2012

CobWeb: A System for Automated In-Network Cobbling of Web Service Traffic

Hitesh Khandelwal
Purdue University, hkande@cs.purdue.edu

Fang Hao
Bell Labs Alcatel-Lucent

Sarit Mukherjee
Bell Labs Alcatel-Lucent

Ramana Rao Kompella
Purdue University, kompella@cs.purdue.edu

T.V. Lakshman
Bell Labs Alcatel-Lucent

Report Number:
12-005

CobWeb: A System for Automated In-Network Cobbling of Web Service Traffic

†Hitesh Khandelwal, ‡Fang Hao, ‡Sarit Mukherjee, †Ramana Kompella, ‡T.V. Lakshman
†Purdue University, ‡Bell Labs Alcatel-Lucent

ABSTRACT

We consider the problem of in-network categorization of all traffic associated with a given set of web services. While this problem can be viewed as a generalization of per-session traffic monitoring, a key difficulty is that we have to construct the entire *session tree* that represents the transitive closure of all traffic downloaded as a result of a user accessing a given web service. Such in-network session tree construction and monitoring is useful for many measurement, monitoring, and new types of billing services such as 'reverse billing' where usage charges are paid for by either the service provider or the ISP itself as an incentive to the user.

Automated construction of the session tree based on network traffic observation is challenging and to our knowledge unaddressed. The challenges arise due to the complexities inherent in today's web services and the lack of universal standards that are followed when designing web services. This necessitates the use of heuristics that rely upon prevalent web service design practices. In this paper, we present a system, called CobWeb, that performs this automated in-network cobbling and monitoring of web services traffic. We evaluate the classification accuracy of CobWeb by extensive experimentation using controlled downloads and by analysis of about 100 popular web sites using large traffic traces (over 700 GB) collected at a major university's gateway. Our experiments suggests that CobWeb can achieve good accuracy with low ($< 5\%$) false positive and negative rates.

1. INTRODUCTION

The design of network-based mechanisms for per-flow traffic measurement has been a topic of much research interest (*e.g.*, [6, 13, 11, 21, 18]). Here, the challenges have been in designing low-cost mechanisms for collecting statistics of millions of flows or sessions at speeds of 10 Gbps and beyond. Another topic of much interest has been in-network-based mechanisms for application identification (*e.g.*, [8]). Here the challenge is in the design of network-based mechanisms to identify the applications that are generating the observed traffic in the network. This identification is needed for per-application network usage reports, application-aware traffic management, service-level-agreement conformance, subscriber management and billing, etc.

In this paper, we focus on a new measurement problem, namely that of associating with a particular web service all the traffic that is generated upon each access to that web service. This entails identifying all the traffic that is due to downloads from the original host for the web service, its content delivery network (CDN), and downloads of embedded objects from third-party services (*e.g.*, advertisements). Unlike per-flow traffic measurement, for this more general problem, it is necessary to construct a "session tree" that represents the transitive closure of all web service accesses that happen as a consequence of accessing a given root web service. Note that the session tree may be quite dynamic and the internal nodes can change across users as well as across time. To our knowledge, this general measurement problem is largely unaddressed.

The motivations for this new class of measurements are largely similar to existing traffic measurement solutions. Today, per-flow traffic measurement and application identification are extensively used by ISPs to gain insight into their network operations. These are often done using several of the specialized monitoring boxes, currently in the market [3, 1], that provide extensive per-session, per-application or per-flow usage and performance reports. These boxes also identify traffic to various popular web-sites by time-of-day, region, etc. However, these solutions are at a coarse-granularity (*e.g.*, IP address, protocol signature); efficient solutions for traffic monitoring of more meaningful aggregates such as the web session trees that we consider in this paper can significantly add to the usefulness of existing monitoring and measurement equipment.

A more recent and emerging potential application is 'reverse billing' where any charges associated with accessing a web service are billed back to the web service provider rather than to users accessing the service. Reverse billing is motivated by the growing shift from flat-rate to tiered-pricing in wireless networks, and in some countries, wired networks as well. Examples are such plans such as those of AT&T which permit 250MB of data usage for about $15 per month and 2GB for $25 in the United States. Thus, web service providers may want to make it attractive to customers by to providing 'toll-free' access to their services. Another

alternate could be a subsidized service model where the ISP may provide access to a web-services such as ESPN, Facebook, or CNN for some nominal fee per month. For example, Vodafone already offers unlimited access to a few sites (*e.g.*, Facebook, Twitter, FourSquare and Myspace) in every new contract in Australia.

For such applications, it is important to construct the session tree in the network, which is challenging for many reasons. Widely used web services are a complex mashup of content from several supporting services including CDNs, third-party advertisement platforms (*e.g.*, ads.doubleclick.com), and other third-party services (*e.g.*, CNN web services using Facebook for friend recommendations). This makes the structure of the session tree difficult to infer. Also, with the inherent flexibility in designing web services, the session tree is rarely static. Moreover, web services are largely personalized—the content served varies with the user even for the same URI. Thus, one cannot use a unique set of URIs to identify a service. Yet another issue is that web services are usually hosted across many data centers causing IP addresses to change based on user location. Also, since a CDN such as Akamai's, can service many different web services the use of IP addresses itself is not sufficient.

In this paper, we describe a system, CobWeb, for this general measurement problem. It automatically performs in-network cobbling of different web services—we use the term "cobbling" for the identification and measurement of all traffic associated with a given web service. To the best of our knowledge, CobWeb is the first system developed for addressing this measurement problem. We assume that CobWeb has access to both upstream and downstream traffic flows, since CobWeb is a system that observes network traffic (hence deployed at the network edge with port mirroring used for access to traffic flows). We use online mechanisms for the cobbling done by CobWeb since we need to effectively handle web service personalization. Also, online mechanisms need less storage and raise fewer privacy concerns. However, they require more processing power, which is not a limiting factor using current multicore processors.

CobWeb works in two stages. It identifies any supporting CDN used by a web service and then, it identifies all the embedded objects downloaded for that web service. The total data usage consists of all the traffic that is associated with access to this service—be it from the original access, from the CDNs, and from all the related chain of accesses that are triggered by the embedded links. As pointed out earlier, designing a system that tracks all the traffic belonging to a web service with 100% accuracy is a big challenge given the lack of any uniform methodology or standards in the composition of web services. The mechanisms that we use necessarily rely on heuristics based on prevalent practices in the provision of web services.

We evaluate our system based on two web traffic traces in total amount of 739 GB collected from a large university campus network, along with traces that are generated at lab controlled environment for emulating user browsing behaviors. We take 70 web sites from the Alexa top 100 US sites, along with the top 26 popular web sites for users in a large university campus network as the target web services. Our results show that the system can achieve an average false positive rate of 3.5% and false negative rate of 4.8% across these 96 web services.

The rest of the paper is organized as follows: We first present the problem statement precisely and discuss naive solutions that do not work well. We discuss our approach in Section 3 followed by implementation details of our system in Section 4. In Section 5, we evaluate each classification heuristic individually, and then show the accuracy of the final combined classification algorithm.

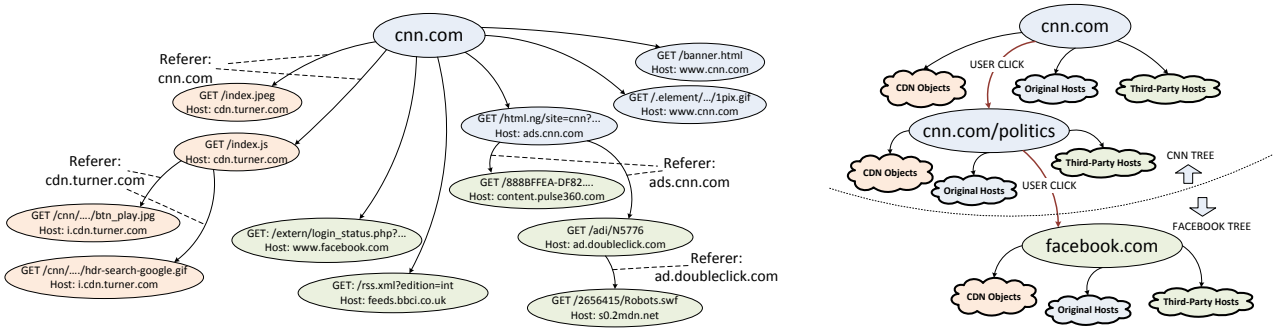
2. PROBLEM STATEMENT

In this section, we start with some preliminaries about web services. We then clearly define our main objective in this paper, argue why the problem is hard, and show that simple solutions do not work well.

2.1 Web Service Preliminaries

A web browser interacts with a web server by sending HTTP requests to the server, and then receiving response messages back. The most common requests are GET (for downloading content) and POST (for uploading content). The contents on a web page, displayed to a user, are usually downloaded via multiple GET/POST requests that are sent to one or more hosts. The page returned in the response to a request to cnn.com message may contain many links to other embedded objects as shown in Figure 1(a). For example, the objects /.element/..../1pix.gif, /banner.html are a are fetched immediately after the main CNN web page is downloaded. Embedded objects may in turn trigger the download of many more embedded objects. For example, as shown in Figure 1(a), the GET request to /index.js on host cdn.turner.com leads to the download of many further embedded objects such as /cnn/.../btn_play.jpg. For this session, more than 150 additional requests are sent to 24 other hosts to acquire all the additional content. Of course, such numbers may change due to dynamic nature of the content.

The hosts that provide content to a web page can be classified into three broad categories: *original hosts*, *CDN hosts* and *third party hosts*. Original hosts are those that belong to the same root domain of the web service, *e.g.*, cnn.com and money.cnn.com. CDN hosts are the servers that are part of the CDNs associated with that main domain (*e.g.*, cdn.turner.com is the main CDN associated with CNN). Third-party hosts provide content such as advertisements, statistics collection, social networking, and so on (*e.g.*, feeds.bbci.co.uk and www.facebook.com for CNN). These hosts may be contacted as a result of the main web page or because of embedded requests as shown in Figure 1(a). For example, we can see more third-party requests to hosts such as ad.doubleclick.com originating after the ads.cnn.com object is fetched.



(a) Constituents of a CNN session

(b) Defining CNN web session

Figure 1: Subfigure (a) shows a subset of URLs downloaded when a user downloads the base `cnn.com` page. Subfigure (b) shows that clicks to third party websites are not part of the the CNN session tree.

2.2 Objective

Our goal is to 'cobble' an entire *session tree* corresponding to a user accessing a given web service in the network (*e.g.*, at an ISP border router). We define a web session more precisely as follows:

- All the content downloaded from the original hosts (*e.g.*, *.cnn.com) responsible for the web service.
- All the content downloaded from CDN servers (*e.g.*, *.cdn.turner.com) the web service uses as part of the session.
- All the embedded objects automatically downloaded for the web service from any servers, *e.g.*, original hosts, CDN hosts or any third-party hosts.

Figure 1(b) shows an example navigation of the CNN web page. The root of the session tree starts at cnn.com that, as discussed in Section 2.1, involves the browser automatically fetching embedded content from the original hosts, CDN hosts or third party hosts. The user click on cnn.com/politics leads to another series of sessions to various hosts and this is again considered part of the session tree since the click leads to a CNN webpage. When a user clicks on a link to a third party website such as facebook.com, we consider it to be outside of the CNN session tree (as shown in the figure).

Though not shown in the example, any clicks to URLs which involve the CDN hosts or original hosts are considered as part of the session tree. For example, if the user clicks on URL cdn.turner.com/xyz.html (a contrived example), it would be considered a part of the session tree. While this example started with cnn.com, a user may directly enter the URL cnn.com/politics into the browser and make that URL the root. However, we do not consider session trees starting directly from a CDN URL as part of the web service since CDNs are known to be shared across different web services. For example, cdn.turner.com may host content from tbs.com.

2.3 Why is the problem hard ?

The *key difficulty* in cobbling of the web sessions in the network comes from the fact that routers only observe a stream of HTTP requests to various web services from a given client, but there is no obvious handle one can use to easily bind the requests that belong to a given session. Simple approaches such as enumerating domain names or IP addresses for their applications (*e.g.*, for reverse billing) do not work well as we shall discuss next. Anecdotal evidence suggests that ISPs today are already using these naive approaches for their applications, primarily because of the lack of a compelling alternative.

Naive Solution 1: Use Domain Names One simple solution to this problem is to use domain names that are associated with the web service. Thus, the router may essentially, for every web service of interest, simply keep the domain names that it needs to match. While such an approach may have worked 10-15 years ago when web services were very simple, *e.g.*, a few servers would serve static HTML content, unfortunately this approach will not work well for today's web services due to their complex composition. Specifically, many modern websites are constructed as a mash-up of many different services, often relying on third party websites as well. We illustrate this complexity in Figure 2, where we show the number of unique URLs and unique domain web pages for about 96 web sites, comprising 70 of Alexa's top 100 web pages and 26 most popular domains observed at a large university gateway . As we can observe from the figure, most of these web pages tend to access many different domains (up to 50) and the number of unique objects fetched is as high as 450.

Perhaps more importantly, many web services involve fetching objects from common domains. For example, the main cnn.com and nytimes.com web pages include an embedded request to facebook.com. Thus, the request to Facebook needs to be classified as part of CNN or NYTimes or even just Facebook depending on the overall context, making it difficult to come up with a blanket rule for all websites. We show this phenomenon in Figure 3, where we quantify the overlap among these 96 websites. Specifically, suppose a website A resulted in the set of URLs, we compute the frac-

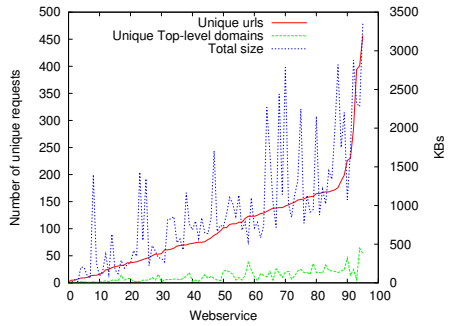


Figure 2: Statistics for sessions of the web services

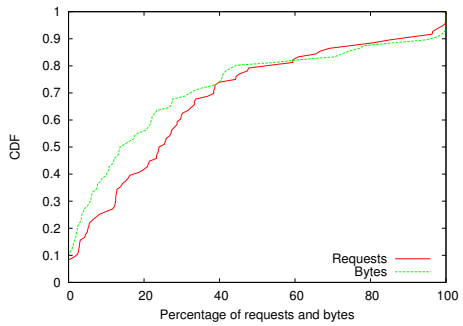


Figure 3: Quantifying domain overlaps across web services

tion of URLs (and their total bytes) in this set, that have the same domain name (*e.g.*, facebook.com) as that of at least one URL in URL set of B. We plot the maximum among all fractions for each of the 96 web sites. As we can see from the figure, the overlap can be quite high; for half of the 96 websites, 30% of requests and approximately 20% of bytes overlap with at least one other website.

Naive Solution 2: Use IP Addresses The next solution we consider is to enumerate IP addresses that correspond to a given web service. Similar to the domain names above, IP addresses cannot easily be used for isolating web services since the same web servers may be hosting content from different providers. For example, today many web services use content delivery networks (CDNs), and often they use the same CDN provider (*e.g.*, Akamai) for hosting content. Thus, the same server IP addresses may be shared across different web services making it difficult to filter out all requests specific to a given web site just by examining network-layer header fields alone. In addition, most CDNs adopt locality-based DNS resolution and so the exact IP address(es) used can change depending on the location.

Other ideas In certain settings, when we have cooperation from a given web service provider, we could potentially obtain a 'site map' of all the host names, and URIs used in constructing the web service. In general, however, we cannot assume that such cooperation is the norm, since the kind of application (*e.g.*, for reverse billing) may not necessarily involve the particular website content provider, and may depend only on an agreement between the customer and the ISP alone. Even if obtain the site map of domains for a web-service, this may not be sufficient because of the overlap between different services we have already discussed before. (For generality, in this paper, we assume no such cooperation.) We cannot also assume any client cooperation since it is too intrusive an approach to run a special agent on the client side machine.

Another possible idea is to collect and store all HTTP data corresponding to a user, and somehow reconstruct the web activity corresponding to that user accessing a given page. Unfortunately, modern websites heavily rely on CSS, javascript, etc.; parsing and interpreting these websites requires a full-fledged javascript engine on the router making it complicated to keep up with line rates.

Thus, it is clear that simple enumeration of either domain names or IP addresses will not work well for our problem. Instead, we need a more sophisticated way to derive the association between web accesses, but not as complex as parsing and interpreting javascript and other web pages in a detailed fashion. We discuss one such middle-ground approach that we propose next.

3. COBWEB DESIGN

In this section, we describe the design of CobWeb, a system for in-network cobbling of web service traffic. We first present an overview of our approach, and then describe the heuristics that form the basis for CobWeb. We assume that both directions of the traffic can be observed by CobWeb.

3.1 Overview

In our approach, we mainly leverage a key field within the HTTP headers, namely the 'Referer' field, which most browsers today set. This field mainly indicates the (previous) page that referred to the current page. For example, when a user clicks on www.cnn.com/politics URL on the www.cnn.com/US web page, the corresponding GET request will contain www.cnn.com/US as the Referer. Note that the Referer field contains both a referrer host (e.g., cnn.com) and referrer URI (e.g., /US) should it be present. We leverage the Referer field to keep track of the navigation chains to identify the roots of the session trees.

While using the Referer field, one can form an association between two different webpages A and B if A led to B, it is not always easy to establish whether B was a result of the user clicking on A, or an automated download. The reason why this is important is that automated downloads need to be counted as part of the actual web service, even if it is to third party domains, *i.e.*, non-origin domains. On the other hand, user clicks to third party domains, that start a new session tree (as we discussed in Figure 1(b)). However, sometimes a user can click on an associated CDN domains (*e.g.*, turner.com for CNN), which must be considered as part of the session tree. Thus, to make this differentiation, it is important to first establish the set of CDNs for a given domain, after which we need methods to differentiate between the

embedded downloads and clicks to third party sites. Note however, all accesses to the CDN cannot be considered as part of that particular web service, since the same CDN may host multiple web services.

Our overall approach therefore consists of the two basic steps: *CDN detection* and *Embedded object detection*. The CDN detection step is an offline process that involves identifying the CDN (or supporting) domains that play an important supporting role for delivering a given web service. We expect that for web services of interest, we separately track their associated CDNs (which rarely change) and incorporate them into the cobbling process. For embedded object detection, which is an online process, the goal is to identify the set of embedded objects fetched as part of a given web page as opposed to those that are retrieved due to user clicks. The key metric used here to distinguish between the two types of retrievals is that embedded object retrieval has much less 'think time' than user-click based retrieval since the embedded objects are automatically fetched by the web browser. We also use the fact that some embedded objects have standard file-types such as javascript (extension .js, .json) that are not usually associated with objects retrieved by user clicks.

Given the importance of the Referer field in our approach, one could argue that it is easy to disable the Referer field since many modern browsers provide the appropriate settings anyway. In most modern browsers, however, the Referer field is by default turned on; very few people even bother to turn it off (or are even savvy enough to turn it off). If indeed, the Referer field were turned off completely by a majority of the users, the whole multi-billion dollar Internet advertisement industry would crumble, since they heavily rely on the Referer field for tracking the source of their clicks. Thus, companies such as Google and Microsoft, have the incentive to keep the Referer field on in Chrome and Internet Explore browsers to support their online advertising businesses.

In the next few subsections, we first discuss each of the heuristics in detail and then present the overall algorithm.

3.2 CDN Detection

The goal of CDN detection is to identify supporting CDNs (if any) for a web service. In the example of Figure 1(a), the focus for this would be the left portion of the tree, i.e., requests with host *.cdn.turner.com that belong to the CDN for CNN. One method for CDN detection is to monitor web request traffic at a network edge router (*e.g.*, campus gateway or ISP border router), and use it to identify the domain that has delivered the most amount of traffic to clients when the pages are downloaded. The main purpose of a CDN is to make sure that static and relatively less frequently changing parts of the web pages (such as javascript objects and some images) are replicated and placed close to the clients. (Note that this is not to say that frequently changing parts are never part of CDNs, but generally CDNs store more static objects than dynamically changing ones.) Hence, it is reasonable to assume that most traffic for a given web site using the CDN will come from the CDN (apart from the main domain itself).

Implementing this idea is not straightforward since we still need to identify, from the monitored traffic, the total traffic to a web site (which is the cobbling problem that we started with). We first focus on the portion of the traffic that can be clearly identified—the HTTP GET/POST requests with referer URL belonging to the root domain. For example, to detect CDN for cnn.com, we first look at all requests where the request has its referer host as cnn.com or sub-domain of cnn.com such as money.cnn.com. This is the traffic for downloading the embedded pages of the root domain (*e.g.*, embedded content for main cnn.com web page) and the traffic for accessing pages when user navigates away from the page (*e.g.*, user clicks on nytimes.com on cnn.com page). Note that in the second case, only the first GET or POST request has referer as cnn.com. The rest of the requests have nytimes.com or subsequent pages as referer. As long as the traffic for accessing the external web sites, in the second case, does not exceed the traffic for accessing the CDN for cnn.com, it will not affect the result of CDN detection. Since users in general are likely to navigate to various different pages *within* the origin or CDN domains, the latter traffic should not be an issue in practice.

Another issue is that web services may use multiple CDNs. For example, cnn.com uses both Level 3 and Akamai CDNs. Fortunately, in many cases, we find that the CDN host contained in HTTP requests is an alias of the canonical name (CNAME) of the actual server. For example, cnn.com has i,z.cdn.turner.com for different types of content. i.cdn.turner.com is an alias for CNAME cdn.cnn.com.c.footprint.net and is owned by Level 3, z.cdn.turner.com is an alias for CNAME z.cdn.turner.com.edgesuite.net and is owned by Akamai. Use of this kind of aliasing is convenient for the web service provider since it allows web pages to be not tied to any particular CDN provider. A web site can switch to other CDNs by simply mapping the alias to other CNAMEs. Given the structure of the CDN aliases, we can detect the "CDN domain" (*e.g.*, cdn.turner.com) instead of specific CDN host (*e.g.*, i.cdn.turner.com). Our algorithm looks at different levels of the host domain, and tries to aggregate them. For example, level-1 (top-level) domain of i.cdn.turner.com is com, level-2 domain of that is turner.com, etc.

CDN Detection Algorithm We start with traffic monitored at the network edge router. The algorithm (shown in Figure 4) starts by looking for all the HTTP GET requests that contain the main host domain as the referer (*e.g.*, cnn.com or ads.cnn.com or money.cnn.com for CNN). Let this total number be n. We also compute the break down of these requests individually to each and every host h (denoted by n_h). For example, if x and y GET requests with referer as cnn.com were made to disqus.com and cdn.turner.com, respectively, we denote n_{disqus}= x and $n_{cdn.turner}$=y. Note that all counts are in number of bytes. Out of all hosts h, we pick the host

req:	HTTP GET or POST request
$host(req)$:	host name in request req
$rhost(req)$:	referer host name in request req
$root$:	root domain
l:	host domain level
n:	number of bytes for all sessions s.t. $rhost(req) = root$
$dom_l(h)$:	level-l domain of host h
$n_{h,l}$:	number of bytes for all sessions s.t. $rhost(req) = root$ and $dom_l(host(req)) = dom_l(h)$
b_i:	number of bytes for session i

for each session i s.t. $rhost(req) = root$
BEGIN
  $n+ = b_i$
  $n_{host(req),l}+ = b_i$
END
for each host h
BEGIN
  $r_{h,l} = n_{h,l}/n_l$
END

$dom_l(h)$ with max($r_{h,l}$) is the top level-l CDN domain

Figure 4: Detecting top level-l CDN domain

with maximum portion of traffic (n_h/n) as the top host.

Suppose we identify i.cdn.turner.com as the top host in this step, we then try the next aggregated level of the top host domain: cdn.turner.com by repeating the counting procedure for level-3 domains. We can continue this procedure for further aggregated levels to get the top domain at each level. Note that we need to avoid the trivial levels such as .com or expanded levels .co.uk and so we stop at level-3. After getting the list of the top domains H_l and their traffic rates r_l, for each level $l = 3, 4, ..$, we use the following heuristic to decide which level of the top domain to use:

Starting from the lowest level (most aggregated) $l = 3$, we select the top level-3 domain as the CDN domain if the difference between the proportion of traffic for the top level-3 domain and the top level-4 domain is above a pre-set threshold t ($r_3 - r_4 > t$). In our system we choose t to be 5%. Otherwise we do not use the level-3 domain, and check the next level $l = 4$. We select level-4 if $r_4 - r_5 > t$. We continue this process until either we find a level l such that $r_l - r_{l+1} > t$ or l is the full host domain. We then select top level-l domain as the CDN domain.

In the cnn example, we have the top level-3 and level-4 domains as cdn.turner.com and i.cdn.turner.com, respectively. We start from the level-3 domain cdn.turner.com and check if the traffic rate difference between cdn.turner.com and i.cdn.turner.com is more than 5% of all traffic with referer field as cnn.com. This turns out to be true, so we choose cdn.turner.com as the CDN domain. Intuitively, when we choose a level l CDN domain, we are essentially combining traffic from all level $l + 1$ CDN domains. We choose to use an aggregated level of CDN domain only if such aggregation makes a difference, *e.g.*, if the aggregated traffic grows by 5%.

Special cases For less popular web sites, it is possible that they do not use any CDN services. One of the following may happen in such cases: (1) Most traffic comes from the origin domain, so that no other traffic will pass the threshold. As a result, no CDN is detected. This is fine since detection based on the origin domain already covers the majority of traffic. (2) The web site itself does not have much content. Most users who access this web site navigate to another web site. This is a corner case where the web site most likely does not provide any useful service. We ignore this case as being not of practical interest.

A more important case that we need to handle involves a web site that heavily uses services from some third party sites. For example, we found that reddit.com is a popular site (in the Alexa top 50 sites in the US) that relies on imgur.com for hosting images, although imgur.com is neither a CDN nor the main supporting domain owned by reddit.com. Given that our algorithm selects the most heavily referred site as the CDN, it will end up picking imgur.com as the CDN for reddit.com, which is not true even though its presence may be vital to the particular web service. However, we cannot allow clicks to imgur.com beyond the embedded requests as belonging to reddit.com service. In that sense, imgur.com should be considered similar to a third-party host, where embedded object accesses from the origin web pages are considered part of the web service in question while user clicks are not.

One other issue is that multiple web sites may claim the same CDN as its own CDN. For example, both cnn.com and adultswim.com use cdn.turner.com as the CDN. This is acceptable for our method, since we can trace back to the origin domain through the chain of referer fields and separate out the requests originating at different origin domains.

3.3 Embedded Object Detection

Besides the traffic from the origin domain and CDN (or main supporting domain), there is also traffic for downloading embedded content from third party web sites. This includes objects such as www.facebook.com/extern/login... and ad.doubleclick.com/adi/... in the example shown in Figure 1(a). In this section, we investigate two methods for detecting requests for fetching such objects—one based on the file-type extensions and the other based on timing.

Classification based on file-types: Our first observation is that certain file-types are almost always embedded. This includes css, js, swf, ico, json, and xml. Download of such files is always triggered automatically by the download of another (embedding) web page since such files are not useful on their own. Our preliminary inspection for a recent one hour trace collected at the large university campus network gateway shows that 15% in terms of requests and 11.3% in terms of bytes of all the HTTP traffic are for such files. In terms of percentage, they cover a smaller portion of traffic

than the CDN, but still significant in terms of volume especially for services that use these embedded objects more frequently. In addition, the almost certainly embedded nature of these objects makes it an accurate classification rule, and is also easy to implement. It also helps cross-checking with other heuristics as we discuss next. While it appears that relying on the file-type extension may allow gaming the system (*e.g.*, by simply renaming other content with these file-types), a user can only cheat if he can collude with the web-service provider since the URLs are managed by the service owner. Such a case therefore is highly unlikely in practice; we discuss cheating and collusion further in Section 6.

Classification based on timing: Our second observation is that the embedded objects are downloaded shortly after its referer page (called the base page). The time interval between the two downloads should typically be shorter than the time it takes for a user to browse a given web page and then click on a URL in the page to navigate to a different page. For convenience, we define the time interval between downloading the base page and the embedded links as *think time*. More precisely, we define think time $T_{think} = T_G - T_R$, where T_G is the time at which the GET/POST request for the embedded page has been sent, and T_R is the time instant at which the last response packet of the corresponding referer page arrived. For example, the think time for URL cdn.turner.com/index.jpeg in Figure 1(a) is interval between the time when the last response packet for its referer cnn.com is received and when the GET request for cdn.turner.com/index.jpeg is sent. Note that it is possible to have $T_{think} < 0$ since browsers can start downloading the embedded URL even before it finishes downloading the entire base page. Intuitively, think time is the time it takes for the browser to process the web page, extract any embedded URLs, and then send subsequent requests to download these embedded objects.

Naive timing heuristic: We may use the following naive heuristic for detecting embedded objects: A session is classified as embedded URL download if $T_{think} < T_{thresh}$, where T_{thresh} is a timing threshold, *e.g.*, 1 second. Unlike the file-type heuristic, the timing heuristic can generate both false negatives (missing requests part of the target web service) and false positives (including requests after the user navigates to third party web pages) depending on the value of T_{thresh}. To understand how practical the timing heuristic is, we take advantage of the file-type heuristic described in previous section.

Results from a real packet trace. We calculate the think time for all embedded file downloads based on the file-type heuristic in a full HTTP trace we collected at the large university gateway. Figure 5 shows the think time distribution. We observe that think time varies across a wide range. Although about 60% of think time falls below 1 second, 10% of think time is above 10 seconds. If we naively set the T_{thresh} to 10s, we will be able to capture almost 90% of all the embedded objects. But, this has the negative effect of increasing the false positives, since 10 seconds is sufficient time for

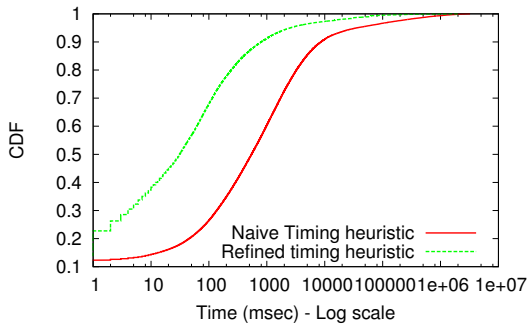


Figure 5: Think time distribution for embedded object downloads with both naive and refined heuristics.

a user to click on a third party link, which will then be classified as an embedded object. Clearly, it is not supposed to be counted as part of this web service, but will be because of the relatively high value of T_{thresh}. If we set T_{thresh} to say 100ms or even 1s, we will miss a large fraction of embedded objects (80% with $T_{thresh} = 100ms$ and 50% with $T_{thresh} = 1s$). It is clear that finding a fixed threshold that will work for a large number of web-services is not easy. Next, we discuss how to improve this further.

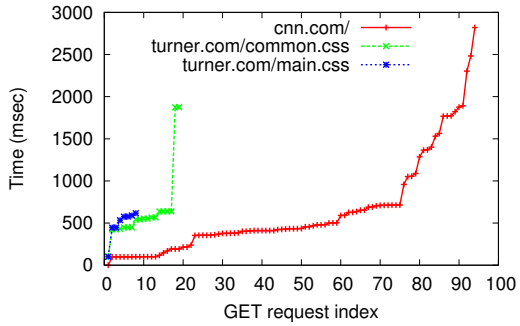


Figure 6: Think time for multiple embedded objects with the same referer

Refined timing heuristic: To better understand why the browser think times are sometimes exceedingly long, we inspect the time sequence of web page downloads more carefully. Figure 6 shows the timing for downloading multiple embedded URLs following the referer URL cnn.com/XXX. The X-axis shows the index of the URLs sorted according to the time when the GET/POST request is sent. URL 0 is the referer URL and other URLs are all embedded URLs. The Y-axis shows the think time for each URL. We observe that the think time increases almost linearly though the requests towards the end are spaced farther apart than at the beginning. The increased spacing is because third-party embedded objects or advertisements are among the last to be requested and they take more time to load as well. Except for this artifact, think times seem to be accumulating over consecutive embedded URL downloads. The reason is because

the browser typically processes each web page sequentially, and instead of sending out all requests for all embedded URLs at the same time, requests are spaced out over time. Browsers also restrict the number of parallel downloads as well, so downloads tend to be somewhat sequential.

Examination of the figure further reveals that the time offset when a GET request for an embedded object is made, relative to the request for the base page, is proportional to the number of GET requests for embedded object downloads. This is why choosing one fixed threshold is hard. Notice, however, that the gap between two adjacent requests is more constant and predictable, compared to the time differences between the original page and the embedded objects. Hence, we propose the use of the following *refined timing heuristic*: For each referer page R, maintain time T_A as the "latest activity time". The activity can be either the last response packet being received for this referer page (T_R), or be a GET request sent for an embedded URL of this referer page (T_G). When a new request is sent with referer R at time $T_{G'}$, we check if $T_{G'} - T_A < T_{th}$, where T_{th} is a chosen threshold. If the condition holds, we classify this request as an embedded URL for R and also update $T_A = T_{G'}$.

The think time distribution of the refined timing heuristic is shown in Figure 5. Most adjacent requests (almost 90%) are within about 100-500ms, which is much less than the human think time. Of course, if the chain of requests is long, the chances of false positives will increase since a user may click on some link. However, the chance of a user clicking on a link before the page completely loads is quite small and this approach therefore works for almost all practical cases. The problem with the naive timing heuristic was that it was trying to choose one threshold for all web sites, whereas the refined heuristic adapts to the number of objects and to the time taken to download all the previous objects.

Note that the tail still contains a small percentage of requests that were sent almost 1000s (about 16.67 minutes) after the previous request. While this may seem like a user click, our file-type heuristic indicates otherwise. On further investigation, we found that the browser was requesting the same embedded object several times (with the same referer). This happens every so often, as in an auto-refresh. If the browser refreshes certain objects automatically after a long time, it could be difficult for us to correctly classify these refreshes as embedded requests. However, for the refresh requests present in the trace, we observed that the file type was almost always of embedded variety. So, our file-type heuristic would have correctly flagged them as embedded.

3.4 Overall Algorithm

The cobble tree construction algorithm combines the multiple heuristics discussed above: CDN detection and embedded object detection based on both file-types and refined timing. To combine them, we run them as two separate procedures. In the first procedure, we detect the CDNs or main supporting domains for each target web site by using the CDN detection algorithm described in Section 3.2. Note that the administrator can choose to include multiple CDNs or supporting domains here based on the few top domains flagged using the algorithm. In the second procedure, we use the detected CDN domains along with the file-type heuristic and the refined timing heuristic to classify the traffic. Given a list of target origin domains, the goal of the algorithm is to classify each connection either as belonging to one of the target domains or as NULL when it does not belong to any target domain.

```
Algorithm:
for each request req
BEGIN
    if host(req) ∈ Origin
        root(url(req)) = Origin
    else if (host(req) ∈ CDN &
            root(referer(req)) ∈ Origin)
        root(url(req)) = Origin
    else if (url(req) is embedded file-type &
            root(referer(req)) ∈ Origin)
        root(url(req)) = Origin
    else if (req passes refined timing test &
            root(referer(req)) ∈ Origin)
        root(url(req)) = Origin
    else
        root(url(req)) = NULL
END
```

Figure 7: Overall classification algorithm for one domain

We classify each HTTP session based on the request message that the client sends to the server. Figure 7 shows the procedure for classifying a request. Each URL is associated with a "root" domain, which can be either NULL or one of the target domains. During processing, the heuristics are applied according to the specified precedence. Note that although conceptually we are isolating the session tree for each target origin domain, we do not need to maintain one unified data structure for the entire tree. Instead we can just maintain the root for each URL so that we know which tree this URL belongs to.

The precedence rules in the algorithm in Figure 7 are intuitive. If the host belongs to the origin domain, or to the CDN domain provided that referer's root belongs to the origin domain, then the host belongs to the origin domain. Then, we perform the file-type and timing checks, coupled with whether the referer's root belongs to the origin domain. Thus, if there is a false positive in the timing heuristic (*i.e.*, a GET request was sent to a third party host as a result of a user click and was not automatically fetched by the browser, but was misclassified by the timing heuristic) the overall algorithm is robust enough to stop further misclassification. For example, while the CNN page is loading, suppose a user clicks on some third party link, say facebook.com, on the CNN page. Because the page is still getting loaded, this user click may be inadvertently classified as an embedded object by the tim-

ing heuristic. The algorithm will set the root(facebook.com) to the origin domain (CNN). Further accesses to links from this third party page, however, will not match any of the rules because their referer would be facebook.com. For this one false positive to cascade into including an entire browsing tree, the user must repeatedly click on one link after another while the page is loading, with virtually no think time. We did not encounter such scenario in our trace.

The algorithm also handles URL shorteners flawlessly. An URL shortener redirects a short URL to the actual long URL using HTTP's 301 return code. When a browser fetches the long URL, the referer field remains NULL and so the cobble tree for the long URL can be formed in its entirety as if the redirect never happened. In a similar fashion near domain names (*e.g.*nyt.com and nytimes.com) can also be handled since usually the shorter name redirects the browser to the longer one. The algorithm can be made even more robust by filtering out the links to embedded objects by parsing the content within the HTTP response. This not only helps reduce the number of false classification of the URLs, but also remedies against auto-refresh and other fraudulent activities trying to circumvent the cobbling process. We however, chose not to implement that in our current algorithm due to the large overhead involved in storing, uncompressing and parsing the content within a HTTP response.

4. IMPLEMENTATION

In this section, we describe the implementation of the CobWeb system. This system is designed to operate on any network edge router with access to bi-directional traffic. It can either run directly in a gateway as a service blade or as a stand alone server that is directly linked to gateways via port mirroring. The system needs configuration information such as target web domain, their CDN names, timing thresholds and so on; it generates periodic reports about the cobbled session trees as output.

CobWeb system consists of a standard off-the-shelf packet sniffer (*e.g.*, libpcap) and a simple HTTP parser that allows us to extract various fields within HTTP packets. Our system can operate on live packet captures or in passive mode to operate on pcap traces. It maintains some minimal state regarding the HTTP session, and tracks requests and corresponding responses. It also stores some timing information to implement the embedded object detection heuristic.

We implement the CobWeb system in C++. Our total source code consists of 4,460 SLOC (Source Lines of Code). All our experiments were conducted on an Endance NinjaBox networking monitoring appliance [2]. Internally, it consists of an 2.5 GHz 8-core Intel Xeon E5420 procesor with a total memory of 16GB, running Linux operating system with a 10Gbps capture card. Current version of CobWeb is based on single process model, but it can be easily parallelized to take advantage of multi-core architectures. In our evaluation, we found that our unoptimized system can keep up with 10 Gbps line rate. Specifically, we found that the system took 2 hours 10 minutes to process a 5 hours trace, suggesting that it can easily keep up with line rates. The scalability is mainly because it only processes less than 0.1% of the actual data.

Trace	Size	Date	Duration
Field2011	227GB	July 29, 2011	1 hour
Field2012	512GB	Jan. 19, 2012	5 hour
Manual	108MB	Jan. 19, 2012	96min total

Table 1: Traces used in evaluation

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of CobWeb system. Our evaluation mainly focuses on measuring classification accuracy. We first explain the basic methodology for measuring the accuracy of our system, and then show results for each component of the algorithm. We also show some performance benchmarks for our unoptimized system prototype.

5.1 Evaluation Methodology

We use mainly two metrics to evaluate the efficacy of any classification algorithm—false negatives and false positives. False negatives occur if the algorithm misses part of the target web site traffic. False positives occur when the algorithm inadvertently includes traffic from web sites not part of the target web site's session tree. In the context of reverse-billing, false negatives cause over-billing for the user and false positives cause loss of revenue for the service provider; it is therefore important to minimize both so that the traffic accounting can be accurate.

Packet trace and ground truth In order to evaluate the web service cobbling algorithm, we take 70 out of the top 100 US web sites listed in Alexa, along with the top 26 popular web sites for users in the campus network. Note that we exclude the sites that heavily rely on HTTPS (*e.g.*, gmail.com) since our algorithm is not applicable there. We also exclude the sites that require user login since such sessions cannot be emulated by manual download and hence it is very difficult to obtain ground truth. Those 96 sites cover a wide range of service categories such as shopping, news, social networking, searching and so on. They also tend to use sophisticated web technologies, and hence is good for testing the effectiveness and robustness of the algorithms. We have collected two packet traces from a 10Gbps large university network gateway link that connects the campus to the rest of the Internet, both listed in Table 1. Note that the 512 GB is the size of the compressed Field2012 trace.

Real traces are very useful for understanding how users navigate through the web sites and what kinds of false positives and false negatives we can encounter when we deploy our CobWeb system in the field. However, such traces are not sufficient by themselves since we do not have access to the ground truth. Obtaining ground truth from traces in the field is challenging since there is no easy way to distinguish between user clicks and embedded downloads—a key re-

CDN domain	Origin domain	From Main	From Self	Others Others
2mdn.net	doubleclick.net	0.247	0.149	0.604
gstatic	google	0.854	0.006	0.140
images-amazon	amazon	0.523	0.207	0.269
imgur	reddit	0.749	0.107	0.144
imwx	weather	0.373	0.263	0.364
mzstatic	apple	0.919	0.001	0.080
turner	cnn	0.428	0.215	0.357
twimg	twitter	0.489	0.011	0.500
yimg	yahoo	0.544	0.105	0.351

Table 2: CDN traffic based on referrer domains

quirement for measuring false positives and false negatives.

Normally, one would assume we could just parse all the links from entire base page and we should be able to identify all the clickable links in the web page. If we see a GET request from the web page for one of the clickable links, we should be able to assume that it is a user click. The difficulty here is identifying the clickable links from the base page. Modern web sites heavily rely on javascript and it is not easy to just search for specific patterns such as "`<a href=.*>`" as in traditional plain HTML web pages. Indeed, we tried searching for such URLs and we found very few links in the entire trace that some user actually clicked on.

Thus, in order to make sure we evaluate our algorithm against credible ground truth, we also simulate a real user clicking on the 96 web sites and collect these traces. We use a web browser to open each of the main web pages and wait for 1 minute to record the traffic trace for each session. Hence we obtain 96 manual traces, also listed in Table 1. Given no interference from any other source, the packets will be pure ground truth. This of course gives us a limited evaluation since we cannot completely capture all the subtleties and complexities that are prevalent in the field. The combined evaluation using both manually clicked ground truth as well as real packet traces will cover more cases leading to a more thorough evaluation than either one of them can achieve individually. (Note that, with virtually no papers in this space, there are no clear established metrics or methodologies or data sources that we can directly use for evaluation.)

False negatives To evaluate false negatives, we mainly rely on manual downloads in a controlled environment. This ensures that we have access to credible ground truth. We run the algorithm on the manual traces and compute the *false negative rate*, the proportion of traffic that is in the manual trace but is missed by the algorithm. For convenience, we also use its complement *true positive rate* in our discussion.

False positives Unlike false negatives, it is difficult to evaluate false positives using manually downloaded web pages, because by construction, we are not injecting any interference in the user click emulation (hence, every GET request is part of ground truth). Here is where the real campus traces prove beneficial. But the issue with the real trace is that it is difficult to reliably isolate traffic between different web sites even with manual inspection, since we do not have any knowledge of user browsing behavior. For example, concurrent or overlapping HTTP sessions may be caused by downloading embedded content on the same web page, or caused by user opening multiple tabs in the browser, or caused by user quickly clicking through multiple URLs.

We address this problem by combining campus trace with controlled browsing as follows. First, we use our cobble algorithm to generate the *cobbled tree* for each user browsing session of each target web site in the campus trace. Recall that the root of each session tree is the starting point for each browsing session for a web page. The leaves of the tree include both the embedded objects that are automatically fetched from any server, and user navigations to other pages within the same web site (as in Figure 1(b)). In order to find out the accuracy of this tree, we use the manual trace as the ground truth (we call this the *ground tree*) and compare it against the cobbled tree. However, one issue is that it is not easy to emulate the user opening other pages within the web site; for ease of evaluation, therefore, we only consider root page downloads (called *pruned cobbled tree*).

In the ideal case, both trees, namely the ground tree and the pruned cobbled tree, should overlap perfectly; branches missing in ground tree implies false positives. However, there is one additional complexity we need to grapple with: Many web sites have very dynamic content which changes for almost every download. This can cause the trees from two different sessions differ and introduce "noise" into the comparison result. Such content is typically advertisements that are randomly selected or customized according to user profile or browsing history. For example, when we make two consecutive downloads for cnn.com, each download generates 150 and 148 GET requests, respectively. Among the 150 requests generated in the first download, 33 requests do not appear in the second download, with 25 being ads. If we compute false positives naively, several extraneous sessions not part of the ground tree will appear in the cobbled tree; this would not be an accurate classification though.

Hence we mark a node as false positive if the following two conditions hold: (1) it is in the pruned cobbled tree, but not in the ground tree; and (2) its URL belongs to third-party non-ads domain. If a node is marked as false positive, then the entire sub-tree below this node is also marked as false positive.

Intuitively, false positives can be classified into two categories: (a) user clicking on third-party non-ads link; and (b) user clicking on ads link. Obviously, case (a) is covered by the above heuristic. For case (b), when a user clicks on an ads link to navigate to a third-party web site, *e.g.*, clicking an ads of nytimes.com served by doubleclick.net on cnn.com page, although the first download may contain URL of ads link (and hence not covered by the above heuristic), the majority of traffic will be for downloading pages on nytimes.com along with embedded objects, which is covered by the heuristic. As a result, we can capture all false positives involving non-ads links and majority of false positive traffic for ads.

5.2 CDN Detection

In order to test the CDN detection heuristic, we first run the CDN detection algorithm with the Field2011 trace and then verify the detection result by using tools such as whois, dig and google search. Table 2 shows an example of the detection result. CDNs for most web sites are correctly detected except reddit.com, for which imgur.com was mistakenly identified as the CDN or main supporting domain. The reason was that lots of reddit users are browsing image links of imgur during the measurement period.

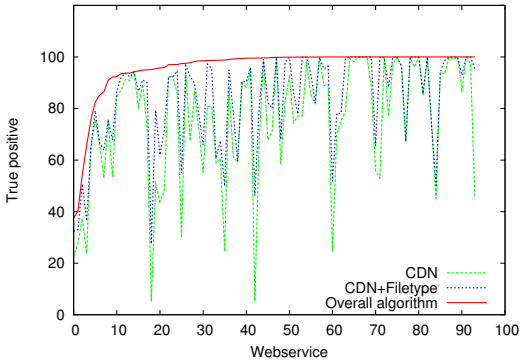
Table 2 also shows the traffic break-down for each CDN (or main supporting) domain (for a subset of websites for space reasons). Traffic for each CDN domain can be referred by either the main domain, the CDN domain itself, or other domains. We observe that although in most cases the majority of traffic for a CDN is referred by its main domain, significant fractions of traffic for many CDNs are referred by external web sites. This confirms our intuition that classification just based on host domains is not sufficient. For example, even though 2mdn.net is the main supporting domain for doubleclick.net, majority of its traffic is referred by external web sites to show ads, and hence should be classified as part of the corresponding external web site traffic. The same can be said for twitter.com's supporting domain twimg.com.

We further run the CDN detection algorithm with the Field 2012 trace. For all 96 top sites, we find that the algorithm correctly identified CDNs for 89 sites (92.7% accuracy). In the other cases another third party site is marked as CDN incorrectly since there is no separate CDN domain for this web site and the third party domain exceeds the 5% threshold. There are also two web sites that use more than one CDN domain, so the algorithm just picks the top one.

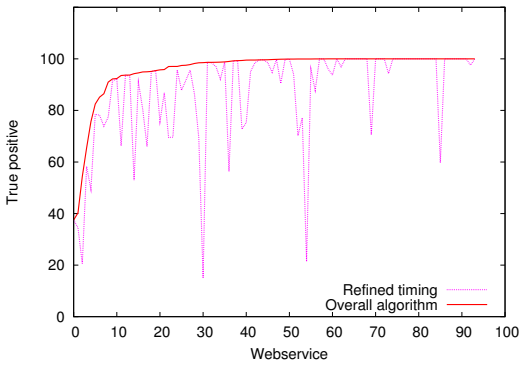
Once the CDN or main supporting domain is determined for the target web site, we can apply the CDN heuristic to classify traffic. In order to find out the proportion of traffic classified by using the CDN heuristic, we construct the cobbled tree as follows: Starting from the target root domain with empty Referer field, we include a HTTP request into the tree if its host belongs to the root domain or CDN domain, and its referrer also belongs to the tree. Here we only evaluate the true positive rate since there are no false positives for the CDN heuristic. We use the 96 manual traces for this evaluation. Figure 8(a) (the green line) shows the proportion of traffic correctly identified by using the CDN heuristic. We observe that true positive rate varies across a wide range between 5.09% to 100%. The average true positive rate over the 96 sites is 75.68%. This indicates that the CDN heuristic is very effective for many web sites, but if used as the only heuristic, it is not robust enough to classify traffic for all web sites.

5.3 File-Type Heuristic

We next investigate how much the file-type heuristic can further improve the classification result. We construct the cobbled tree in a similar way as before, and in addition, include a session if the URI is for an embedded file type and its referrer host belongs to the tree. Similar to CDN heuristic, file-type heuristic does not generate false positives and we use the 96 manual traces for evaluation. Figure 8(a) (blue line) shows the proportion of traffic classified correctly by combining file-type heuristic with the CDN heuristic. Note that such embedded files can be downloaded from either CDN or third party. As a result, the two detection methods have certain overlap between their classification results. We observe the effectiveness of the file-type heuristic varies across web sites. On an average, the method with combined heuristics can correctly classify 83.75% of total traffic, an 8.08% increase over the original CDN heuristic. In other words, the file-type heuristic classifies an additional 8.08% traffic for embedded file downloads from third party web sites.



(a) True positives for CDN and file-type heuristics



(b) True positives for refined timing heuristic

Figure 8: Proportion of traffic correctly classified by individual heuristics and overall algorithm. Webservices are sorted based on the True positive rate for the overall algorithm

5.4 Refined Timing Heuristic

To evaluate effectiveness of the refined timing heuristic, we construct the cobbled tree using this heuristic, and use the method discussed in Section 5.1 to evaluate mainly true positive rate. We use the T_{thresh} value of 500ms for these results. Since, this heuristic is only part of our overall algorithm, we later study the sensitivity of the T_{thresh} value

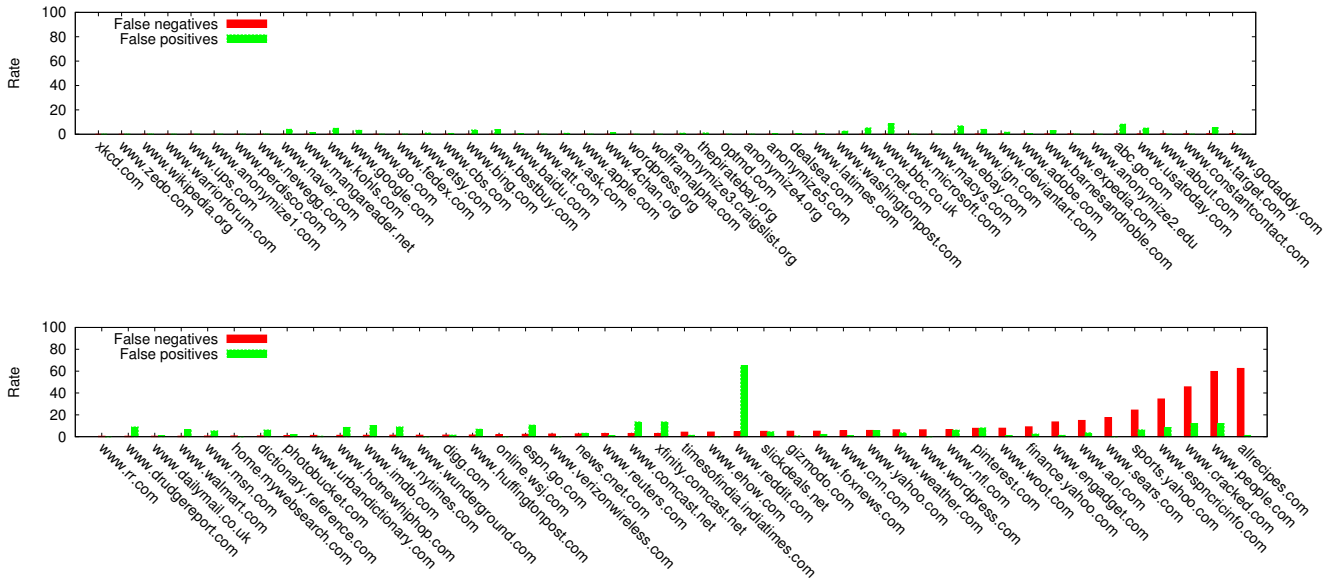


Figure 9: False positive and false negative for all the webservices with timing threshold 500 ms.

on the performance of the overall algorithm. We evaluate true positive rate by using the 96 manual traces as before. Figure 8(b) shows the true positive for each target web site. We observe that the true positive rate varies across a wide range between 14.98% to 100%. The average true positive rate over all web sites is 87.11%. Again, as shown in the Figure 8 the combination of all these heuristics has the best chance of correctly classifying most of the traffic. We discuss this further in the next subsection.

5.5 Overall Algorithm

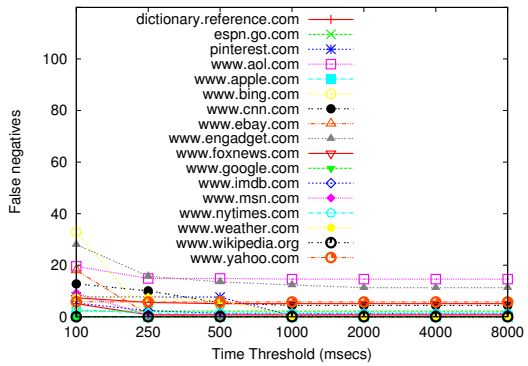
We now evaluate the overall algorithm (shown in Figure 7) that combines all the heuristics. We first use the manual traces to evaluate false negatives, and then use both Field2012 campus trace and manual traces to evaluate false positives. In the latter case, there are typically multiple browsing sessions for each of the 96 sites. We take the average false positive rate of all the sessions for the same site. Figure 9 shows both the false negative and false positive rates by using the overall algorithm. Timing threshold is 500ms. Result is shown in two rows, sorted based on false negative rate. We observe that the overall algorithm performs very well for most web sites. The average false positive rate across all web sites is 3.54%. With very few exceptions, the false positive rate is below 10%. One anomaly is www.reddit.com with false positive 65.12%. We conjecture that this is caused by very frequent content update due to active user postings. Since we compare all sessions within the 5 hour field trace with just one ground truth trace, the "ground truth" is outdated for most sessions. To verify this hypothesis, we collect a shorter 20 min trace on the same campus gateway along with a ground truth trace. For the 48 reddit sessions that we capture, the average false positive rate is now 5.2%, indeed much lower. This confirms that the high false positive rate we have observed in Figure 9 is indeed caused by the evaluation artifact rather than the cobbling algorithm.

The average false negative rate is 4.14% across all web services. Except the last 8 sites, all false negatives for all other sites are below 10%. Further investigation of the trace shows that the high false negatives for the last 8 sites are mostly caused by flash video players. Unlike the browser, the video players often have empty Referer field in their GET requests. Recall that our algorithm relies on Referer field for cobbling the session tree; if the Referer field is not set, our algorithm assumes it is not related to this session tree and will miss the traffic completely. Not setting the Referer field is alright for the CDN or main domain accesses, since they will still be accounted for, although as part of a different cobbled tree.

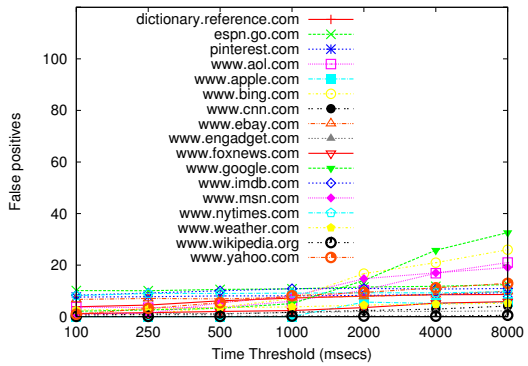
We think this problem can be addressed by correlating the sessions without Referer field with those with Referer field. For example, when we see a GET request with empty Referer field, we take its URL and find the "best matching" URL of another request among all the cobbled trees of the same user within a certain time period, say 5 sec. The intuition here is that if the flash player retrieves video from a domain, the browser is likely to retrieve some other content from the same domain. We have tested out this approach on the four domains with highest false negative rate, and found that their false negative rate is reduced to 19.1%. Work is still needed to further validate such approach, and exploring other alternative correlation methods.

5.6 Sensitivity to the Timing Threshold

The only parameter in our algorithm is the timing threshold T_{thresh}. Clearly, setting T_{thresh} too high leads to fewer false negatives, but also increases the number of false positives. It is therefore important to identify a good threshold that can balance the two. We vary the timing threshold and observe its impact on the false positives and false negatives;



(a) False negatives



(b) False positives

Figure 10: Sensitivity to the timing threshold T_{thresh}.

results for top 17 web services (in terms of traffic volume) in the Field2012 trace are shown in Figure 10.

In Figure 10(a), we show that we can decrease the false negative percentage of some of the web sites by increasing the timing threshold. For example, bing's false negative percentage comes down from 35% to almost zero when the threshold is increased from 100ms to 250ms. But we also find that the false negatives of several web sites are not affected by the threshold. One reason is that if the Referer field is missing, there is not much gain in accuracy by increasing the timing threshold.

On the other hand, false positives generally increase with the increase in the timing threshold (shown in Figure 10(b)). google is most sensitive to the timing threshold since its search results have many third party web sites, and also user tends to click on search results faster than when they browse individual web sites. But web sites such as cnn and wikipedia are much less sensitive to increase in timing threshold. One reason is that many web sites are designed to attract users to their own pages and hence have very few clickable third party URLs. Such differences between different web sites in their sensitivity to the timing threshold suggests that it may be beneficial to adopt a different timing threshold for different web sites. For the web sites we have studied we find that timing threshold of 250ms to 500ms seems to be a good trade-off between false positive and false negative rates.

App name	Main	Supporting	Ads	Third party
cnn	58.69	41.31	0.00	0.00
yelp	30.37	69.63	0.00	0.00
washingtonpost	90.21	0.02	9.77	0.00
amazon	62.98	37.02	0.00	0.00
nytimes	98.90	0.00	0.87	0.23
facebook	11.15	87.73	0.00	1.11
twitter	0.00	98.13	0.00	1.87
imdb	0.00	97.68	0.29	2.03
bestbuy	80.56	16.67	0.00	2.78
dictionary	56.96	0.00	39.04	4.00
bbc	17.82	72.80	0.38	9.01
huffingtonpost	18.33	67.56	4.85	9.26
abc	81.01	9.72	0.00	9.27
engadget	0.28	79.90	1.42	18.39
ebay	4.32	84.18	0.00	11.51
wunderground	69.80	0.00	0.28	29.93
weather	13.94	19.64	30.44	35.99

Table 3: Mobile app traffic break-down per domain in percentage of bytes

6. DISCUSSION

Mobile Apps. Browsers on the mobile phone behave in a fashion similar to the desktop browsers in terms of filling in the Referer fields. However, mobile apps often ignore optional fields such as User Agent and Referer. To investigate this, we ran several mobile apps on an Android phone, randomly navigated within the apps, and captured traces. We found that Referer field is empty in majority of the requests. Thus, we believe that our cobbling algorithm, in its current form, may not be directly applicable to mobile apps. However, further investigation reveals that the mobile apps send requests to a significantly small number of domains as compared to the web browsers (refer to Figure 2). We found that the average number of top level domains per app is just 5. Moreover, as Table 3 shows, for majority of the apps more than 90% of the traffic belongs to the main and the supporting domains for the app. Hence it is likely that CDN detection and simple static traffic rules may work well for mobile apps, although a more careful study is still needed. We plan to investigate such techniques as part of our future work.

Video Services. There are several popular video service sites that also use HTTP streaming to deliver video content. Examples include Youtube, Netflix, and Hulu. Our algorithm does not directly apply to such web sites since they typically have multiple domains for supporting content delivery. For example, both Netflix and Hulu use three CDNs (Akamai, Level3, and LimeLight). Youtube also has several domains to support video and image download although they do not use third party CDNs. However, since there are very few of such large scale video service sites, it is affordable to make specific rules for such web services. Such rules can be made based on CDN host or domain names as well as signatures in the message [4].

Cheating and Collusion. The cobbling algorithm can potentially be vulnerable to cheating by a user since the client browser determines what Referer field to set. For instance, a

user can disable the Referer field so that the traffic cannot be easily associated with any web service. Or a user can modify the Referer field to falsely associated the traffic in question with a different web service. However, unless most of the Internet users cheat, we can always correlate individual user sessions with the common and vast majority and hence can detect anomaly and blacklist such individuals. Collusion between users and web service providers make the detection more difficult. However this should be very rare since there is not much incentive to do so in practice. For instance, in the case of reverse billing, it is unlikely that the web service provider will collude with a user since one of the parties has to pay for the traffic.

Prefetching and Auto-Refresh. Prefetching at the origin or CDN server does not affect our algorithm. In addition, prefetching by the client browser is similar to user accessing such content, so it will be classified accordingly. There are several scenarios for auto-refresh. First, auto-refresh of the entire page is similar to user accessing the page again, and hence it is classified correctly. Second, auto-refresh of individual objects such as ads are also fine if the same link has been classified before. Last but not the least, if the auto-refresh is for a different link, then we can search for similar links in the cobbled tree similar to what we have done for sessions with missing Referer fields. Further work is needed to fully evaluate the impact of such strategy.

7. RELATED WORK

Major search engines and portals such as Yahoo and Bing have long been offering catalog of web sites based on the type of their services. Research work has also been done in the past to catalog web services by using techniques such as machine learning [17, 9]. The focus in our paper is not to catalog the web services, but to isolate each target individual web service traffic to assist better billing and service management. To the best our knowledge, this is the first work to consider this problem and as such very few related works exist.

There exists related work in traffic classification and identification in general, especially at the application level [14, 19, 8]. Moore and Zuev have proposed a machine learning approach based on Bayesian analysis to classify Internet traffic into categories such as P2P, multimedia, WWW, etc [15]. Karagiannis *et al.* propose BLINC system [8] for traffic classification in the dark using communication graphlets for characterizing different types of applications. A survey on traffic classification techniques based on machine learning is given in [16]. Prior research has also been done to identify specific applications such as Skype [5, 7] based on payload signatures and packet timing characteristics.

There have been some measurement studies to understand the new web technologies such as Ajax (*e.g.*, [20, 12, 10]). Our goal in this paper however is to provide a mechanism to identify sessions corresponding to a web service.

8. CONCLUSIONS

We have presented the CobWeb system for in-network cobbling of traffic associated with a given set of web services. Such system can enable new types of monitoring and measurement capabilities, and have the potential to even enable new revenue models such as reverse billing for service providers. While the classification algorithm is based on a combination of multiple heuristics based on CDN and HTTP request timing, the association between different web requests is made possible using the HTTP Referer field, which is an essential component of Internet advertising industry today. Our extensive evaluation suggests CobWeb can achieve low false positive and false negative rates, and can potentially sustain a 10Gbps link.

We view CobWeb only as the first step towards solving the complex problem of web service classification. Specifically, CobWeb cannot currently handle encrypted traffic (*e.g.*, using HTTPS) since it relies on information within the HTTP requests that may not be visible in the network. While most traffic today is not HTTPS, extensively covering all such cases is a big challenge.

9. REFERENCES

[1] Allot. http://www.allot.com.
[2] Endace ninjabox. http://www.endace.com.
[3] Sandvine. http://www.sandvine.com.
[4] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *IEEE INFOCOM*, 2012.
[5] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli. Revealing skype traffic: when randomness plays with you. In *ACM SIGCOMM*, 2007.
[6] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. In *ACM SIGCOMM*, Aug. 2004.
[7] F. Hao, M. Kodialam, and T. Lakshman. On-line detection of real time multimedia traffic. In *ICNP*, 2009.
[8] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. In *ACM SIGCOMM*, 2005.
[9] I. Katakis, G. Meditskos, G. Tsoumakas, N. Bassiliades, and I. Vlahavas. On the combination of textual and semantic descriptions for automated semantic web service classification. In *AIAI*, 2009.
[10] E. Kiciman and B. Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. *SIGOPS Operating System Review*, 41(6):17–30, 2007.
[11] R. R. Kompella and C. Estan. The power of slicing in internet flow measurement. In *IMC*, 2005.
[12] M. Lee, R. R. Kompella, and S. Singh. Active measurement system for high-fidelity characterization of modern cloud applications. In *Proceedings of USENIX Conference on Web Applications*, 2010.
[13] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *ACM SIGMETRICS*, 2008.
[14] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. pages 313–326, October 2006.
[15] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *ACM SIGMETRICS*, 2005.
[16] T. T. Nguyen and G. Armitage. A survey of techniques for Internet traffic classification using machine learning. In *IEEE Communications Surveys and Tutorials*, 2008.
[17] N. Oldham, C. Thomas, A. Sheth, and K. Verma. Meteor-s web service annotation framework with machine learning classification. In *Int. Workshop on Semantic Web Services and Web Process Composition*, 2004.
[18] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani. Fast monitoring of traffic subpopulations. 2008.
[19] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-Service Mapping for QoS: A Statistical Signature-based Approach to IP Traffic Classification. In *ACM IMC*, 2004.
[20] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The New Web: Characterizing AJAX Traffic. In *International Conference on Passive and Active Network Measurement*, April 2008.
[21] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: towards programmable network measurement. In *ACM SIGCOMM*, 2007.