, (c) web application penetration attempts, which is the step following a scan that successfully discovers security vulnerabilities, and (d) compromised botnet nodes inside one's network. In section 4.5, we provide a more in-depth discussion of earlier work in these areas, and how they differ from the problem at hand.

In this paper, we present Scanner Hunter, arguably the most effective method to detect HTTP scanners, and study scanners and their behavior over six months. The key novelty of Scanner Hunter is that it is a graph-based approach that overcomes the *needle-in-the-haystack* problem by zeroing on the collective actions of scanners, who search for similar resources. In addition, we provide an extensive study of HTTP scanning activities over a six-month period to understand their scanning patters and their spatiotemporal behavior. We use real web-logs collected from a University campus from March 2012 to September 2012 with over 1.9 billion HTTP requests from 12.8 million external IPs.

Note that, to enable further research in this novel direction, we will share our data, appropriately anonymized, for research purposes at the time of publication.

1. Scanner Hunter detects scanners with 96.5% precision. Our approach identifies malicious communities of IPs in an appropriately constructed bipartite graph that captures the interactions between two sets of entities: the likely malicious IPs and the resources[1] they have requested. Using a soft co-clustering technique, our approach identifies groups of IPs and finally uses a labeling step to separate groups that engage in scanning from those of benign users. The use of clustering techniques on the graph makes the detection more accurate: scanning behaviors are easier to recognize in a group than in individual IPs, especially when scanners are often stealthy. As a proof of concept, we conducted experiments with simple heuristics and standard Machine Learning techniques and observe that they only achieved 19.1% and 54.3% precision respectively, compared to 96.5% for Scanner Hunter. Even though a False positive rate of 3-4% may seem high, we want to stress that detecting scanners from millions of users at such a high precision is not trivial. We will provide more details about the experiments in Section 4.3.

2. HTTP scanning is widespread: 4,000 scanners per week for a medium-sized University network. We conducted an extensive study on scanning activities for a period of six months to understand: (a) their spatial and temporal properties, (b) their mechanisms and the effort to evade detection, and (c) what they are looking for. We highlight the most interesting observations.

a. Intensity: Roughly 4,000 unique IPs scan the websites hosted by the University

[1]A resource in this context is a file of any type ***and*** the full path leading to it. For an example, `/phpmyadmin/scripts/inc.php`

every week. At the same time, 80% of the IPs that appear every week have never been seen before.

b. Spatial distribution: Scanners are diversely placed across the IP space, which suggests that the IP prefix-based blocking of scanning activities would require a large and fine-grained filter set.

c. Complacency: Scanners are not concerned about getting caught. For example, more than one third of the scanners used an unusual User-Agent in their requests and this User-Agent is `mozilla/4.0`.

d. Categories of scanners: We identify four major categories of scanners based on the resources they look for: (i) user registration and login pages, (ii) website management interfaces, (iii) pages with potential vulnerabilities that may allow activities like remote code execution, and (iv) compressed web archives such as `wwwroot.zip`, which are the products of website backup activities and should not be publicly visible.

The remainder of the chapter is organized as follows. In Section 4.2, we present background information and the methodology behind Scanner Hunter. In Section 4.3, we detail our datasets and the experiments we performed on the datasets to evaluate the performance of Scanner Hunter. In Section 4.4, we provide our extensive study of scanners' behaviors. We provide a brief overview of what work has been done in the general area of vulnerability scanning and how they differ from our work in Section 4.5. We conclude with Section 4.6.

4.2 Methodology

In this section, we first provide some definitions and background and then present our method, Scanner Hunter.

4.2.1 Definitions and background

An **HTTP request** is usually either a `POST` or `GET` request. A `POST` request sends information to the target website (e.g. filling an online form) and a `GET` request retrieves information from the website. When a remote web server receives a HTTP request, the server will process it and return a *response* along with a response code that indicates whether the request succeeded.

A **failed HTTP request** is indicated by response code `40X`, where `X` is a single-digit number. For instance, the response code `404` indicates that the resource does not exist and `403` says that the server is refusing to return the requested resource [47]. Here, we refer to requests that trigger a response with code `40X` as *failed HTTP requests*, and the others, *successful HTTP requests*.

A **User-Agent** is a string contained in the `User-Agent` field of an HTTP request that provides some description about the application that generates the request. A typical value is the name of a web browser.

The resource requested by an HTTP request (`example.com/inc/login.php?user=admin`) usually has four separate components: a domain name (`example.com`), a directory name (`/inc/`) that contains the file sought by an HTTP request, the name of the file (`login.php`), and a set of parameters (`user=admin`) that may alter the server's response.

We define a **Re-Path**, which stands for ***Resource-Path*** to be the portion of an URL that contains both the directory path and the name of the file. For example, the Re-Path in the example is `/inc/login.php`.

Although we have already defined **HTTP scanners**, here we want to highlight some interesting behaviors that we will exploit in detecting them. Note that these behaviors on their own are not sufficient without the additional techniques of our method, which we will describe in the next section. Recall that scanners explore a website in a targeted fashion and their probing is a preparation for a subsequent exploitation. The behavior of scanners exhibits the following characteristics: 1) they search blindly for resources of interest and so many of their HTTP requests will fail, and 2) they do not request or even download embedded objects that could not easily lead to exploits like `jpg` images.

Benign crawlers are automated programs chiefly used by search engines to index the Web. They are different from HTTP scanners in that:

- They only follow existing links instead of actively searching for resources that may or may not exist.
- They state what they are in the User-Agent field (e.g. Googlebot, Yahoo! Slurp, etc).
- The IP of the machine running the crawler program is usually registered by the company it belongs to. More specifically, the name obtained through a reverse-DNS query would match what is shown by the User-Agent.
- They would (optionally) request the `robots.txt` file from the website that contains instructions as to where they're allowed to crawl.

A **stealthy crawler** is similar to a benign crawler in behavior except that it customizes its User-Agent field to make it look like their HTTP requests come from a regular web browser. There already exists work devoted to the detection of stealthy crawlers [55], and it is not a focus for our work.

4.2.2 Methodology

To identify HTTP scanners, there are four main steps as described in Figure 4.2 in Scanner Hunter methodology. Scanner Hunter executes them as follows.

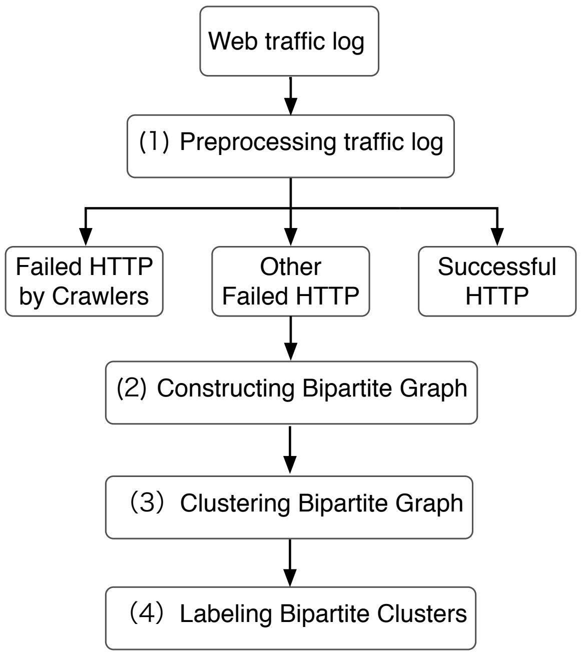

Web traffic log
(1) Preprocessing traffic log
Failed HTTP by Crawlers
Other Failed HTTP
Successful HTTP
(2) Constructing Bipartite Graph
(3) Clustering Bipartite Graph
(4) Labeling Bipartite Clusters

Figure 4.2: The main operations in Scanner Hunter

- ***Preprocessing***: Scanner Hunter processes the web traffic logs to filter out requests unlikely to be from HTTP scanners.

- ***Bipartite graph construction***: Scanner Hunter captures the suspicious HTTP requests in a bipartite graph where one set of vertices correspond to IPs and the other to the Re-Paths they requested.
- ***Co-clustering***: Scanner Hunter uses a soft co-clustering algorithm to group the IPs into communities where IPs in the same community have similar behaviors.
- ***Community labeling***: Scanner Hunter inspects each community output from previous step and label each of them as a community of users or scanners.

We now describe the working and rationale of each step in detail. At the end, we will also explain why each step is necessary for achieving high precision in identifying scanners.

1. Preprocessing: Given the input in the form of web traffic logs, which may be collected by passively monitoring HTTP traffic or collected from web server logs (like Apache log files), Scanner Hunter reads in all HTTP requests from the logs and separate them into three different categories: successful HTTP requests, failed HTTP requests generated by web crawlers, and other failed HTTP requests. As their names suggest, the first category contains only successful HTTP requests, the second failed HTTP requests generated by known and benign web crawlers, and the third all other failed HTTP requests.

There are several reasons for including this step. First, popular websites receive an extremely high number of HTTP requests each day and processing all of them would consume a lot of time and resources. Second, we have seen from manual inspection, that most scanners are very aggressive in probing for vulnerabilities, and they produce a large number of failed requests as we also mentioned earlier.

In addition, we need to filter out failed requests that may come from benign crawlers, because they may generate them in the process of following all possible links whether they may be broken or not. Furthermore, the same crawlers may frequently re-check the web pages they have indexed before even after those same web pages have been removed. To ensure that Scanner Hunter does not confuse the behaviors of benign crawlers with those of scanners, we first attempt to detect the presence of well known and benign web crawlers in the data according to the properties we mentioned in Section 4.2.1. We consider an IP to be a benign crawler if:

1. The User-Agent fields of its requests indicate that the application that generated the requests is a benign crawler (Googlebot, Yahoo! Slurp, etc.) and the IP has requested the file `robots.txt`.

2. The information collected from reverse-DNS lookup of the IP indicates that the IP belongs to the organizations that User-Agent claims it is. For example, if an IP advertises itself as a Googlebot and does indeed come from an IP prefix owned by Google according to the reverse-DNS lookup, we consider it as a benign web crawler.

2. Bipartite graph construction: In this step, Scanner Hunter constructs a weighted undirected bipartite graph $G = \langle I, P, E \rangle$ where I is the set of IPs that produced the failed HTTP requests, P the set of Re-Paths that the IPs requested unsuccessfully, and E the set of edges. An edge $e \in E$ between IP $i \in I$ and Re-Path $p \in P$ if and only if IP i requested the Re-Path p but failed to access it during the monitoring interval. Figure 4.3

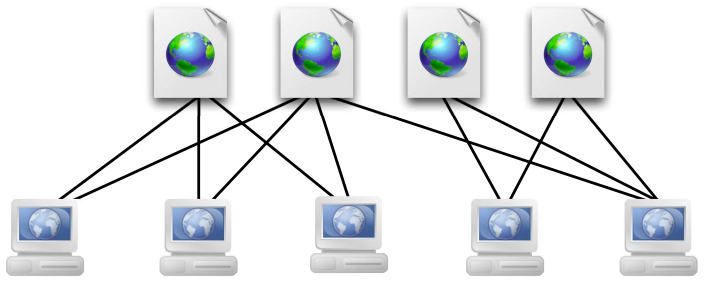


Figure 4.3: An example of bipartite graph

shows an example of such bipartite graph.

We associate weights to the edges of the graph in order to capture their interactions in more detail. We consider three approaches for defining edge weights:

(i) Uniform weight for all edges. In this case, the weight of each edge e between IP i and Re-Path p, $w(e)$, would be 1.

(ii) The weight of the edge e is the number of times that i requested p (**request count**).

(iii) Same as (ii), but we normalize the weight of each edge connected to p by dividing its request count by the sum of all request counts of all the edges connected to p. More precisely, if $\{e_1, e_2, ..., e_k\}$ are the edges incident to p and $\{n_1, n_2, ..., n_k\}$ are the respective request counts, $w(e_j) = \frac{n_j}{\sum_j n_j}$

The intuition behind weight function (iii) is that we want to minimize the importance of any path p that is too popular by decreasing the weight between p and any IP i that requested it. This can account for the cases where a popular website becomes unavailable.

Our experiments show that weight functions (i) and (ii) achieved comparable precision, which is higher than that of (iii). That fact that (iii) did not perform better than the other two is surprising and counter-intuitive, but upon further inspection, we noticed that the probability of a popular website in our data being unavailable is relatively low, though non-zero. We adopted (i) because of its simplicity. Furthermore, the community structure, which we discuss below, in the bipartite graph seems sufficient for detection and the frequency of request does not seem to encode any information.

3. Co-clustering: Given the bipartite graph G generated in the previous step, Scanner Hunter uses a soft clustering algorithm to partition G into a set of communities $C = \{c_1, c_2, \ldots c_n\}$. We adapted and used an algorithm inspired by Phantom [64], which we tailored to our problem. Each community c_i is a bipartite subgraph of the original G. Therefore, $c_i = \langle I_i, P_i, E_i \rangle$ where I_i is the set of all IPs in c_i, P_i the set of all Re-Paths in c_i, and E_i the set of all edges exclusively between I_i and P_i. We expect the output communities (used interchangeably with bipartite subgraphs from now on) from the soft clustering algorithm to have the following properties:

- The IPs (I_i) in each community are well-connected to each other through the Re-Paths (P_i) they requested.
- The soft-clustering approach may put an IP in more than one community because the IP may request many different Re-Paths.

The input to our adapted co-clustering algorithm is the bipartite graph G, represented by a weighted adjacency matrix M. In the beginning, the algorithm considers the graph as a single community that consists of all of the IPs and Re-Paths. For each community c:

1. The algorithm leverages a Singular Value Decomposition (SVD) technique to figure out how to best "cut" c into two child communities c_a and c_b so that a vertex in c_a will be more strongly connected to the other vertices in c_a than with those in c_b.

2. The algorithm computes the ***cohesiveness*** value of each of the two child communities. The cohesiveness of a child community indicates how well separated it is from its sibling.

   Given a child community c_a and its sibling c_b, let $\gamma(c_a)$ and $\gamma(c_b)$ be the cohesiveness of c_a and c_b respectively, and they are calculated as follows. If E_c is the set of edges contained in the initial community c and E_{c_a} the set of all edges contained in c_a, $\gamma(c_a) = \dfrac{|E_{c_a}|}{|E_c|}$.

3. Let T_γ be a predefined threshold for the cohesiveness value that dictates when Scanner Hunter would stop dividing communities. More specifically, if $\gamma(c_a) \geq T_\gamma$ and $\gamma(c_b) \geq T_\gamma$, the algorithm will retain both c_a and c_b and proceed to check if each of them can be divided further into smaller communities. Otherwise, the community c will not be divided further.

4. The algorithm stops when there is no community that can be subdivided further.

The curious reader can consult [64, 28] for details.

4. Community labeling: Even legitimate users may form communities in a bipartite graph because some popular web pages or embedded objects may become unavailable by accident. For this reason, in this step, Scanner Hunter inspects each community individually and employs a heuristic to determine whether the community are of benign

IPs or HTTP scanners. To this end, we carefully studied the scanners' activities to find potentially useful metrics. After many experiments, we narrowed down to two metrics that gave high precision: **average HTTP failure ratio** and **average embedded objects ratio**. The two metrics in combination capture the behavior of the scanners sufficiently for our algorithm to achieve high precision. Even though legitimate users may produce failed requests, their failure ratio is usually low. Moreover, scanners tend to not download embedded objects such as images or video and instead focus more on resources like `php`, `asp`, and `mdb` (database) files, which are more likely to be exploited.

We now define the average failure ratio and average embedded object ratio of a community c to capture the difference between communities of scanners and legitimate users.

- For IP address i, let T_i is its number of requests, and F_i its number of failed requests. Its failure ratio, $FR(i)$, is given by: $FR(i) = \frac{F_i}{T_i}$
- Let N_i be the number of requests that IP i made for non-HTML document type resources (for example, images, javascript objects, etc.), the embedded-object ratio for i is: $ER(i) = \frac{N_i}{T_i}$
- The average failure ratio and the average embedded-object ratio of all IPs in a community c are respectively:

$$FR(c) = \frac{1}{|c|}\sum_{i \in c} FR(i), \quad ER(c) = \frac{1}{|c|}\sum_{i \in c} ER(i)$$

If there exists a community c where $FR(c) > T_{FR}$ and $ER(c) < T_{ER}$, Scanner Hunter will label c as a community of HTTP scanners. All IPs in these community are considered as HTTP scanners. This heuristic is successful in excluding communities that may be formed by many IPs requesting misplaced resources. We will explain the rationale behind this heuristic in the following section.

We also would like to stress that ***both*** co-clustering and labeling are integral to the performance of Scanner Hunter because i) inspecting individual IPs and relying on the IPs' failed requests alone will generate a high amount of False Positives (which we will show in Section 4.3 and ii) blindly labeling each community after co-clustering will also lead to mislabeling benign IPs as scanners, as we have explained in Step 4 of Scanner Hunter.

4.3 Performance and evaluation

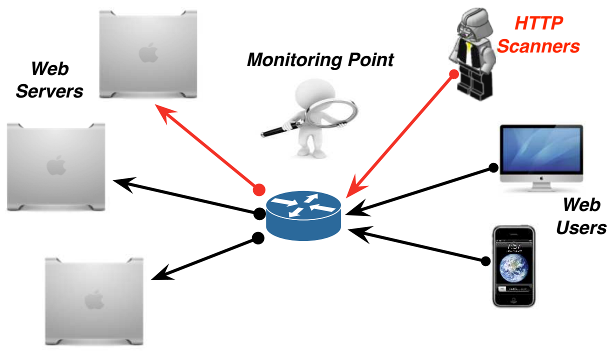


Figure 4.4: The setup of web traffic log collection

In this section, we present: the dataset we used for our study, how we select the proper values for parameters in Scanner Hunter, and how evaluate the performance of Scanner Hunter in terms of Precision and the number of detected HTTP scanners.

4.3.1 Dataset

Our real-world dataset consists of 28 weeks of Web traffic collected from a University campus network. The setup of data collection is shown in Figure 4.4. Our data collection tool was deployed at the only edge gateway router connecting the University to the Internet, which all incoming and outgoing Web traffic passes through. Note that we are only focusing on incoming HTTP requests, e.g. requests towards websites internal to the University, and their associated responses. For each IP packet on TCP ports 80, 8000, and 8080 of the University's servers, the tool performs Deep-Packet Inspection (DPI) to tell whether it is a HTTP request or response. If so, it logs the following important fields for each HTTP transaction: timestamp, client IP, server IP, URL, referrer, content-type, content-length, response code and User-Agent.

In Table 4.1, we provide the statistics of the entire dataset, which contains 1.9 billion of HTTP transactions from 12.8 million external IPs to about 1096 different internal websites. In this paper, we define a website by the two top-level domain names of its host name. For instance, we count `a.c.com` and `b.c.com` as the same website. The websites in our dataset are hosted by the University and most of them are online forums powered by Bulletin Board Systems (BBS) and department websites.

We also present a number of weekly averages because we apply Scanner Hunter on the weekly partitions rather than the full dataset. We will discuss the motivation for partitioning data into weeks in the next section.

Metrics: We use two metrics to measure Scanner Hunter's performance: **P**recision and the total number of labeled scanners. Given the numbers of True Positive (TP), False

Metric	HTTP requests	Failed HTTP requests	Websites	External IPs	IPs with failure(s)	Web crawler IPs
Per week	70.6M	7.5M	1029	929.6K	409.8K	10119
All	1976.8M	210.2M	1096	12.8M	805.6K	57408

Table 4.1: The web traffic traces used in our study

Positive (FP), the Precision rate, which is the fraction of IPs we label as scanners that are in fact scanners, is calculated as follows: $P = \frac{\text{TP}}{\text{TP} + \text{FP}}$.

Validation. Given that we do not have the ground-truth in our dataset, we use sampling and manual verification to assess the accuracy of identification. We manually assess if each IP is an HTTP scanner or not based on all its HTTP transactions during the monitoring interval. Specifically, we consider its User-Agent, the Re-Paths it asked for, its failure ratio, its embedded-object ratio, the referrers for each of its requests, and the HTTP response codes that it received when the requests themselves failed. We are very conservative in our manual verification and only label those IPs with very obvious scanning activities as scanners.

To evaluate the precision, we randomly select 300 IPs from labeled scanners by Scanner Hunter, then manually verify them as described before, and label an IP as a True Positive, if it clearly exhibited the characteristics of an HTTP scanner. Otherwise, we count it as a False Positive.

4.3.2 Parameter selection

a. Cohesiveness threshold T_γ: The soft co-clustering algorithm has the cohesiveness threshold T_γ as its sole parameter. To recap, the cohesiveness threshold determines

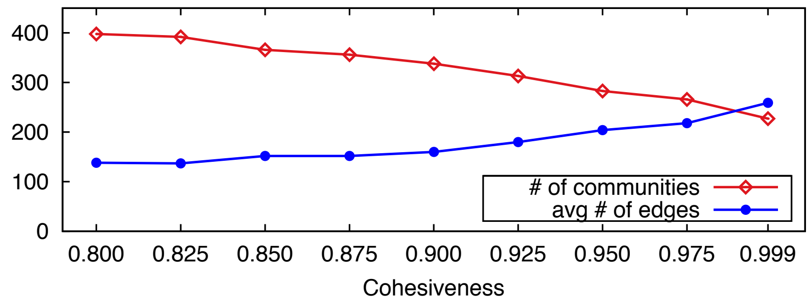


Figure 4.5: The total number and average size of non-trivial communities with varied cohesiveness.

when the algorithm should stop subdividing a community. As Figure 4.5 shows, the lower the value of cohesiveness, the co-clustering algorithm creates more smaller but more strongly connected non-trivial communities. A non-trivial community is one in which there is more than one IP and one Re-Path. Essentially, lower cohesiveness value leads to the creation of communities in which the IPs are more similar in behaviors because they have requested similar resources.

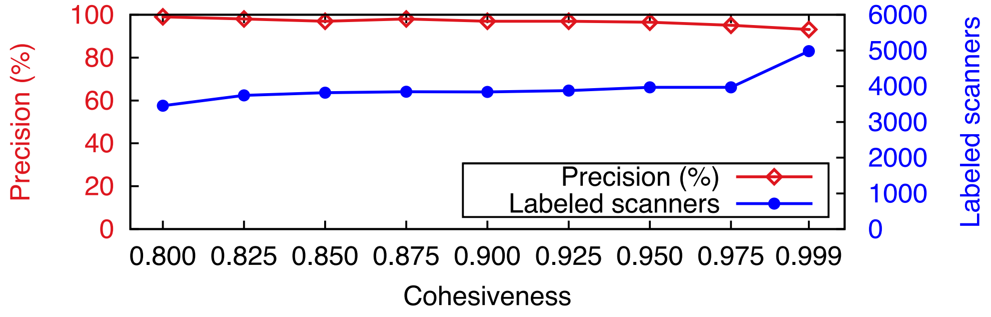


Figure 4.6: Precision and the number of IPs labeled as scanners with varied cohesiveness

To determine the right threshold value, we conducted an experiment where we varied T_γ and tried to assess the performance of Scanner Hunter on the training partion d. For each value of T_γ shown in Figure 4.6, we can see that when we set the value of T_γ

to 0.975, we achieve a good balance between precision and the total number of labeled scanners. This is the value that we use in the rest of this paper.

b. Embedded and failure ratios: Once we have the communities, Scanner Hunter performs its labeling step. To recap, given a community $\mathbf{c_i}$ in the set of all communities $\mathbf{C}$, the community's failure ratio is $\mathbf{FR}(\mathbf{c_i})$ and $\mathbf{ER}(\mathbf{c_i})$ its embedded-object ratio. Let T_{FR} and T_{ER} be thresholds such that if $FR(c_i) > T_{FR}$ and $ER(c_i) < T_{ER}$, we label c as a community of HTTP scanners and c_i as a community of users otherwise. We will show below the process that helps us decide the values for T_{ER} and T_{FR}.

We first picked one weekly partition of our data as a training dataset, henceforth referred to as d, and ran Scanner Hunter's clustering step on d with a cohesiveness value of 0.975. Given the set of communities $C = \{c_1, c_2, ..., c_n\}$ produced by this step, we represent each c_i as a pair of coordinates (x_i, y_i) where $x_i = FR(c_i)$ and $y_i = ER(c_i)$ and plot the set C as a heat map as shown in Figure 4.7.

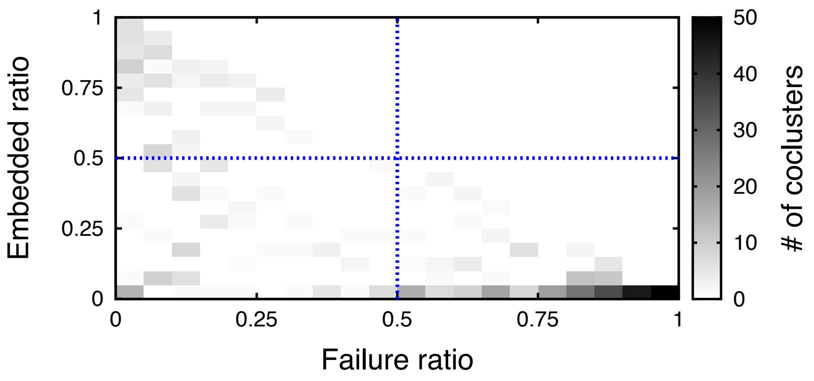


Figure 4.7: T_{FR} and T_{ER} of communities generated by a cohesiveness value of 0.975

Recall that an HTTP scanner tends to have a higher failure ratio and a lower embedded-object ratio than a legitimate user because the scanner does not know exactly where the vulnerable resources reside and does not want to waste bandwidth downloading objects it does not need. This observation, in turns, implies that the communities located in the **lower right** quadrant of the heat map in Figure 4.7 would be the most suspicious.

We begin with the community with highest failure ratio and lowest embedded-object ratio and set T_{FR} and T_{ER} to this community's values, in essence defining a rectangular area limited by the lines $y = T_{ER}$, $x = T_{FR}$, $x = 1.0$, and $y = 0$. We then manually inspected each community inside this area to judge whether each IP in that community is an HTTP scanner before we expanded the area by increasing T_{ER} and decreasing T_{FR}. What we observed in this process is that by setting T_{ER} around 0.5 and T_{FR} around 0.5, Scanner Hunter achieved a reasonable balance between the number of correctly identified scanners and the number of mislabeled legitimate users. For simplicity's sake, we set $T_{ER} = 0.5$ and $T_{FR} = 0.5$.

We also repeated this process with different values of cohesiveness and came to the same conclusion each time: with 0.5 and 0.5 as the values for T_{ER} and T_{FR} respectively, Scanner Hunter achieves a high level of accuracy.

4.3.3 False positives

We wanted to further investigate why our approach misclassifies some IPs as HTTP scanners. Our manual verification process helped in this investigation. 89% of all misclassified IPs belong to stealthy crawlers masquerading as regular users. Upon manual inspection, these IPs downloaded very few embedded objects when they crawled a small number of pages, some of which happen to be unavailable due to broken links. The resultant high failure ratio and low embedded-object ratio confused our algorithm.

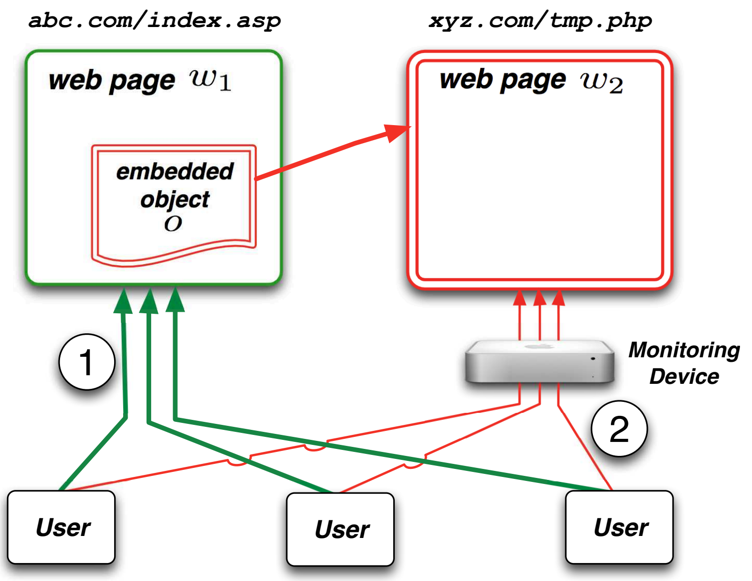


Figure 4.8: The reason for False Positives

The remaining 11% of the False positives are legitimate users. The reason Scanner Hunter identifies them as scanners can be summarized in Figure 4.8. Let's say that there exists a web page w_1 at the URL `abc.com/index.html` and there is an embedded object o on w that is hosted by website w_2 at the URL `xyz.com/tmp.php`, where our monitoring tool is placed. In the case that w_1 is very popular and o is suddenly unavailable, all of the requests intended for o will consequently fail.

Because our monitoring point is at w_2, it would appear as if a large number of users are requesting a non-existing resource because our monitoring tool is not aware of w_1. As a result, Scanner Hunter will group the IPs requesting o into a community in which the failure ratio and embedded-object ratio exceed the predefined thresholds. All of the IPs in this community will then be labeled as HTTP scanners. This is the reason why Scanner Hunter mislabeled that very small percentage of benign IPs in our dataset.

4.3.4 Measuring the precision

We used one week's data to establish the values of the three parameters of our Scanner Hunter, namely cohesiveness threshold T_γ, embedded-object ratio threshold T_{ER}, and failure ratio T_{FR}.

Scanner Hunter detects scanners with 96.5% on average. We assess the performance of our approach on five other weekly data partitions in Figure 4.9 (top flat red line). On average, the precision of Scanner Hunter is 96.5%. This is a success, given that there are nearly one million external IPs in each weekly data partition. We validated these Precision rates in through sampling and manual inspection, using the same process as in the training set.

Figure 4.9 (lower jagged blue line) shows the total number of IPs labeled as scanners by Scanner Hunter for each of the weekly partitions. Note that the sharp drop in identified scanners on the 8^{th} weekly partition is an artifact of the data collection: for three days out of seven of the 8^{th} week, there web-records were lost.

4.3.5 Baseline approaches

In this section, we discuss two baseline approaches: simple heuristic and machine-learning algorithm. Both of these methods develop profiles and compare IPs in isolation, in

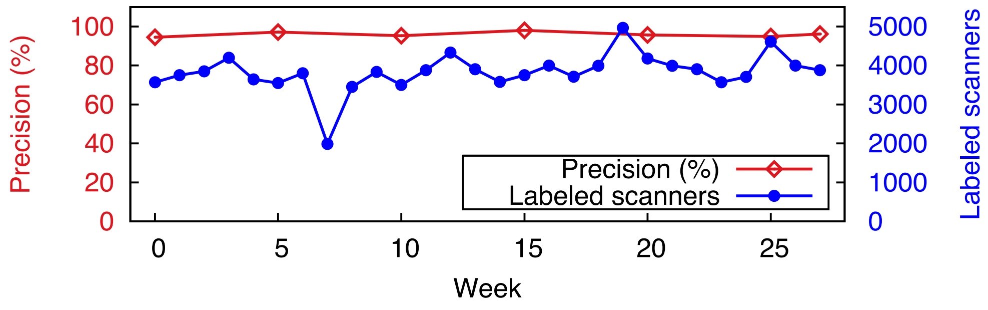


Figure 4.9: Precision of and the total number of detected HTTP scanners in different weeks

contrast to our graph-based approach.

First, the simple heuristic decides if an IP is a scanner only based on its failure and embedded-object ratios. Second, the machine-learning (ML) approach uses the HTTP requests of scanners labeled by Scanner Hunter as training data for future classification. Note that we are not aware of any other method to obtain a large training set. We introduce both approaches in order to provide some insight into the difficulty of the problem Scanner Hunter was designed to solve.

Both the simple heuristic and the ML approach perform poorly (19.1% and 54.3% respectively), roughly half of that of Scanner Hunter. We attribute this to the use of communities that helps create more robust profiles compared to profiles per IP address.

1. Simple heuristic: The simple heuristic determines if an ***IP*** i is an HTTP scanner or not based on the two metrics: $FR(i)$ and $ER(i)$. Specifically, if $FR(i) > T_{FR}$ and $ER(i) < T_{ER}$, then i is labeled as a scanner. Recall that Scanner Hunter determines whether a community is one of HTTP scanners based on the community's failure ratio FR and embedded-object ratio ER. Here we use the same thresholds, that is, $T_{FR} = 0.5$ and

$T_{ER} = 0.5$.

For evaluating the simple heuristic, we randomly selected a weekly data partition D_i and applied the heuristic on it. We obtained the Precision rates in the same way that we described in Section 4.2.

2. Machine-learning algorithms: From the machine leaning software collection WEKA [116], we selected three commonly used algorithms: Support Vector Machine (SVM), K-Nearest Neighbors (K = 1, 3, 5), and Decision Trees.

Features. For each algorithm, we consider a total of 15 features: failure ratio, embedded-object ratio, suspicious referrer ratio, the number of and the ratio of requested non-existing Re-Path, the maximum, average, and minimum size of connected component in the referrer graph, the maximum number of consecutive failed HTTP requests, the Inter-Arrival Time (IAT) between two consecutive failed requests, and the number of retries after a failure. Using WEKA's feature elimination capability, we narrowed the set of 15 features down to a total of 9: failure ratio, embedded-object ratio, suspicious referrer ratio, the ratio of requested non-existing Re-Paths, the maximum size of connected component in referrer graph, the maximum number of consecutive failed HTTP requests, the IAT between two consecutive failed requests, and the number of retries after a failure.

To evaluate the ML algorithms, we picked two consecutive weekly data partitions D_i and D_{i+1}, the former for training and the latter for testing. We first applied Scanner Hunter on D_i to obtain a list of n potential scanners and selected also n IPs determined to be benign by Scanner Hunter; we call this first set M and the second B. We then used the HTTP requests produced by each of the IP in both sets to train the ML algorithms, which

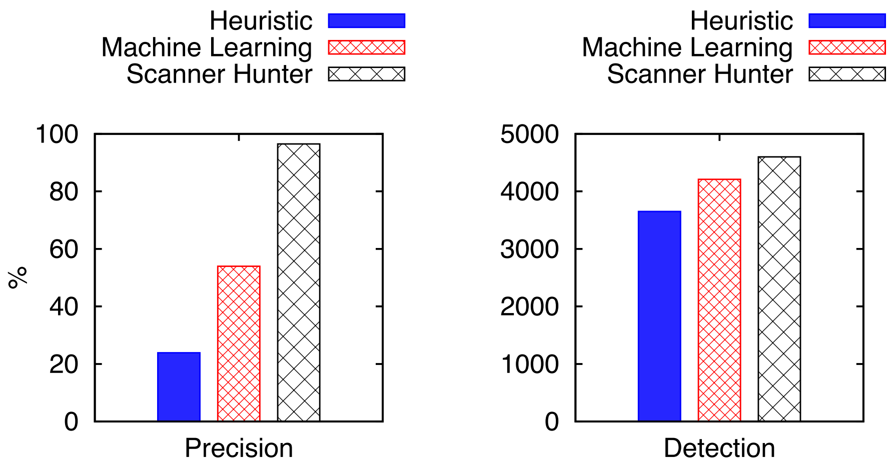


Figure 4.10: The precision of Scanner Hunter and the baseline approaches.

were then used to classify the IPs in the data partition D_{i+1}.

Note that the IPs in set M do not include all the IPs labeled as scanners by Scanner Hunter in D_i. We have to exclude some IPs from Scanner Hunter's results because the ML algorithms require that there are enough HTTP requests produced by each IP (in this case, at least 5) to be effective. Each of the IPs in M then had at least 5 HTTP requests associated with them.

From our experiments, three ML algorithms achieved comparable performances. The Decision Tree algorithms (more specifically "Adtree") in fact produced the highest precision rate. From this point forward, we will use the generic term "ML learning" algorithm to refer to Decision Trees in our performance comparisons.

We can see from Figure 4.10 (left) that the precision rates for the simple heuristic and ML approaches are only 19.1% and 54.3%, respectively. This once again underscores an important observation we have provided earlier in the paper: that scanners and users themselves are diverse in their behaviors suggests that simple approaches that focuses only on an individual classification target are not sufficient and shows the need for a more advanced, graph-based solution.

4.3.6 Discussion

Why we partitioned the data into weeks. Since the University that provides us with data is only a campus of medium size, it is of limited visibility to the outside world. Although we could have split the data into daily partitions instead, the amount of scanner traffic each day would be small and therefore there would be limited information available to the co-clustering algorithm, which is a very important component to Scanner Hunter. Without enough HTTP requests, there would be more scanners who would be isolated from one another and fewer communities of them. As a result, even though Scanner Hunter would still retain very high precision rates, the amount of scanners detected would be much lower.

If a network administrator wants to use Scanner Hunter, we would advise them to tune the monitoring interval according to the size of their network, as a large enough one may attract enough requests from scanners to enable a finer-grained partitioning of data. They should also think more carefully about the trade-off between the need for the early detection of scanners and the ability to identify more of them.

Scanner Hunter is light-weight. Scanner Hunter consumes the most time in step 3 in methodology: co-clustering a bipartite graph into communities. However, it takes no more than 15 minutes for Scanner Hunter to extract the community information for a week's worth of data on a 2Ghz, dual core desktop that has only 4GB of RAM. Scanner

Hunter can achieve this efficiency because the preprocessing step has already filtered out information unlikely to be related to scanners.

We assume 40X response codes from websites in case of failure for the operation of Scanner Hunter. Even though the vast majority of websites respond with a `40X` code when a request fails for some reason or other, we are aware that there are rare cases where the websites do not follow the standard practice and would instead return other response code. In this case, they would return a customized web page containing a message that states reason why the request failed. Scanner Hunter can still work as expected as long as there is proper documentations for the inconsistency in response codes. Scanner Hunter then would take into account this information when it preprocesses the log files. Furthermore, Scanner Hunter's community-based approach would still work even without standard responses from the websites in the rare cases. Because most other websites would still respond with `40X` codes, the failed requests to these websites would be visible in the log files and Scanner Hunter would still be able to detect the community structure of the scanners.

4.4 Profiling scanners in the wild

In this section, we present the results of the extensive study we conducted on the behaviors of HTTP scanners in the wild. Some of the highlights from our study are as follows: 1) scanners are positioned diversely across IP space and number of unique /24 IP prefixes are nearly as high as the number of scanner IPs; 2) more than 90% of the returning scanners look for new resources and at least half of the Re-Paths they requested is new;

3) they spent little attention on disguising themselves to avoid detection in terms of User-Agent and referrer fields in the requests; 4) there are four categories of vulnerabilities that the scanners in our dataset are interested in.

4.4.1 Spatiotemporal properties of scanners

One of the more obvious questions that comes to mind is whether a network administrator can prevent scanning activities by blocking HTTP requests from certain IP prefixes. The answer is that such a solution will ***not*** be very practical or efficient because of two conclusions that we made once we have studied more closely the spatial and temporal properties of scanners: 1) the filter set will have to be large enough to cover all of the diverse IP prefixes 2) the filters would have short life-span and will be of limited use because most (80%) HTTP scanners, as we will show, are seen only ***once*** and never come back.

The rationale behind conclusion (1) is demonstrated by Figure 4.11, in which we show the percentage of distinct prefixes that remain each time we remove the least octet from the IP address. We can see that, for example, the number of distinct /16 IP prefixes accounts for at least 60% of the total number of original IPs. Furthermore, because HTTP scanners are as diversely placed across the IP space as users. It follows that an extremely large and fine-grained filter set would be needed to block the scanners.

The rationale for conclusion (2) can be found in Figure 4.12, the result of the experiment in which we kept track of each scanner labeled by Scanner Hunter and count how many daily visits they paid to the monitored websites during one week and during the entire six-month period. Interesting enough, no more than 20% of the scanners return to the

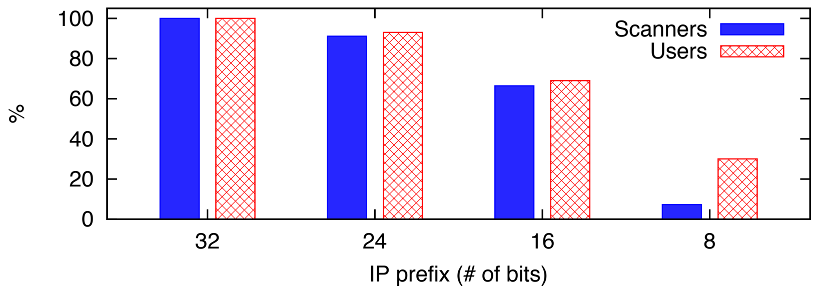


Figure 4.11: The number of IP prefix of HTTP scanner IPs and web user IPs

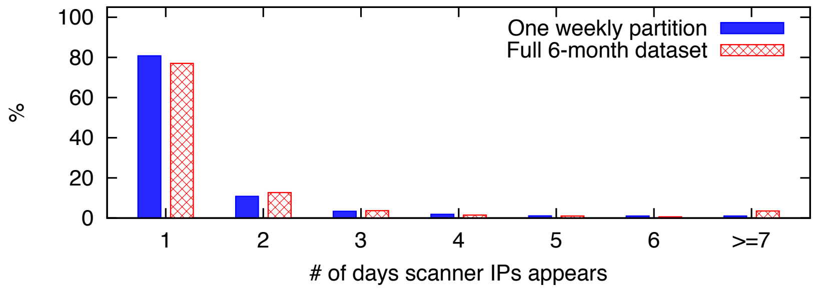


Figure 4.12: The total different days of scanner IPs show up in the monitoring intervals

websites hosted by the University campus in either periods. In fact, less than ***5%*** of them came back for a third time. This clearly indicates that no IP prefix-based, static blacklists can keep up with the scanning activities.

What we would like to know then, given that some (20%) scanners do return to the same websites, is whether scanners look for the same resources or different ones during subsequent visits. To find out, we performed an experiment on the entire six-month-long dataset as follows. Let the set S be scanner IPs that visited the monitored websites more than twice on different days. For each $s \in S$ we calculated the Average Ratio of new resources

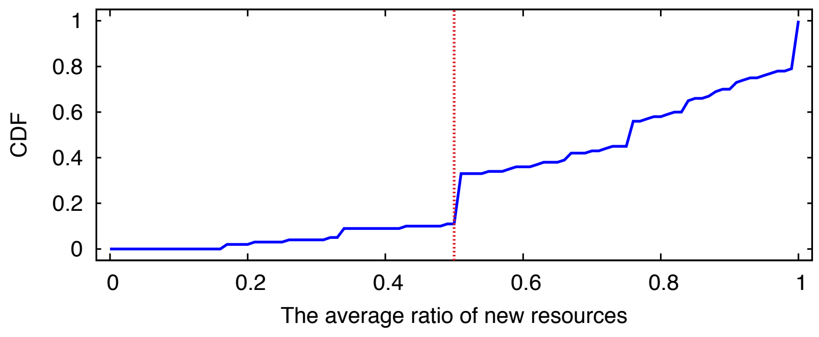


Figure 4.13: The ratio of new resources scanners look for when they come back

that s looks for using the following formula:

$$AR(s) = \frac{1}{n-1} \sum_{i=2}^{n} \frac{|W_i \setminus \bigcup_{j=1}^{i-1} W_j|}{|W_i|}$$

where

- n is the total number of daily visits that s paid to the monitored websites.
- W_i is the ***set*** of Re-Paths sought by s on the i^{th} visit in the dataset.
- $\bigcup_{j=1}^{i-1} W_j$ represents the accumulative set of every resource that s has requested up to the $(i-1)^{th}$ visit.
- The $(i-1)^{th}$ and i^{th} visits do not have to be on consecutive days.

What we discovered from the experiment is shown in Figure 4.13: for more than 90% of the returning scanners, half of what they look for each time is new. This indicates that the scanners that do come back tend to probe for resources that they did not request before.

User-Agent	**# of requests**	**# of scanners**
mozilla/4.0	31.2%	860
mozilla/3.0 (compatible; indy library)	14.5%	527
mozilla/4.0 (compatible; msie 6.0; windows nt 5.1; sv1)	6.0%	276
mozilla/4.0 (compatible; msie 6.0; windows nt 5.1; sv1;)	3.8%	429
<empty>	2.5%	132
mozilla/5.0 (windows; u; windows nt 5.1; en-us; rv:1.9.2) gecko/20100115 firefox/3.6	0.8%	73
mozilla/4.0 (compatible; msie 6.0; windows nt 5.1)	0.8%	55
mozilla/5.0 (compatible; msie 9.0; windows nt 6.1; trident/5.0)	0.8%	40
ineturl:/1.0	0.5%	104
mozilla/4.0 (compatible; msie 6.0; windows nt 5.1; sv1; .net clr 2.0.50727)	0.5%	65
	61.4%	2561

Table 4.2: The 10 most popular User-Agents in our dataset.

4.4.2 The tools used for HTTP scanning

Not only are we interested in identifying HTTP scanners, we also would like to know about the existing tools that allow them to carry out the act. As it turns out, the values of the User-Agent fields in the HTTP requests can give us an insight into the types of applications the scanners use. To this end, we show Table 4.2, where we listed the top 10 most popular User-Agents used by the scanners identified by Scanner Hunter. Although most of them are simpler than User-Agent strings produced by major browsers, there are three of them that deserve the most scrutiny:

- `mozilla/4.0`: Even though the User-Agent string mentions mozilla, the requests with this User-Agent string were not generated by the popular Firefox browser.
- `<empty>`: This simply means that the User-Agent field is empty. By default, none of the major web browsers generate requests without a value for the User-Agent field.
- `ineturl:/1.0`: This User-Agent string can be found by a number of HTTP-based applications that use a specific library. In this case, it may be possible that a number

of malicious entities used the library to implement their scanning tools.

At the same time, we were also interested in the distribution of number of User-Agents per scanning IP's address, as we would like to know what how many applications a scanner would typically use. From Figure 4.14, we can clearly see that each IP in the 90% of the IPs labeled as scanners by Scanner Hunter produced requests with exactly one User-Agent string, indicating that they used only one application for their purpose.

There are IPs associated with more than one User-Agent string, as seen in the same Figure. When we manually looked into the HTTP requests produced by those IPs, we observed that a) different User-Agents showed up at different times during the day and b) different User-Agents requested different Re-Paths. Therefore we believed that most of them were different scanners that happened to fall under the same IPs due to the effect of Network Address Translation (NAT).

Interestingly, however, there is a reason for us to believe that there were also scanners that used more than one application at a time. For example, there are four IPs located inside the Autonomous System number `50543` that are suspicious because:

- Each of the four IPs appeared on ***multiple*** blacklists [2].
- Each IP is associated the exact same two User-Agent strings.
- When we looked more deeply into their HTTP requests, we see that their behaviors are very similar. For example, suspicious IP s_1 sent two set of HTTP requests, each with a different User-Agent string. Moreover, the requests are ***close in time*** and the two User-Agent strings almost always appeared in a specific order.

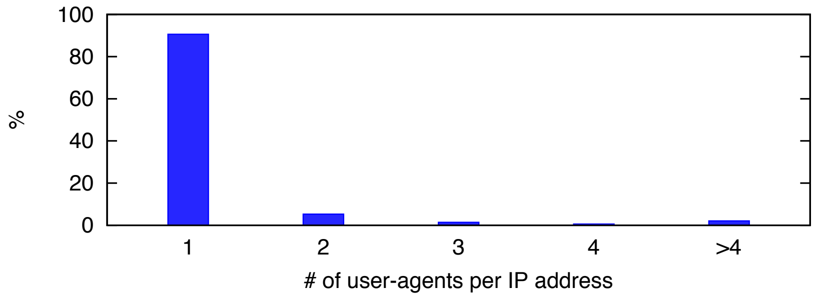


Figure 4.14: The number of User-Agent per scanner IP

Given the above facts, we came to the conclusion that there is a strong likelihood each of those IPs used two applications for malicious purposes: one application to probe for the existence of vulnerable resources and the other to analyze the resources that the first application actually discovered. Even more interesting is the fact that the same two User-Agents could be seen with each of the 4 IPs: `xpymep.exe` and `mozilla /4.0 (compatible; msie 6.0; windows 98; win 9x 4.90)`. According to [4], `xpymep.exe` is the name of a binary that is considered unsafe to download.

One natural question that can be asked is whether a network administrator can block scanning activities using the values of the User-Agent fields in the HTTP requests. The answer is no because of two reasons: 1) the False Positive rate would be high and 2) even though such a solution may mitigate scanning activities in the short term, it would not be effective in the long run due to how trivial it is to change the value of the User-Agent field. After all, scanners leave their User-Agent strings as-is possibly because of the little attention the problem has received. Once the scanners realize that the User-Agent strings may be a liability, we believe they would spend more effort on disguising the User-Agent more carefully.

4.4.3 The complacency of scanners

We have seen from the previous section that scanners do not spend too much time disguising their User-Agents, neither do they try to obfuscate the values of their referrer fields.

Figure 4.15 shows a big difference in terms of referrers between the requests generated by users and scanners: 90% of the times, the referrer field of a scanner's request is empty versus 10% for a user.

Recall that a request usually has nothing for their referrer field when a user types the URL directly into the URL bar of their browser or clicks on a bookmark. However, there are many embedded objects on a web page, so the referrer field in every request to an embedded object will be the URL of the original page. This is the reason why only a small percentage of legitimate requests have empty referrer fields.

This is not the case with scanners because a) the pages they requested do not exist most of the times and therefore there were no subsequent requests for embedded objects b) even if the pages existed, the scanners tend to only request the pages without downloading the embedded objects. Also noteworthy is the fact that in about 5% of the HTTP requests generated by scanners, the referrer field and the URL field are identical.

One surprise that came up during our study is that a small percentage of requests by scanners contain the names of popular search engines in their referrer fields as seen in Figure 4.15. When we inspected these referrer URLs more closely, we realized that they all contained ***search operators*** [6] like `allinurl` or `allinfile`. The search operators, when coupled with specific keywords that give a clue as to what vulnerabilities the scanners are

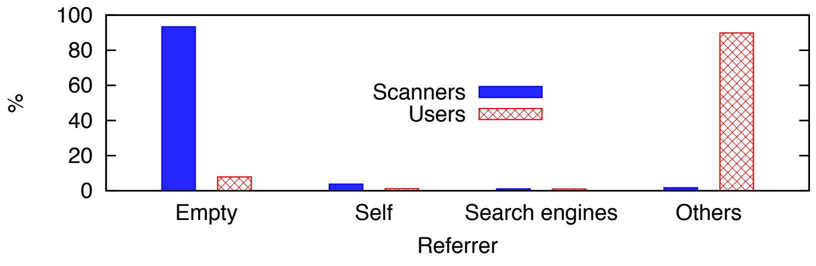


Figure 4.15: The popular referrers of HTTP requests sent by scanners

looking for, form query filters (for example `allinurl:admin/wp-login.php`) that can be used to fine-tune the results returned by search engines. The inquisitive readers can visit [1], a database that documents popular query filters used by hackers to find vulnerable resources.

We believe the search engines appeared in the referrers of scanners' requests because they took advantage of popular search engines to find vulnerable resources more efficiently. Since the search engines already indexed web pages on the Internet, scanners can use the search operators to 1) quickly and accurately locate vulnerable resources and 2) avoid the brute-force scanning of websites, which may take too much time and leave the evidence of their scanning in the logs.

4.4.4 What scanners look for

In this section, we present our findings regarding the resources that a scanner may be most interested in because armed with this knowledge, network administrators can gain a better understanding of the scanners' motivations and better secure their networks. For this purpose, we collected all the Re-Paths requested by the IPs labeled as scanners by Scanner

Group	# IPs	# Re-Paths	File types	URL patterns	Description
1	913	337	`asp`(58.3%) `php` (36.7%)	reg.asp, user_reg.aspx bbs/reg.php, login.asp	User registration and login pages
2	664	1143	`asp` (91.2%)	*member/index_do.php	Unclear
3	569	11273	`rar/zip/7z` (94.7%)	*wwwroot.zip	Backup files
4	433	9098	`php` (87.1%)	{include | data | plus}/*.php	Unclear
5	203	4836	`asp` (99.8%)	wp-{admin | login | comments | trackback}*.php	`wordpress`-related vulnerabilities
6	126	689	`php` (94.7%)	wp-content/*/thumb.php wp-content/*/timthumb.php	Vulnerabilities that may allow remote code execution [8]
7	112	401	`php` (100.0%)	*/demo/index.php	Demo versions of web services, which may be more vulnerable than actual services
8	88	638	`asp` (88.3%)	*admin*	Admin-related pages
9	59	123	`asp/aspx` (87.2%)	*/{fckeditor | ewebeditor | htmledit}/*	Some websites allow the change of their contents through certain editor interfaces, and it is these interfaces that the scanners searched for
10	58	139	`php` (90.6%)	*{phpmyadmin | sql}*	Control panels for backend databases

Table 4.3: The top 10 largest group of HTTP scanners and their targets

Hunter for the entire ***6-month*** duration of our full dataset, extract their file types, and ranked them according to their popularity in Figure 4.16.

We can see that ***three*** types of files dominate the list and account for 97% of the requests: `php`, `asp/aspx`, and `rar/zip/7z`. The most probable explanation would be that `php` and `asp/aspx` scripts are the least secured with respect to access permissions or more likely to contain exploitable vulnerabilities. The `rar/zip/7z` are compressed archives that, in the context of web server management, tend to be files used for backup or storing outdated data and, if left unsecured, would provide information that could enable future exploitations.

Because knowing the popular file types scanners look for is not enough to figure out their ***purposes*** for doing the scanning in the first place, we proceed to look more closely into the rest of the Re-Paths they requested in our data set. To achieve this goal, we merge

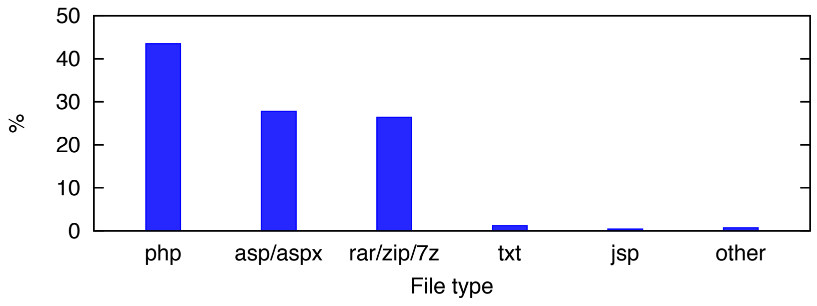


Figure 4.16: The popular file types of resource requested by HTTP scanners

scanner communities into categories based on the similarity in their request Re-Paths. More precisely, we perform the following actions on each of the communities labeled as scanners by Scanner Hunter:

1. Extract all Re-Paths from the community and remove all non-alphanumeric characters from each of the Re-Paths.
2. Convert each processed Re-Path into a set of ***trigrams***. A trigram in this case is any sequence of three consecutive characters. Once the trigrams have been extracted, the community is represented by the set of unique trigrams.

Given the set of communities $C = \{c_i\}$, $1 \leq i \leq |C|$, let t_i be the trigram set representation for each c_i. We then compute the Jaccard similarity measurement between every possible pair: $J(c_i, c_j) = \frac{|t_i \cap t_j|}{|t_i \cup t_j|}$. We then:

1. Pick the pair of communities with the largest similarity value and, if this similarity is more than a threshold T_J, we merge them to form a new community.
2. Calculate the similarity measurement between the new community with all old ones.

3. Repeat the above two steps until the largest similarity between any two communities is less than T_J.

We experimented with different values for the Jaccard similarity threshold T_J and found out $T_J = 0.3$ works reasonably well. Table 4.3 shows the top 10 biggest groups (out of 42) that remain once the merging concludes. The first column contains the ranks of the groups according to the number of unique IPs, the second the number of scanning IPs in each group, the third the number of unique Re-Paths found in each group, the fourth the most dominant file types as well as the percentage of the Re-Paths with these file types, the fifth a brief description for the observed patterns in each group's Re-Paths, and the final column some explanations regarding the vulnerabilities the scanners were aiming for.

Note that there may be other grouping schemes that can be used at this stage, but we adopted this approach because it is simple and the groups it produced are cohesive in ways that we will soon provide an explanation for.

The following are the key take-aways from our close inspection of the final groups.

1. Individual groups are homogenous in terms of User-Agents and file types even though we only merged communities based on the trigrams in their Re-Paths. In fact, for each group, the most dominant User-Agent often accounts for at least 90% of the scanner IPs and the most popular file type appears in at least 87% of the Re-Paths except for Group number 1.

This observed homogeneity within groups shows our grouping scheme works well for the purpose of discovering popular categories of vulnerabilities the scanners are interested in.

2. There are four major categories of resources the scanners want to find on the websites hosted by the University campus. They are as follows.

(i) Public login and registration pages for regular users. Related group(s): 1.

(ii) Control panels exclusive to administrators. For example, web interfaces for website management and interactions with backend databases. Related group(s): 5, 8, 9, 10.

(iii) Vulnerable pages, which may allow intrusion through remote code execution, cross-site scripting, or SQL injection. Related group(s): 6.

(iv) Backup files that may contain sensitive information for exploits. Related group(s): 3.

These categories pose very high risk. If the scanners can successfully exploit them, they can seriously compromise the security of the websites. For example, the scanners in Group number 6 were primarily concerned with exploiting a known vulnerability in Wordpress [8] that would allow remote execution of malicious code.

There are some groups whose intents are not obvious to us, for example group numbers 2, 4, and 7, so we did not list them in any of the above categories.

3. Backup files are a major target of scanners. It is surprising how aggressive (11,273 Re-Paths for 569 IPs) they are in Group number 3 of the table in the way they tend to have more unique Re-Paths per IP than other groups.

4. Toward a systematic categorization of scanner intents. With the help of this grouping scheme, we are able to uncover the top 10 major vulnerability categories from the web traffic of the University campus. This represents another contribution of our work. Using Scanner Hunter and the grouping scheme, network administrators would be well-informed about the most popular vulnerabilities that a scanner would be looking for. In that case, the administrator would be able to secure the vulnerable resources if they existed in the network.

4.5 Related work

To the best of our knowledge, there exists no work that provides a solution to the specific problem of HTTP scanners. The most recent work in an related area is that of Jacob et. al. [55], who provided an approach to detect stealthy crawlers that, unlike scanners, perform no malicious acts but may try to collect information from websites whether or not the owners want them to. In fact, the work of Jacob et. al. is only one among many [110, 78] that focus on the detection of crawlers.

There exists a different type of scanners that are primarily concerned with performing ***network reconnaissance***: IP and port-level scanning activities meant to to discover IPs on remote networks who may be actively listening on ports that, because they are unsecured, would allow an attacker to seize control of the machines. Much work, too, have been done on this topic [72, 11, 107, 33] but it is out of the scope of this paper. A brief review on network reconnaissance techniques can be found in [15].

Even as there is a lack of focus on the detection of HTTP scanners in the research community, there is a body of work on developing applications for security professional to test the security of specific applications [31, 56, 58] and there have been some efforts invested in the field of detecting covert and malicious web traffic [24]. In the same vein, there is existing work in the literature that confirms [86] our observation that because the behaviors between users themselves are diverse, it is difficult to automatically look at an IP in isolation and decide whether the IP is engaging in scanning activities.

4.6 Conclusion

In this work, we identify and study the problem of HTTP scanners, which as we saw, is by far not trivial.

Our main contribution is Scanner Hunter, an effective method to detect HTTP scanners that achieves a 96.5% detection precision. The key novelty of Scanner Hunter the use of a graph-based approach that overcomes the *needle-in-the-haystack* problem by comparing the group behavior of scanners versus that of legitimate users. The precision of Scanner Hunter is more than double of that of baseline solutions, while detecting more scanners at the same time.

As our second contribution, we provide an extensive study of HTTP scanning over a six-month period to understand: (a) what they are looking for and (b) their spatial and temporal behaviors. It is clear that the problem is acute, with roughly 4,000 unique IPs scanning a medium-sized University network every week and 80% new IPs every week.

We provide an initial effort towards inferring the intention and targets of scanning in a systematic way. We identify four major categories based on the resources scanners look for: (i) user registration and login pages, (ii) website management interfaces, (iii) pages with potential vulnerabilities, and (iv) compressed web archives such as `wwwroot.zip`, which are the products of website backup activities and should not be publicly visible.

The ultimate goal of our work is to (a) raise the awareness to this problem and (b) provide an initial but promising detection technique to mitigate the risk posed by HTTP scanners.

Chapter 5

Conclusion

Internet traffic has been undergoing constant evolution as new applications and technologies emerge. In this dissertation, we introduce three novel network traffic analysis tools to help ISPs and network administrators better understand the traffic in their networks. To demonstrate their effectiveness, we reverse-engineer user behaviors from real-world network traffic traces and conduct extensive studies on user behavior patterns.

Our first main contribution is SubFlow, a conceptually different network traffic classification approach that leverages the power of a subspace clustering technique. The key novelty is that our approach can learn the intrinsic statistical fingerprint of each application's traffic in isolation and using application specific features, which deviates from universal features in typical classification approaches. Our approach solves the dilemma that each application is captured best by specific features, but if we use all these features together in a single feature space, the classification performance suffers due to "the curse of dimensionality". We show that SubFlow has a great deal of potential by applying it on five different network traffic traces. SubFlow performs very well with minimal manual intervention: it identifies traffic of an application with very high accuracy, on average higher than 95%, and can detect new applications successfully.

Second, network traffic has been increasingly dominated by web traffic and the HTTP protocol has become the most prevalent means for applications to provide their services. However, understanding web traffic only from HTTP headers is far from trivial given the complex and interconnected websites of today. To address this, we develop a systematic approach called ReSurf which can reconstruct user requests with more than 95% precision and 91% recall. We also showcase interesting results that one can obtain from raw network traffic using ReSurf: (1) web users are continuously being exposed to advertising and tracking services and that in our traces, 50-60% of user requests (think clicks) interact with tracking or advertising services; (2) The click-through stream of users accessing websites are "shallow" in that the majority of streams have a maximum length of two; and (3) The amount of time that mobile users spend on a website is 300% more than the users on a wireline connection.

Third, we identify and study the problem of HTTP scanners, which pose serious danger but have not received enough attention. Our main contribution is Scanner Hunter, an effective method to detect HTTP scanners that achieves a 96.5% detection precision. The key novelty of Scanner Hunter is the use of a graph-based approach that overcomes the *needle-in-the-haystack* problem by comparing the group behavior of scanners to that of legitimate users. We also study HTTP scanning over a six-month period to understand: (a) what they are looking for, (b) their spatial and temporal behaviors, and (c) a systematic way of inferring the scanners' intention and grouping them into four major categories based on the resources they requested: (i) user registration and login pages, (ii) website management interfaces, (iii) pages with potential vulnerabilities, and (iv) compressed web archives such as `wwwroot.zip`, which are the products of website backup activities and should not be publicly visible.

In conclusion, we believe that SubFlow, ReSurf and Scanner Hunter enable ISPs, network administrators and researchers to understand, manage and model Internet traffic and its users better.

Bibliography

[1] Google hacking database. `http://www.exploit-db.com/google-dorks/`.

[2] He bgp toolkit. `http://bgp.he.net/`.

[3] http://easylist.adblockplus.org/en/.

[4] http://f.virscan.org/xpymep.exe.html. `http://f.virscan.org/xpymep.exe.html`.

[5] http://www.alexa.com/topsites.

[6] Search operators. `http://www.googleguide.com/advanced_operators.html`.

[7] Visual networking index (vni). `http://www.cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html`.

[8] Wordpress timthumb plugin - remote code execution. `http://www.exploit-db.com/exploits/17602/`.

[9] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Web content cartography. In *ACM IMC*, 2011.

[10] E. Allen and D. Haskin. IP version 6 over PPP. RFC 2472, Internet Engineering Task Force, December 1998.

[11] William H Allen, Gerald A Marin, and Luis A Rivera. Automated detection of malicious reconnaissance to enhance network security. In *SoutheastCon, 2005. Proceedings. IEEE*, pages 450–454. IEEE, 2005.

[12] D. Antoniades, E.P. Markatos, and C. Dovrolis. One-click hosting services: a file-sharing hideout. In *ACM IMC*, 2009.

[13] I. Assent, R. Krieger, E. Muller, and T. Seidl. INSCY: Indexing subspace clusters with in-process-removal of redundancy. In *2008. ICDM'08*, pages 719–724. IEEE, 2009.

[14] P. Barford and et al. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review*, 1998.

[15] Richard J Barnett and Barry Irwin. Towards a taxonomy of network scanning techniques. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, pages 1–7. ACM, 2008.

[16] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *ACM IMC*, 2009.

[17] Fabrício Benevenuto. Characterizing User Behavior in Online Social Networks Categories and Subject Descriptors. *Transition*.

[18] I. Bermudez, M. Mellia, M.M. Munafò, R. Keralapura, and A. Nucci. Dns to the rescue: Discerning content and services in a tangled web. In *ACM IMC*, 2012.

[19] L. Bernaille, R. Teixeira, and K. Salamatian. Early application identification. In *2006 ACM CoNEXT conference*. ACM, 2006.

[20] Laurent Bernaille, Renata Teixeira, and Kave Salamatian. Early Application Identification. In *ACM CoNEXT*, 2006.

[21] A. Bianco, G. Mardente, M. Mellia, M. Munafò, and L. Muscariello. Web user-session inference by means of clustering techniques. *IEEE/ACM Transactions on Networking*, 2009.

[22] A. Blum. Learning boolean functions in an infinite attribute space. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 64–72. ACM, 1990.

[23] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli. Revealing skype traffic: when randomness plays with you. *ACM SIGCOMM CCR*, 37(4):37–48, 2007.

[24] Kevin Borders and Atul Prakash. Web tap: detecting covert web traffic. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 110–120. ACM, 2004.

[25] M. Butkiewicz and et al. Understanding website complexity: Measurements, metrics, and implications. In *ACM IMC*, 2011.

[26] S.E. Coull, M.P. Collins, C.V. Wright, F. Monrose, M.K. Reiter, et al. On web browsing privacy in anonymized netflows. In *Proceedings of the 16th USENIX Security Symposium*, pages 339–352, 2007.

[27] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM CCR*, 37(1):5, January 2007.

[28] Inderjit S Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–274. ACM, 2001.

[29] Marios D Dikaiakos, Athena Stassopoulou, and Loizos Papageorgiou. An investigation of web crawler behavior: characterization and metrics. *Computer Communications*, 28(8):880–897, 2005.

[30] Derek Doran and Swapna S Gokhale. Discovering new trends in web robot traffic through functional classification. In *Network Computing and Applications, 2008. NCA'08. Seventh IEEE International Symposium on*, pages 275–278. IEEE, 2008.

[31] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: a state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 26–26. USENIX Association, 2012.

[32] Paul Ducklin. Dhs website falls victim to hacktivist intrusion. `http://nakedsecurity.sophos.com/2013/01/07/dhs-website-falls-victim-to-hacktivist-intrusion/`, 2013.

[33] Wassim El-Hajj, Fadi Aloul, Zouheir Trabelsi, and Nazar Zaki. On detecting port scanning using fuzzy based intrusion detection system. In *Wireless Communications and Mobile Computing Conference, 2008. IWCMC'08. International*, pages 105–110. IEEE, 2008.

[34] C. Elkan and K. Noto. Learning classifiers from only positive and unlabeled data. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 213–220. ACM, 2008.

[35] T. En-Najjary, G. Urvoy-Keller, M. Pietrzyk, and J.L. Costeux. Application-based feature selection for internet traffic classification. In *22nd International Teletraffic Congress (ITC)*. IEEE, 2010.

[36] T. En-Najjary, G. Urvoy-Keller, M. Pietrzyk, and J.L. Costeux. Application-based feature selection for Internet traffic classification. In *Teletraffic Congress (ITC)*, pages 1–8. IEEE, 2010.

[37] J. Erman, A. Mahanti, and M. Arlitt. Internet traffic identification using machine learning. In *IEEE GlobeCom*. Citeseer, 2006.

[38] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(9-12):1194–1213, 2007.

[39] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Semi-supervised network traffic classification. In *ACM SIGMETRICS*. ACM, 2007.

[40] Jeffrey Erman, Martin Arlitt, and Anirban Mahanti. Traffic Classification Using Clustering Algorithms. In *ACM SIGCOMM MineNet*, 2006.

[41] Jeffrey Erman, Alexandre Gerber, Mohammad T. Hajiaghayi, Dan Pei, and Oliver Spatscheck. Network-aware forward caching. *WWW*, page 291, 2009.

[42] Jeffrey Erman, Alexandre Gerber, and Subhabrata Sen. HTTP in the home: it is not just about PCs. In *ACM SIGCOMM Computer Communication Review*, 2011.

[43] Jeffrey Erman, Anirban Mahanti, and Martin Arlitt. Internet traffic identification using machine learning. In *IEEE GlobeCom*, 2006.

[44] Jeffrey Erman, Anirban Mahanti, Martin Arlitt, and Carey Williamson. Identifying and Discriminating Between Web and Peer-to-peer Traffic in the Network Core. In *WWW*, 2007.

[45] M. Ester, H.P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data mining*, volume 1996, pages 226–231. Portland: AAAI Press, 1996.

[46] Martin Ester, Hans-peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *ACM SIGKDD*, 1996.

[47] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[48] Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. ACAS: automated construction of application signatures. In *ACM SIGCOMM MineNet*, 2005.

[49] Kathryn Hempstalk and Eibe Frank. Discriminating against new classes: One-class versus multi-class classification. In *Advances in Artificial Intelligence*, pages 325–336. Springer, 2008.

[50] Kathryn Hempstalk, Eibe Frank, and I. Witten. One-class classification by combining density and class probability estimation. In *Machine Learning and Knowledge Discovery in Databases*, pages 505–519. Springer, 2008.

[51] A. Hintz. Fingerprinting websites using traffic analysis. In *International conference on Privacy enhancing technologies*, 2002.

[52] Cheng Huang, A Wang, J Li, and KW Ross. Measuring and evaluating large-scale CDNs. In *ACM IMC*, 2008.

[53] S. Ihm and et al. Towards understanding modern web traffic. In *ACM IMC*, 2011.

[54] Marios Iliofotou, Brian Gallagher, T. Eliassi-Rad, G. Xie, and M. Faloutsos. Profiling-By-Association: a resilient traffic profiling solution for the Internet backbone. In *ACM CoNEXT*, 2010.

[55] Gregoire Jacob, Engin Kirda, Christopher Kruegel, and Giov anni Vigna. Pubcrawl: protecting users and businesses from crawlers. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 25–25. USENIX Association, 2012.

[56] Torben Jensen, Heine Pedersen, Mads Chr Olesen, and René Rydhof Hansen. Thaps: automated vulnerability scanning of php applications. In *Secure IT Systems*, pages 31–46. Springer, 2012.

[57] K. Kailing, H.P. Kriegel, and P. Kröger. Density-connected subspace clustering for high-dimensional data. In *SDM*, 2004.

[58] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256. ACM, 2006.

[59] N. Kammenhuber, J. Luxenburger, A. Feldmann, and G. Weikum. Web search clickstreams. In *ACM IMC*, 2006.

[60] Nils Kammenhuber, Julia Luxenburger, Anja Feldmann, and Gerhard Weikum. Web search clickstreams. In *ACM IMC*, New York, New York, USA, 2006. ACM Press.

[61] T Karagiannis, A Broido, M Faloutsos, and Kc claffy. Transport Layer Identification of P2P Traffic. In *ACM IMC*, 2004.

[62] T. Karagiannis and et al. BLINC: multilevel traffic classification in the dark. In *ACM SIGCOMM*, 2005.

[63] T Karagiannis, K Papagiannaki, and M Faloutsos. BLINC: Multi-level Traffic Classification in the Dark. In *ACM SIGCOMM*, 2005.

[64] Ram Keralapura, Antonio Nucci, Zhi-Li Zhang, and Lixin Gao. Profiling users in a 3g network using hourglass co-clustering. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, pages 341–352. ACM, 2010.

[65] Hyunchul Kim, Kimberly Claffy, Marina Fomenkov, Dhiman Barman, Michalis Faloutsos, and KiYoung Lee. Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices. In *ACM CoNEXT*, 2008.

[66] R. Kohavi and G.H. John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1-2):273–324, 1997.

[67] H.P. Kriegel, P. Kroger, M. Renz, and S. Wurst. A generic framework for efficient subspace clustering of high-dimensional data. In *Fifth IEEE International Conference on Data Mining,*. IEEE, 2005.

[68] B. Krishnamurthy and C. Wills. Privacy diffusion on the web: A longitudinal perspective. In *ACM WWW*, 2009.

[69] B. Krishnamurthy and C.E. Wills. Generating a privacy footprint on the internet. In *ACM IMC*, 2006.

[70] Craig Labovitz and et al. Internet Inter-Domain Traffic. In *ACM SIGCOMM*, 2010.

[71] A. Le, A. Markopoulou, and M. Faloutsos. Phishdef: Url names say it all. In *IEEE INFOCOM*, 2011.

[72] Cynthia Bailey Lee, Chris Roedel, and Elena Silenok. Detection and characterization of port scan attacks. *Univeristy of California, Department of Computer Science and Engineering*, 2003.

[73] Wei Li, AW Moore, and Marco Canini. Classifying HTTP traffic in the new age. In *ACM SIGCOMM Poster*, 2008.

[74] X. Li and B. Liu. Learning to classify texts using positive and unlabeled data. In *International joint Conference on Artificial Intelligence*, volume 18, pages 587–594. Citeseer, 2003.

[75] Yeon-sup Lim, Hyun-chul Kim, J. Jeong, C. Kim, T.T. Kwon, and Y. Choi. Internet traffic classification demystified: on the sources of the discriminative power. In *ACM CoNEXT*, 2010.

[76] Christoph Lindemann and Lars Littig. Coarse-grained Classification of Web Sites by Their Structural Properties. In *WIDM*, pages 35–42, 2006.

[77] Christoph Lindemann and Lars Littig. Classifying Web Sites. In *WWW Poster*, pages 1143–1144, 2007.

[78] Anália Lourenço and Orlando Belo. Applying clickstream data mining to real-time web crawler detection and containment using clicktips platform. In *Advances in Data Analysis*, pages 351–358. Springer, 2007.

[79] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.* Insecure, 2009.

[80] Justin Ma, Kirill Levchenko, Christian Kreibich, Stefan Savage, and Geoffrey M Voelker. Unexpected Means of Protocol Inference. In *ACM IMC*, 2006.

[81] B.A. Mah. An empirical model of http network traffic. In *IEEE INFOCOM*, 1997.

[82] Gregor Maier, Anja Feldmann, Vern Paxson, and Mark Allman. On dominant characteristics of residential broadband internet traffic. *ACM IMC*, page 90, 2009.

[83] D. Martin, H. Wu, and A. Alsaid. Hidden surveillance by web sites: Web bugs in contemporary use. *Communications of the ACM*, 46(12):258–264, 2003.

[84] A. McGregor, M. Hall, P. Lorier, and J. Brunskill. Flow clustering using machine learning techniques. *PAM*, 2004.

[85] Anthony McGregor, Mark Hall, Perry Lorier, and James Brunskill. Flow Clustering Using Machine Learning Techniques. In *PAM*, 2004.

[86] Mark Meiss, Filippo Menczer, and Alessandro Vespignani. On the lack of typical behavior in the global web traffic network. In *Proceedings of the 14th international conference on World Wide Web*, pages 510–518. ACM, 2005.

[87] Andrew Moore and Konstantina Papagiannaki. Toward the Accurate Identification of Network Applications. In *PAM*, 2005.

[88] Andrew Moore and Denis Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *ACM SIGMETRICS*, 2005.

[89] A.W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):50–60, 2005.

[90] Tatsuya Mori, Ryoichi Kawahara, Haruhisa Hasegawa, and Shinsuke Shimogawa. Characterizing traffic flows originating from large-scale video sharing services. In *TMA*, 2010.

[91] E. Muller, S. Günnemann, I. Assent, and T. Seidl. Evaluating clustering in subspace projections of high dimensional data. *Proceedings of the VLDB Endowment*, 2(1):1270–1281, 2009.

[92] Thuy T T Nguyen and Grenville Armitage. A Survey of Techniques for Internet Traffic Classification using Machine Learning. *IEEE Communications Surveys and Tutorials*, 4th edition, March 2008.

[93] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *ACM SIGKDD Explorations Newsletter*, 6(1):90–105, 2004.

[94] M. Pietrzyk, J.L. Costeux, G. Urvoy-Keller, and T. En-Najjary. Challenging statistical classification for operational usage: the adsl case. In *Proceedings of the 9th ACM IMC*, pages 122–135. ACM, 2009.

[95] L. Popa, A. Ghodsi, and I. Stoica. Http as the narrow waist of the future internet. In *ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[96] Xiaoguang Qi and Brian D. Davison. Web page classification. *ACM Computing Surveys*, 41(2):1–31, February 2009.

[97] MA Rajab, Fabian Monrose, Andreas Terzis, and N. Peeking Through The Cloud: DNS-based estimation and its applications. *Proceedings of the 6th international conference on Applied cryptography and network security*, 2008.

[98] S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, NJ, 1995.

[99] Bernhard Sch and Robert Williamson. Support Vector Method for Novelty Detection. In *Pattern Recognition*, volume 12, pages 582–588, 2000.

[100] Dominik Schatzmann, Wolfgang Mühlbauer, Thrasyvoulos Spyropoulos, and Xenofontas Dimitropoulos. Digging into HTTPS : Flow-Based Classification of Webmail Traffic. In *IMC*. ACM, 2010.

[101] F. Schneider, B. Ager, G. Maier, A. Feldmann, and S. Uhlig. Pitfalls in HTTP traffic measurements and analysis. In *International Conference on Passive and Active Measurement (PAM)*, 2012.

[102] F. Schneider, A. Feldmann, B. Krishnamurthy, and W. Willinger. Understanding online social network usage from a network perspective. In *ACM IMC*, 2009.

[103] Fabian Schneider, Sachin Agarwal, Tansu Alpcan, and Anja Feldmann. The New Web: Characterizing AJAX Traffic. In *PAM*, pages 31–40, 2008.

[104] Fabian Schneider, Anja Feldmann, Balachander Krishnamurthy, and Walter Willinger. Understanding online social network usage from a network perspective. In *IMC*, New York, New York, USA, 2009. ACM Press.

[105] D. Senie and A. Sullivan. Considerations for the use of dns reverse mapping. 2008.

[106] Inc.) Senie, D. (Amarathan Networks and Inc.) A. Sullivan (Command Prompt. Considerations for the use of DNS Reverse Mapping draft-ietf-dnsop-reverse-mapping-considerations-06, 2008.

[107] Siraj A Shaikh, Howard Chivers, Philip Nobles, John A Clark, and Hao Chen. Network reconnaissance. *Network Security*, 2008(11):12–16, 2008.

[108] F.D. Smith and et al. What TCP/IP protocol headers can tell us about the web. In *ACM SIGMETRICS Performance Evaluation Review*, 2001.

[109] Eduardo J Spinosa and De Leon F De Carvalho. Learning novel concepts : beyond one-class classification with OLINDDA. In *PKDD*, 2007.

[110] Athena Stassopoulou and Marios D Dikaiakos. Crawler detection: A bayesian approach. In *Internet Surveillance and Protection, 2006. ICISP'06. International Conference on*, pages 16–16. IEEE, 2006.

[111] I. Trestian and et al. Unconstrained endpoint profiling (googling the internet). In *ACM SIGCOMM*, 2008.

[112] Ionut Trestian, Supranamaya Ranjan, Aleksandar Kuzmanovic, and Antonio Nucci. Unconstrained endpoint profiling (Googling the Internet). In *ACM SIGCOMM*, 2008.

[113] S. Turner and T. Polk. Prohibiting secure sockets layer (ssl) version 2.0. Rfc, IETF, March 2011.

[114] Yu Wang, Yang Xiang, and S.Z. Yu. An automatic application signature construction system for unknown traffic. *Concurrency and Computation: Practice and Experience*, 22(13):1927–1944, 2010.

[115] N. Williams, S. Zander, and G. Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. *ACM SIGCOMM CCR*, 36(5):5–16, 2006.

[116] Ian H Witten, Eibe Frank, Leonard E Trigg, Mark A Hall, Geoffrey Holmes, and Sally Jo Cunningham. Weka: Practical machine learning tools and techniques with java implementations. 1999.

[117] Charles Wright, Fabian Monrose, and G.M. Masson. HMM Profiles for Network Traffic Classification. In *VizSEC/DMSEC*, pages 9–15, 2004.

[118] Charles V Wright, Fabian Monrose, and Gerald M Masson. On Inferring Application Protocol Behaviors in Encrypted Network Traffic. *J. Mach. Learn. Res.*, 7:2745–2769, 2006.

[119] G. Xie and et al. ReSurf: Reconstructing Web-Surfing Activity From Network Traffice. `http://www.cs.ucr.edu/~xieg/pubs/ReSurf_TR.pdf`.

[120] K Xu, Z Zhang, and S Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applications. In *ACM SIGCOMM*, 2005.

[121] Q. Xu, J. Erman, A. Gerber, Z.M. Mao, J. Pang, and S. Venkataraman. Identifying diverse usage behaviors of smartphone apps. In *ACM IMC*, 2011.

[122] S. Zander, T. Nguyen, and G. Armitage. Automated traffic classification and application identification using machine learning. In *30th Anniversary. The IEEE Conference on Local Computer Networks*, pages 250–257. IEEE, 2005.

[123] Sebastian Zander, Thuy Nguyen, and Grenville Armitage. Automated Traffic Classification and Application Identification using Machine Learning. In *IEEE LCN*, 2005.