Integrity of the Web Content: The Case of Online Advertising

Nevena Vratonjic, Julien Freudiger and Jean-Pierre Hubaux
School of Computer and Communication Sciences, EPFL, Switzerland
firstname.lastname@epfl.ch

Abstract

Online advertising is a major source of revenues in the Internet. In this paper, we identify a number of vulnerabilities of current ad serving systems. We describe how an adversary can exploit these vulnerabilities to divert part of the ad revenue stream for its own benefit. We propose a collaborative secure scheme to fix this problem. The solution relies on the fact that most of online advertising networks own digital authentication certificates and can become a source of trust. We also explain why the deployment of this solution would benefit the Web browsing security in general.

1 Introduction

Over the last decade, online advertising has become a major component of the Web, leading to large annual revenues (e.g., $22.7 billion in the US in 2009 [17]). Most of online services rely on online advertising as their main source of revenue. Unsurprisingly, the ad revenue has caught the eye of many ill-intentioned people who have started abusing the advertising system in various ways. In particular, *click fraud* has become a phenomenon of alarming proportions [14].

Recently, a new type of *ad fraud* has appeared, consisting in the on-the-fly modification of the ads themselves. A prominent example is the *Bahama botnet*, in which malware causes infected systems to display to end users altered ads as well as altered search results (e.g., Google) [6]. Another reported example of such a botnet is Gumblar [13].

Botnets usually consist of compromised end users' PCs that are turned into *bots* and controlled remotely by a bot master. Lately, botnets started targeting wireless routers: from the botnet's point of view, wireless routers have the advantage of being almost always connected to the Internet and of being vulnerable [11, 15]. Once a wireless router is infected with a malware and turned into a bot, the botnet master can instruct the bot to perform on-the-fly modifications of the ads of the web pages that pass through the router.

If the modification of ads is successful, users are displayed ads that are different from what they would otherwise be. Consequently, users may click on altered ads and generate revenue for the bot master instead of the advertising network (AN). Thus, the modification of the ads undermines the business model of ANs. In addition, such attacks may also negatively affect the security of end users, the reputation of websites and the revenue of "legitimate" advertisers.

In order to prevent such man-in-the-middle attacks, well-known solutions consist in deploying authentication and data integrity mechanisms to help guarantee the end-to-end security of communications, as done with HTTPS [30]. Nevertheless, such mechanisms have various drawbacks that hinder their large-scale deployment [30, 31, 36]. First, these authentication mechanisms make use of digital certificates to enable the authentication of web servers. Digital certificates are inherently expensive because central trusted authorities must manually verify the identity of web servers. An alternative for web servers is to generate self-signed certificates. These certificates are hard to validate for end users and could be tampered with in the man-in-the-middle attacks. In order to help users properly verify self-signed certificates, the system of network notaries is built that monitors over time consistency of web servers' public keys [36]. However, this solution also has several drawbacks [36]. Second, the protection of the web content relies on cryptographic operations that induce a large computation cost on servers [30].

For these reasons, various alternative approaches were proposed to protect web content in an efficient fashion [28, 29]. Previous work suggests to *encrypt* all web communications using opportunistic encryption [28]: a secure channel is set up without verifying the identity of the other host. This provides a method for detecting

tampering with web pages, but only for expert users who know how to check certificates. However, it does not defeat man-in-the-middle attacks because an adversary can still replace the certificates used for authentication to impersonate websites. Another approach focuses on the protection of web content *integrity* by detecting in-flight changes to web pages using a web-based measurement tool called web tripwire [29]. Web tripwire hides javascript code into web pages which detects changes to an HTTP web page and reports them to the user and to the web server. Web tripwire offers a less expensive form of a page integrity check than HTTPS but as acknowledged by the authors is non-cryptographically secure.

In this work, we first model the threats to ad serving systems caused by on-the-fly modifications of ads. We show novel attacks by selfish adversaries that generate significant revenue. Second, we evaluate the feasibility of such attacks by implementing a proof of concept code on wireless routers. We show the limitations that botnet designers may face in practice to deploy such attacks. Finally, we propose a novel solution to fix the problem of ad fraud and provide integrity of web content. Our solution is based on the collaboration of the parties involved in advertising systems. We rely on the fact that most of advertising networks own digital certificates and can become a source of trust. We use hash-chains [20] to provide data integrity in an efficient fashion. We show that our scheme is significantly more efficient than HTTPS and provides the same level of data integrity. Importantly, all involved parties have incentives to participate in this collaborative scheme.

2 Advertising on the Internet

Internet advertisement is generally constituted of a short text, an image, or an animation embedded into a web page and linking to an advertised website. The purpose of an ad is to generate traffic to the advertised website and, consequently, to increase the revenue for the advertised products or services. Hence, for many of the websites users visit, a number of advertisements appear together with the content of a Web page.

2.1 Ad Serving Architecture

Ads are embedded into web pages either through an ad serving system, or by websites themselves. The prevalent model of the Internet advertisement serving architecture is depicted in Figure 1. In this model, *Advertisers* (AV) subscribe to an *Ad Network* (AN) whose role is to automatically embed ads into related web pages. Ads are stored at *Ad Servers* (AS), which belong to the ad network. Ad networks have contracts with *Websites* (WS)

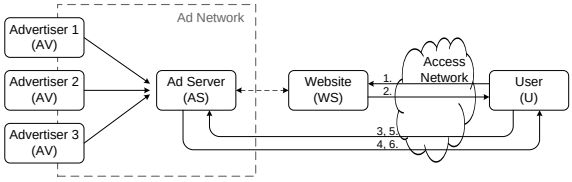

Figure 1: The advertisement serving architecture. WS and AS have a contractual agreement (dashed arrow) that lets AS add advertisement to WS's web pages. The protocol illustrated with arrow numbers is given below.

that want to host advertisements. When a *User* (U) visits a website that hosts ads (Figure 1, step 1), the user's browser starts downloading the content of the web page (step 2), and is then directed to one of the ad servers belonging to the ad network (step 3). During the first communication with the ad server, a script is served to the user (step 4) that executes on the user's machine and fetches ads from the ad server (step 5). The ad server chooses and serves ads that match users' interest (step 6) in order to maximize the potential ad revenue.

The protocol can be modeled as follows:

1. $U \rightarrow WS$: GET URL_{WS}
2. $WS \rightarrow U$: p
3. $U \rightarrow AS$: GET URL_{AS}, WS_{ID}
4. $AS \rightarrow U$: script
5. $U \rightarrow AS$: GET $URL_{AS}, WS_{ID}, c_{AS}$
6. $AS \rightarrow U$: ads

where $\rightarrow$ means communications over HTTP.

With the first two messages (Figure 1), U fetches a web page p identified with URL_{WS} from a WS. The web page p contains an ad frame f redirecting U towards the ad server AS. The redirection includes the URL_{AS} of the AS and a unique identifier WS_{ID} of the WS (required by the AS to properly transfer potential advertisement revenue to the WS). With the next two messages, U fetches typically a *script* from the AS (e.g., Javascript) that executes locally on the user's machine, and collects certain parameters that influence selection of ads by the AS, including the HTTP cookies c_{AS} if were deposited by AS during previous interactions with U. Cookies uniquely identify users and enable profiling of their browsing preferences. With the last two messages the script fetches ads from the AS. The browser merges the ads with previously downloaded elements of p.

This approach is widely used and has several advantages: (i) the HTML code that directs users to fetch ads is simple and easy to maintain, as only one line of code (a reference to the Javascript) is added in the ad frames, (ii) it is scalable, as the workload is distributed over users, (iii) it allows ad servers and advertisers to keep the control, as ads are stored and maintained at their servers, and

(iv) it enables ASs to track users across multiple websites. There are several drawbacks as well. Because users fetch ads from third-party servers, the ad serving technology slows down the display of web pages, consumes extra bandwidth and affects the privacy of users [26].

There are other online advertisement serving techniques. For example, a PHP script running on a website can locally embed advertisements and serve them to users together with the content of a web page. This technique is not very popular because it puts more workload on the Web servers compared to the previous approach, thus it does not scale well.

2.2 Targeted Advertisement

A notable difference between online and traditional ad serving (e.g., television, radio...) is that online ads can be *targeted* to individual user's interests. Ads that appear on a user's screen are selected by ASs in real time to match the user's interests. Typically, ads are targeted to the content of the web page hosting the ad, users' interests extracted from their browsing history (e.g., using HTTP cookies) or users' geographical location.

Targeted advertising aims at increasing users' interest in the advertised products. At the AS, users' interests can be expressed with *keywords*. The AS associates ads with each keyword and runs auction algorithms to select the most relevant ads and the order in which they appear on the web page, with the goal of maximizing the profit of both advertisers and websites hosting the ads.

2.3 Revenue Models

There are two main revenue models: Advertisers may pay the ad network on a per *impression* or per *ad click* basis. In the per impression model, advertisers pay the AN for the exposure of their ads to end users, i.e., there is a *cost-per-mille* (CPM) (cost to expose one ad to one thousand users). In per ad click model, advertisers pay the AN a *cost-per-click* (CPC) for each user-generated click that directs the user's browser to the advertised website. The AN gives a fraction of the ad generated revenue to the WS that hosted the ad.

These models provide incentive to participate in the ad serving system: the AV earns the revenue created by ads, the AS earns money for storing the ads and finding a proper WS to display ads, and the WS earns money for hosting ads and directing the U towards advertised WS.

3 Threats

In this section, we define the adversary model considered through the rest of the paper and we identify a number of possible attacks on ad serving.

3.1 Adversary model

We consider both a *selfish* adversary intending to take advantage of the ad serving system and a *malicious* adversary intending to harm it. A selfish adversary exploits the system with the goal of diverting part of the ad revenue for itself. In contrast, a malicious adversary may perform any type of attacks on the ad system.

The adversary can be part of the ad serving architecture (Figure 1) or part of the access network that provides Internet connectivity to end users. As discussed in Section 2, all entities of the ad serving architecture benefit from the delivery of ads to end users. However, the access network that carries all users' traffic does not receive any ad revenue. Thus, the access network may be tempted to tamper with the transiting data to generate benefits for itself. In this paper, we focus on an adversary located in the *access network* that is a legitimate member of the network.

We assume an *active* adversary $\mathcal{A}$: $\mathcal{A}$ controls the access network and has the ability to *eavesdrop*, *alter*, *inject* and *delete* messages. In other words, it can perpetrate Man-in-the-Middle attacks. Such an adversary can take various forms in practice. It can be *local* or in the worst case *global*. For example, a local adversary can deploy its own network of access points (APs) or hijack existing wireless access points [33]. Lately, access points have been compromised by botnets [11]: APs are infected with a malware (i.e., turned into *bots*) and remotely controlled by a botnet master. With botnets $\mathcal{A}$ can increase the scale of the attacks.

Today, ISPs need to invest into the infrastructure to support the increased demand for bandwidth and to accommodate government requirements (e.g., blocking part of the P2P traffic) [1]. There is no clear answer on how ISPs will obtain a return on that investment. Thus, ISPs might be tempted to abuse the control over the Internet traffic they transport to generate additional revenue. ISPs have the means to monitor and modify the packets in real time (e.g., Deep packet inspection (DPI) [21]). DPI is a packet filtering technology that allows for the automatic examination and tampering of both the header and data payload of packets.

3.2 Attacks on Ad Serving

Given the large revenue generated by online advertising, an adversary has significant economic incentive to exploit the online ad serving in order to divert part of the revenues to its own benefit. In this section, we describe various MitM attacks against ad serving that can be perpetrated by an adversary $\mathcal{A}$ located in the access network.

3.2.1 Injecting Advertisements

$\mathcal{A}$ can inject advertisements in web pages traversing the access network by either *adding* new advertisements or *replacing* already embedded advertisements. By injecting ads, $\mathcal{A}$ thus bypasses the traditional ad serving model. The attack is successful if the adversary can obtain revenue with the injected advertisements. The achievable revenue depends on users seeing the advertisement (CPM model) or finding an ad interesting and clicking on it (CPC model). To maximize the success of the attack, the adversary should thus increase the *visibility* of injected ads and *target* them to users' interests and the content of corresponding web pages.

We present various types of injections of ads.

Pollution attack We call a *pollution attack* the injection of advertisements not necessarily targeted to the web page's content or users' interests that is highly visible. Rogers, a Canadian ISP, was reported to add content, notably advertisement for their own services, into any web page that traversed their access network [35]. This was done by injecting into web pages a single line of code that causes the user to fetch and execute a Javascript as if it was part of the content of the web page.

Pollution attacks generate revenue for the adversary but also spoil the appearance of web pages and may thus harm the reputation of websites.

Targeted attack A more sophisticated version of the pollution attack consists in adding ads *targeted* to users' interest and the content of web pages. For example, $\mathcal{A}$ can add highly targeted and visible ads into search engine results. Search engines facilitate targeted advertising as search queries indicate users' interest at the considered moment. In addition, surveys have shown that more than half of the users click on one of the first two native (i.e., non-sponsored) results [25] of Search Engine Result Pages (SERP). Knowing this, $\mathcal{A}$ can add its ads at the top of SERPs, resulting in a substantial increase of users' traffic on a website of the adversary's choice. $\mathcal{A}$ could also commercialize such services to advertisers.

Similarly, the adversary could inject ads into location-based services (LBS) results. LBS provide points of interests (POIs) near users' location. LBS results are typically presented on a map (e.g., Google maps) to help users locate POIs. Usually, LBS include ads in their results that are targeted to users' location and interests. Knowing this, $\mathcal{A}$ can add its ads at the top of the list of POIs. Hence, when a user queries a LBS for a POI in his vicinity, the adversary can influence the user's choice.

In practice, several ISPs already work with advertisers to legally add advertisement to web pages. For example, Phorm [2], is a personalization technology company that offers an ad serving platform to ISPs. It currently partners with Virgin Media, UK and is engaged in market trials with KT, Korea. Another example is "free" ISPs [32]: their business model consists in providing free Internet access and generating revenue from injected ads.

3.2.2 Deleting Advertisements

An adversary can also remove ads from web pages. For example, an ISP can automatically filter out all the ads and offer this as a service to its customers. Blocking ads is already possible at the end users [3], but doing it network-wide would work transparently for users. In addition, $\mathcal{A}$ could have an agreement with certain advertisers or ad networks to filter out ads from competitors. In practice, the infrastructure to block specific HTTP content is in place.

3.2.3 Website Impersonation Attacks

The adversary can manipulate the website IDs to control which website earns revenue from hosting ads. We call such attack a *Website impersonation attack*. For example, if the adversary registered its website with the ad network, then it could replace the IDs of other websites with its own website ID. Consequently, the adversary would be paid for all the traffic that was generated at victim websites. In practice, such IDs are not protected by any cryptographic means and could thus easily be replaced.

3.2.4 Phishing Attacks

Instead of stealing the ad revenue, a malicious adversary can replace legitimate ads with malicious ads to divert users' traffic towards phishing websites. Such an adversary affects the security of users as it was shown that users usually believe that advertised links are trustworthy. For example, $\mathcal{A}$ can replace the sponsored links in search results with links to phishing websites.

3.3 On-the-fly Ads Modifications

In this section, we explain how the MitM attacks on advertisements can be implemented in practice.

First, an access network must identify ad objects in the HTTP traffic. This can be done by checking the destination IP addresses or URLs of the requested objects. The adversary can leverage on the lists of IP addresses and domain names of the most popular ad servers that are used by ad blocking softwares [3] to filter out advertisements. If there is a match between a URL of a requested object and a URL in the list of ad servers, $\mathcal{A}$ can conclude that the requested object is an advertisement. In addition, an adversarial ISP can use DPI technology to identify packets containing ads.

Once $\mathcal{A}$ identifies ad objects, it can alter the ad traffic either: (i) locally, without the help of external resources or (ii) remotely, by redirecting users' requests towards servers chosen by the adversary. To locally alter ad traffic, the adversary relies only on the locally available resources (e.g., an access point). To remotely alter ad traffic, the adversary can for example redirect the ad traffic to another AS by modifying *URLs* of objects referenced in ad frames. When a user fetches a web page, $\mathcal{A}$ modifies on-the-fly the payload of packets carrying the *URLs* of the ASs. Hence, ads are fetched from different ASs.

3.4 Economic Impact

In this section, we assess the potential revenue of an adversary modifying ad traffic on-the-fly as explained in Section 3. We take a *bottom-up* approach: we model the browsing behavior of users, estimate the number of ads affected by attacks and derive the ad revenue at stake.

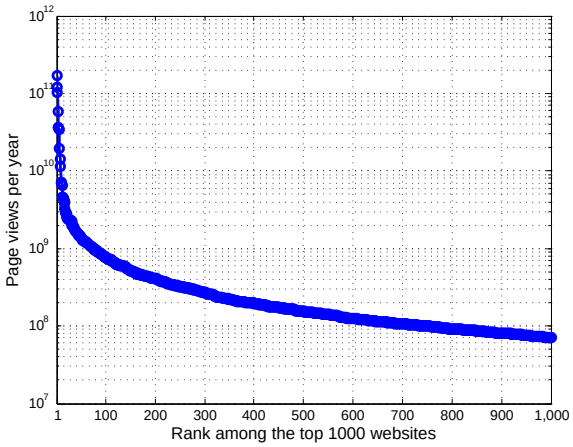

Figure 2: Popularity of the top 1000 websites based on page views per year.

Users' browsing behavior is profiled by web analytic companies, such as *Compete.com*. We base our analysis on measurement data we obtained from *Compete.com* about the number of page views and the number of unique visitors on each of the 1000 most popular websites in 2009 (Figure 2). The exposure of users to online ads has been evaluated extensively in [27] showing that $h = 58\%$ of the top websites host ads and that on average there are $a = 8$ ads per page. We estimate the total number of ads users see on the the top 1000 websites during a year (I_t) as done in previous work [10, 34]:

$$I_t = \sum_{i=1}^{1000} (\text{Page views on website } i) \cdot h \cdot a \qquad (1)$$

The data from *Compete.com* is aggregated over all the visitors of websites and does not give individual user browsing profiles. Thus, the average number of ads I_u a user sees on the the top 1000 websites during a year is:

$$I_u = \sum_{i=1}^{1000} \frac{\text{Page views on website } i}{\text{\# of visitors of website } i} \cdot h \cdot a \qquad (2)$$

We now compute the potential annual ad revenue R generated per user. To do so, we take into account that for a fraction ϕ of ads the ad network charges advertisers based on the number of *impressions* and for the remaining $1 - \phi$ based on *performance* (e.g., click-throughs) [17]:

$$R = \phi \cdot I_u \cdot CPI + (1 - \phi) \cdot I_u \cdot p \cdot CPC \qquad (3)$$

where CPI is the cost-per-impression, CPC is the cost-per-click and p is the probability that a click occurs on an ad (i.e., click-through rate). Due to the large number of ads, the cost-per-mille (CPM) representation is usually preferred for impression based ads (CPM=CPI$\cdot$1000). Both CPM and CPC depend on the type of ads and the hosting website. It is difficult to obtain a complete picture of CPMs and CPCs for the online advertising space, thus we rely on the average estimates reported in practice: CPM= \$2.39 [10] and CPC= \$0.5 [12]. The probability p that a click occurs on an ad is around 0.1% [18]. The CPM pricing model accounts for $\phi = 35\%$ of ad revenues and CPC for 65%, as reported in [17]. Based on expression (3), we estimate that the annual ad revenue generated on the top 1000 websites per user is $R = \$494$. The total ad revenue generated at the top 1000 websites is \$4.88 billion.

We differentiate between adversaries based on: (i) the number of users α the adversary $\mathcal{A}$ can affect and (ii) the resources $\mathcal{A}$ has to implement the attacks, which determines the fraction β of ad traffic (and consequently ad revenue) it can modify. The upper bound of estimated revenue ($R_{\mathcal{A}}$) $\mathcal{A}$ can gain by perpetrating attacks is:

$$R_{\mathcal{A}} = \alpha \cdot \beta \cdot R \qquad (4)$$

This model assumes that advertisers are willing to pay to $\mathcal{A}$ at most the same CPMs and CPCs as to the original ad network. We consider various values of α and β corresponding to different adversaries that appear in practice and derive the associated revenue gains in Table 1.

Table 1: $\mathcal{A}$'s potential annual revenue gain.

Adversary	α	β	$R_{\mathcal{A}}$ (in US \$)
Home wireless AP	[1, 10]	1	[494, 4.94K]
Hot Spot AP	[10, 100]	1	[4.94K, 49.4K]
Botnet	[1K, 100K]	1	[494K, 49.4M]
WSC	[10K, 2M]	1	[4.94M, 988M]
ISP	[50K, 15M]	1	[24.7M, 7.41B]

Note that these results are obtained from a sample of users visiting the top 1000 websites exclusively, and hence cannot be trivially generalized to the entire US population. Instead, our results measure the *economic incentive* of an adversary to tamper with the traffic of the users that access the top 1000 websites.

We consider a single compromised home wireless AP, a compromised hot spot AP, a network of compromised APs [33] or a botnet [11], a wireless social community (WSC) [8] and an ISP [16]. Figure 3 represents the estimated revenue $R_{\mathcal{A}}$ for the entire range of values $\beta \in (0, 1]$, considering the maximal value of α of each adversary from Table 1. The results show that even a small subset of routers controlled by an adversary can cause a significant loss of ad revenue for ad networks. Also, even by applying the attack on a small portion of traffic ($\beta = 0.1$), $\mathcal{A}$ can earn a significant revenue.

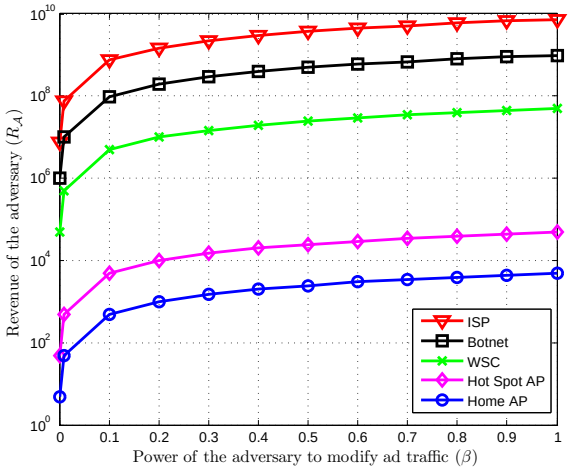

Figure 3: $\mathcal{A}$'s potential annual revenue gain (in US $).

We note that ISPs have a tremendous incentive to divert even a small fraction of the ad revenue. The total ad revenue in US in 2009 is $22.7 billion [17], meaning that the ad revenue per day is $62.2 million. Although some ISPs would not engage in such activities due to unforeseen legal consequences or the risk of damaging their reputation, reports [9] mention that such behavior is observed in practice in some countries. In [34], the authors use game theory to model ISPs' economic incentives to perform MitM attacks on ad systems and show that under certain conditions diverting revenue from online advertisements may maximize the revenue of a rational ISP.

4 Implementation of Attacks

In order to test the feasibility of the attacks, we implemented them on a wireless router. The attacks are executed locally on the wireless router in a transparent way to users. We used an Asus WL-500G Premium wireless router with 32Mb of memory and a 266Mhz processor. We uploaded an OpenWRT [4] firmware on the router as it provides many customization features. We used the latest compatible OpenWRT version, the Kamikaaze 8.09 with kernel 2.6.27.

The attacks rely on two main components: a transparent proxy (Squid v2.6) to parse HTTP traffic and executables to implement the attacks. With this setting, the attack is transparent as there is no change in the address bar of users' browsers.

The attacks consist of the following four steps:

(i) The router intercepts user generated HTTP traffic on port 80. The interception is done using NAT with a simple pre-routing rule.
(ii) The traffic is redirected to the transparent proxy (Squid) running on port 3128.
(iii) A C program called *redirector.c* analyzes all requested URLs. The redirector program detects matches with predefined rules (e.g., request to an ad server). If there is a match, the redirector program executes a PHP script implementing one attack depending on the matched rule. If there is no match, the redirector program outputs the original link and the proxy serves the original web page.
(iv) We use the built-in Busybox HTTPD server with PHP to generate web pages dynamically and forward them to the users.
Using this setting, we implemented three different attacks.

Pollution Attack We implemented the pollution attack using a similar script as in [35]. We add a javascript to every web page which creates an HTML frame. In our implementation, the HTML frame shows an EPFL logo.

Injecting Advertisements We considered two scenarios in this implementation: we change ad server URLs or change ad server javascripts.

To identify the URL of an AS, we use the list of known ASs from Firefox plugin AdBlock. The list consists of regular expressions where each expression represents a URL of an AS. When the redirector program analyzes URLs, it matches them with the URLs from the AdBlock list. In this implementation, if there is a match, the URL is locally replaced by the URL of the EPFL logo.

We replace Google ads with Yahoo! ads and vice versa by swapping the corresponding javascripts. To do so, we store both scripts at the router. If the requested URL corresponds to the Google (Yahoo!) JavaScript from one of the ad servers, the URL is redirected to a local path on the router to the stored Yahoo!'s (Google's) JavaScript.

Targeted Injection We implemented two targeted attacks: on search engine results (SERPs) and on location-based services (LBSs).

Our SERP attacks work with Google and Yahoo! search results. The attacking PHP script first downloads the original search result page based on users' query. Then, the received data is parsed searching for the unique sequence of characters in the HTML source code defining the beginning of the search results area. This sequence was identified by analyzing the HTML source code of the original SERP prior to implementation. We add a link to the EPFL website as a first search result for all search queries.

Our LBS attack targets users who are using Google Maps. Google Maps use AJAX technology in order to asynchronously communicate with the LBS server. This enables users to move around a map without refreshing the entire web page. The results of a user query are sent in the form of banners called *markers*. All the asynchronous information downloaded from the LBS server (i.e., maps and markers) are implemented in JavaScript. The attack intercepts the javascript and modifies it by injecting a forged marker: we advertise the same fake restaurant for all queried locations. The restaurant always appears as the first link in the results.

4.1 Evaluation of Attacks

Evaluation Metrics In order for the attacks to be transparent to users, the on-the-fly modifications performed by the router should not affect the load time of web pages. We consider two criteria.

First, we evaluate the *delay* added by attacks on the load time of web pages, i.e., the difference between the time to load a web page through the router performing on-the-fly modifications of ad traffic and the time to load the web page through a standard router.

Second, we evaluate the *scalability* of local on-the-fly modifications, i.e., the number of parallel requests that the router running the attacks can support compared to the number of parallel request a standard router can support. This is particularly relevant in a multi-user environment, where the router has to modify on-the-fly and in parallel the traffic of several users.

Evaluation Setup We measure the delay of loading times of three different types of web pages. Each web page triggers a different type of attack:

- www.20min.ch/ro/ : This is a Swiss newspaper web page. With this web page, we replace Google ads with Yahoo! ads.
- www.google.com/search?q=cars : This is a Google search for a keyword "cars". This web page triggers the targeted injection attack on SERPs.
- maps.google.com : This is an example of LBS website and this web page triggers the targeted injection attack on LBSs.

Each page is loaded with three different router settings: (i) the router without the proxy and the attacks, (ii) the router running the proxy but without the attacks and (iii) the router running the proxy and the attacks.

We wrote a Perl script that opens each page sequentially with Firefox. There is a 15 seconds pause between each load to ensure that web pages are completely loaded. Another Perl script parses the log files of the router proxy to compute the loading times of each page, based on the time of the first and the last request. In each scenario, web pages are loaded 15 times and we compute the average load time.

We evaluate the scalability of the attacks by measuring the maximum number of parallel requests that the router supports when running the proxy and the attacks.

We use a Perl script which generates a number of parallel wget requests to retrieve the content of web pages. We download a web page that corresponds to Google SERP with a query "cars", i.e., 'http://www.google.com/search?q=cars'. Note that to fully load this web page 11 GET requests are created. We increase the number of parallel requests and measure the average load time. In practice, we evaluate the average load time per request by parsing the log files of the router proxy.

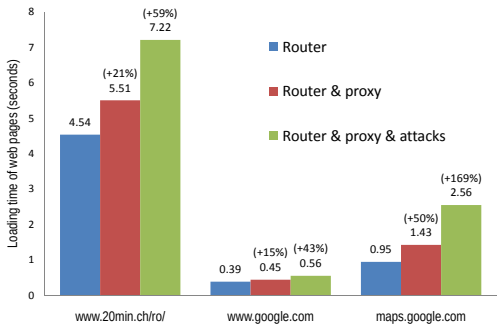

Figure 4: Web pages loading times.

Results We show in Figure 4 the average loading time of each web page when downloaded with different router settings. For each web page, three bars correspond to the loading times in the three scenarios. Delays are shown relative to the reference value with standard router settings. We observe two causes of delay: (i) the proxy at the router and (ii) the on-the-fly modifications by the attacks. The delay introduced by the proxy depends on the

type of elements in web pages and is more significant for web pages that have images. The delay introduced by on-the-fly modifications depends on the type of attack. Both, the proxy and on-the-fly modifications, cause network delay and affect the download time of web pages. The rendering time at the browser was almost constant and did not account for the difference in loading times.

By observing the loading time for growing number of parallel requests to Google search, we obtain that the router can withstand about 230 parallel connections, i.e., GET requests. If we increase the number of connections above 230, the router freezes. This result shows that the scale of the attacks depends on the type of websites. In case of a Google search, the router can modify the traffic of more than 20 users in parallel, however the attack may not scale well for websites like www.20min.ch that alone generate around 180 GET requests. The router supports a limited number of parallel connections because the proxy (Squid) uses a fair amount of memory. Out of the 32Mb available, only 6MB of memory are left once the router is running OpenWRT, a PHP client, Squid and the attacks implementations. Squid allocates memory for each parallel connection and when the available memory goes below 1MB, the router Freezes. A simple solution to improve the scalability of the attack consists in adding some memory to the router through the USB ports and allocate Squid's swap memory to it.

5 Countermeasures

The attacks described in this paper exploit vulnerabilities of the ad serving system in the communications: (i) between users and web servers and (ii) between users and ad servers. To protect against these attacks, we must guarantee the *authenticity* and *integrity* of both web pages and advertisements. To do so, communicating parties must derive security associations (SAs), i.e., establish shared security information between them to support secure communication. Note that confidentiality is not required to thwart the considered attacks.

In the following, we first explain the limitations of traditional approaches to derive SAs and then describe a new collaborative solution in Section 5.2.

5.1 Traditional Approaches

There are well-known protocols to establish SAs at different levels of the IP stack, such as Internet Protocol Security (IPSec) [22] or Transport Layer Security (TLS) [30].

IPSec The Internet Protocol Security (IPSec) protocol secures communications between users and IPSec servers at the *network* layer. IPSec is typically used by Virtual Private Networks (VPN), not web servers. Thus, IPSec does not provide end-to-end security between a user and a website, because an adversary may be located between IPSec servers and a website.

TLS The Transport Layer Security (TLS) protocol secures end-to-end communication at the *transport* layer. The secure version of HTTP, i.e., *HTTPS*, relies on TLS to secure sensitive browsing data. HTTPS is thus a straightforward solution to secure the ad serving system.

However, there are two problems with the large scale deployment of HTTPS: first, authentication issues when deploying HTTPS in practice, and second, HTTPS introduces a significant overhead.

Authentication Problem The TLS authentication procedure supposes that web servers prove their identity using a public/private key pair and a corresponding digital certificate. As there is no initial trust between a client and a server, independent trusted third parties (TTPs) verify the identity of servers and issue signed certificates proving the ownership of a given public key by a server. We refer to a TTP that issues certificates (e.g., $X.509$ certificates) as a Certification Authority (CA). The certificate of each CA (i.e., a root certificate) is preloaded into web browsers by software vendors. Users can then verify transparently the validity of servers' certificates using trust chains.

Previous work exposed several issues with such authentication procedures [30, 31]. For example, malicious web servers can downgrade security parameters during the connection establishment [30] or governments can force local CAs to issue bad certificates [31]. In these scenarios, authentication fails and data integrity is not guaranteed.

The most prevalent authentication issue with HTTPS is that most websites cannot afford certificates signed by a trusted CA, and prefer instead to provide their own untrusted certificates. Certificates signed by CAs tend to be used mostly by large and profitable websites. Other websites favor the use of *self-signed certificates*: a web server signs its own certificate with its private key. Thus, users must trust websites directly. A number of surveys on users' browsing habits [23] show that the vast majority of users do not understand the concept and accept self-signed certificates without proper verification. If users accept invalid self-signed certificates, they may establish SAs with malicious websitess and thus be victims of man-in-the-middle attacks.

To help users properly verify self-signed certificates, the authors of [36] suggest that *network notaries* collect server's public keys over time in a public database. When a client receives a self-signed certificate and its corresponding public key from a server, it contacts the notaries to obtain previous public keys used by that server. This

additional information helps users make better security decisions. However, this solution has several drawbacks. First, any independent entity can propose to install and maintain a notary server. Hence, the solution may fail to protect against MitM attacks as some notaries may be malicious [36]. Second, it takes some time to build trustworthy records before the service becomes reliable.

Overhead HTTPS introduces a significant communication and computation overhead. The major part of the overhead is due to the initial key exchange [19]. In the case of web browsing, sessions tend to be short and frequent. Hence, the initial key exchange overhead relative to the session duration will be high. As investigated in [19, 29], the throughput of an HTTPS server can be significantly lower than the throughput of an HTTP server. Because confidentiality is not required, we may configure HTTPS to only verify data integrity and authentication. However, the initial key exchange is still required and the overhead remains significant.

5.2 Collaborative Approach to Securing Online Advertising

We propose a novel solution where the WS hosting advertisements collaborate with the AS to build a secure ad serving system. We design a collaborative secure protocol for ad serving that leverages on: (i) the existing trust in ASs (based on their valid certificates), (ii) the business relationships between ASs and associated WSs that host ASs' ads and (iii) economic incentives of ad networks to protect their ad revenue.

We assume that ASs own valid certificates as they typically belong to major companies that already own valid certificates. In contrast, WSs may not always have the means to acquire a certificate from a CA. The game theoretic analysis in [34] shows that when facing the threats of MitM attacks discussed in Section 3, the AS has economic incentives to help affiliated WSs secure their communications. Hence, we suggest to leverage on ASs to help users properly authenticate WSs and protect the integrity of their communications. The collaboration benefits both entities: first, WSs do not have to acquire valid certificates and can rely on the AS's valid certificates; second, the secure communications between WSs and users guarantee that ASs protect their ad revenue from MitM attacks. By protecting their own interests, ad networks and websites indirectly provide secure communications to web users, giving incentives to users to adopt this mechanism as well.

We propose two versions of a collaborative secure protocol, that we call Data Integrity in Advertising Services Protocol (**DIASP**).

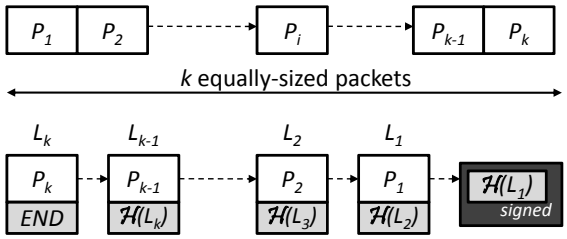

Figure 5: Authenticated hash-chain generation.

5.2.1 DIASP Primitive

There are multiple primitives to protect the authenticity and integrity of communications. A computationally efficient method consists in computing the hash of a web page and signing it. However, the browser cannot start rendering the page before downloading the entire content and verifying the signature. In this work, we use light-weight authenticated hash-chains [20] that enable the real-time rendering of web pages. In [24] hash-chains are used to solve the impossibility of proxy caching web pages with HTTPS.

Authenticated hash-chains Authenticated hash-chains (AHs) protect the integrity of a message m by computing the hash of many subparts of the message rather than the hash of the entire message at once. First, the content of a web page is split into k equally sized packets, $P_1, ..., P_k$. An *END* tag is concatenated to the last packet P_k in order to mark the end of the hash-chain, $L_k = P_k||END$. Second, consider a one-way Hash function $\mathcal{H}$ (e.g., SHA-1). Each packet is concatenated with the hash of n previous packets as shown in Figure 5. In this example, we consider $n = 1$ to minimize the size of packets. The WS computes $\mathcal{H}(L_k)$ and concatenates it with the previous packet $L_{k-1} = P_{k-1}||\mathcal{H}(L_k)$. Hence, if the integrity of P_{k-1} is properly verified, the integrity of P_k is verified as well. The WS can thus repeat this operation to create a hash-chain. Finally, the integrity of the entire chain depends on the integrity of the first packet. This is typically guaranteed by signing the first packet. In this way, a WS can compute $AH(m)$ and guarantee the integrity of a message m. To verify the integrity of a message, the user only needs to verify the signature of the first packet ($H(L_1)$). The following packets are verified based on the hash in the corresponding previous packet.

5.2.2 DIASP v.1

DIASP v.1 creates two hash-chains: one for the web page content $AH(p)$ and one for the ads $AH(a)$. In both hash-chains, the first element (i.e., $H(L_1)$ and $H(L'_1)$)

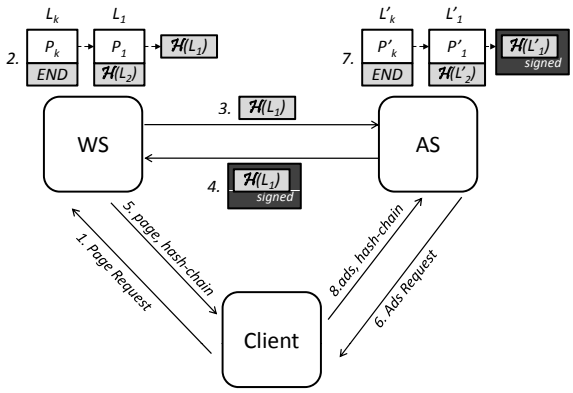

Figure 6: Communication schema of DIASP v.1.

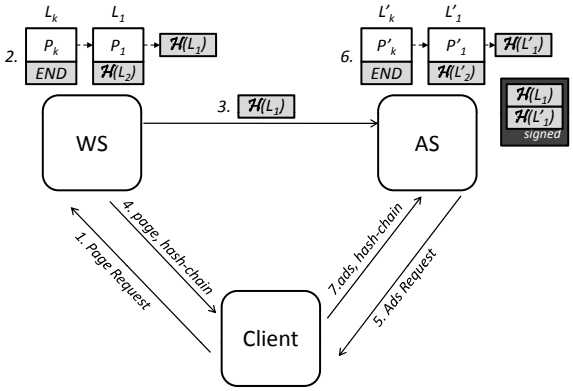

Figure 7: Communication schema of DIASP v.2.

is signed with the AS's private key.

We detail the execution of DIASP v.1 in Figure 6 and summarize it as follows:

1. $U \rightarrow WS$: GET URL_{WS}
2. WS: Computes $AH(p)$
3. $WS \xrightarrow{s} AS$: $\mathcal{H}(L_1)$
4. $AS \xrightarrow{s} WS$: $\sigma_{AS}(\mathcal{H}(L_1)))$
5. $WS \rightarrow U$: p, $AH(p)$ with $\sigma_{AS}(\mathcal{H}(L_1))$
6. $U \rightarrow AS$: GET URL_{AS}, WS_{ID}
7. AS: Computes $AH(a), \sigma_{AS}(\mathcal{H}(L'_1))$
8. $AS \rightarrow U$: a, $AH(a)$ with $\sigma_{AS}(\mathcal{H}(L'_1))$

where $\xrightarrow{s}$ means that communications are over HTTPS and $\sigma_{AS}(m)$ is the digital signature of a message m by the AS. The WS authenticates the AS based on AS's valid certificate and the AS can authenticate the WS based on some secret information that is established during the registration process of the WS to host AS's ads. Users can check the integrity of the first packet and start to dynamically render a web page as soon as they receive packets from the hash-chain. This solution also allows users to independently check the integrity of the two communication channels.

However, DIASP v.1 has two main drawbacks: (i) it requires two signatures per web page hosting ads and thus creates additional computation overhead; (ii) it is incompatible with the current implementation of browsers as it uses cross domain signatures, i.e., the hash-chain $AH(p)$ is downloaded from URL_{WS} but is signed by the AS. The certificate of the AS corresponds to a different domain name than the WS. Hence, browsers might warn users about the domain mismatch. A potential solution (requiring changes in browsers' implementation) is to help browsers differentiate between valid (WSs and ASs using DIASP v.1) and invalid (MitM attacks) mismatches. To do so, web browsers could maintain a white list of valid WS-AS associations.

5.2.3 DIASP v.2

We propose a second version of DIASP that bypasses the drawbacks of DIASP v.1. DIASP v.2 concatenates the first packet of hash-chains $AH(p)$ and $AH(a)$ to create a single element $\mathcal{H}(L_1)||\mathcal{H}(L'_1)$. The AS only needs to sign this element to authenticate both hash-chains. We detail DIASP v.2 in Figure 7 and summarize it as follows:

1. $U \rightarrow WS$: GET URL_{WS}
2. WS: Computes $AH(p)$
3. $WS \xrightarrow{s} AS$: $\mathcal{H}(L_1)$
4. $WS \rightarrow U$: p, $AH(p)$
5. $U \rightarrow AS$: GET URL_{AS}, WS_{ID}
6. AS: Computes $AH(a), \sigma_{AS}(\mathcal{H}(L_1)||\mathcal{H}(L'_1))$
7. $AS \rightarrow U$: a, $AH(a)$ with $\sigma_{AS}(\mathcal{H}(L_1)||\mathcal{H}(L'_1))$

DIASP v.2 solves the problem of domain mismatch and only requires one digital signature per web page hosting ads. Still, users cannot verify the integrity of a web page before receiving the signed elements from the AS (step 7). This may add some delay in rendering web pages. We note that browsers can make several requests in parallel and that WS can reduce this potential latency by placing the links to ASs at the beginning of the HTML page. In addition, measurements from [27] indicate that it is the number and size of ad objects that increase the download time of a web page and not the latency of communications.

5.3 Discussion

We discuss the implementation of DIASP in practice.

Type of the Web content There are three main types of content on the Web: static, dynamic and personalized content.

In the case of static content, the web server computes the hash-chain (step 2) and communicates with the AS (step 3) only once and then the same hash-chain can be served to all visitors of the website.

In the case of dynamic content (e.g., blog, newspapers), the web server computes the hash-chain (step 2) and communicates with the AS (step 3) each time the page is updated with new content. Between the updates, the WS can serve the same hash-chain to all visitors.

In the case of personalized content (e.g., Facebook), the WS serves different pages to different users, an additional mechanism is needed to link hash-chains to corresponding pages and visitors. To do so, the WS assigns a randomly chosen unique number ID to each user request (step 1). Since the WS serves a personalized page p (consequently, a different hash-chain) to each user the WS has to communicate to the AS (step 3) the ID together with the first element (i.e., $\mathcal{H}(L_1)$) of the associated hash-chain, such that the AS can link the hash-chains it signs with the corresponding users' requests for ads. The AS keeps all $\sigma_{AS}(H(L_1), H(L'_1))$ associated with IDs until it receives a request from a user (step 5) with an ID that matches one of the records. After step 7, the AS deletes the record with this ID. To protect user privacy, IDs are changed at every interaction with WS and AS.

Usability of DIASP DIASP may affect user browsing experience when an integrity check fails (i.e., ads or web pages have been altered). We envision two possible policies: first, the elements that failed the integrity check are not displayed; second, users are warned by browsers and make a decision.

If ads are not displayed, the ad network loses ad revenue anyhow since the ads have been tampered with, but the adversary modifying the traffic also does not earn any revenue. This diminishes the adversary's incentives to mount MitM attacks. In addition, this protects ASs' and WSs' reputation from the injection of inappropriate content and users from the injection of malicious ads.

If users are prompted, users have to interpret the warning messages before making a decision. DIASP could even issue warnings specifying whether the content or ads has been tampered with. It is out of the scope of this work to determine which policy should be favored.

Bootstrapping DIASP DIASP cannot be used with all HTTP communications as not all websites host ads and have an association with an AS with a valid certificate. Hence, browsers must be able to determine which protocol to use (DIASP or HTTP) to communicate with a given website.

One approach consists in maintaining a white list of all websites that use DIASP. Hence, before communicating with a website, a browser first checks if the website is white listed and requires to run DIASP. Such white lists can be maintained by leveraging on existing databases that provide black lists of potential phishing websites. Such databases are updated by major companies (e.g., Google or Yahoo) and most browsers are already configured to check them before communicating with websites.

Another approach is to specify in DNS records the use of DIASP. For instance, DNS replies would specify in addition to the IP address of a website whether it uses DIASP.

5.4 Evaluation of DIASP

In this section, we compare the performance of DIASP with HTTPS in terms of web page loading time. We set up a localhost server with both HTTP and HTTPS protocols and use the Apache benchmark software [5] to measure loading times. We estimate the performance of HTTPS by measuring the loading time of web pages using HTTPS (without encryption). Similarly, we estimate the performance of DIASP by measuring the loading time of the same web pages using HTTP and adding the computation times of hash and signature functions (Table 2) found in [7].

According to the measurements of [27], the average size of a web page with ads in the *.com* domain is 301KB, out of which 51KB are for ads. Thus, we estimate the loading time of a 250KB content from a web server and 51KB of ads from an ad server.

Table 2: Performance of different functions.

Hash functions		Signature functions	
Algorithm	**MB/s**	**Algorithm**	**ms/operation**
SHA-1	160	RSA 1024 Signature	1.48
SHA-256	116	RSA 1024 Verification	0.07
SHA-512	103	RSA 2048 Signature	6.05
		RSA 2048 Verification	0.16

We load the same web page 1000 times and obtain that the average loading time of the content (respectively, ads) with HTTPS is 46.19ms (40.54ms) and only 0.72ms (0.34ms) with HTTP. As the transmission time is equal with both HTTPS and HTTP (i.e., we run the server locally), the difference is caused by the HTTPS handshake which is expensive in terms of computation and communication overhead. We estimate the total loading times using a conservative approach by assuming that communications between WS and AS are sequential (whereas in practice, web browsers can make parallel requests). The total loading time of a web page with ads over HTTPS is: 46.19ms +40.54ms=86.73ms.

The total loading time with DIASP v.1 is: $p+AH(p)+\sigma+V(\sigma)+V(AH(p))+a+AH(a)+\sigma+V(\sigma)+V(AH(a)) == (0.72+2.15+1.48+0.07+2.15+0.34+0.44+1.48+0.07+0.44)ms = 9.34ms$ where $V()$ corresponds to the verification of a signature or a hash-chain. The total loading time with DIASP v.2 is:

$p + AH(p) + a + AH(a) + \sigma + V(\sigma) + V(AH(p)) + V(AH(a)) = 7.79ms$.

The results show that DIASP v.1 is ≈ 9 times faster than HTTPS and DIASP v.2 is ≈ 11 times faster. Note that we do not consider the use of HTTPS accelerator. In the case of larger file sizes (> 301KB), the overhead introduced by HTTPS handshake becomes less significant. Still, based on preliminary estimates DIASP reduces the loading time of web pages compared to HTTPS.

6 Conclusion

With the attacks presented in this paper, we have shown the tremendous power of the access network and the impact it can have on the ad serving system. This impact can translate into revenue loss for the advertisers, ANs and websites and into lower security for the users. These threats create incentives for all the entities to deploy the proposed collaborative scheme to secure the ad serving system, and protect the revenues and Web browsing. We have shown that providing authentication and data integrity is necessary, for the security of both, the content of web pages and the ads themselves. Thus, ad serving would not only finance the Internet but would also fuel the deployment of online security.

7 Acknowledgments

We would like to thank Maxim Raya, Mark Felegyhazi, Anthony Durussel, Anjan Som and Raoul Neu for their insights and suggestions on earlier versions of this work, and the anonymous reviewers for their helpful feedback.

8 Availability

The code of the implemented attacks is available at http://icapeople.epfl.ch/freudiger/alterads/

References

[1] http://www.perftech.com.

[2] http://www.phorm.com.

[3] http://adblockplus.org.

[4] http://openwrt.org.

[5] ab - apache http server benchmarking tool.

[6] Botnet caught red handed stealing from Google. http://www.theregister.co.uk/2009/10/09/bahama_botnet_steals_from_google.

[7] Crypto++ 5.6.0 benchmarks. www.cryptopp.com/benchmarks.html.

[8] Fon. www.fon.com.

[9] Growing number of ISPs injecting own content into websites. http://www.techdirt.com/articles/20080417/041032874.shtml.

[10] How much ads cost. www.emarketer.com/Article.aspx?R=1007053.

[11] Network stealth router-based botnet. dronebl.org/blog/8.

[12] U.S. average CPC by category, June 2009. http://www.clickz.com/3634374.

[13] Viral Web infection siphons ad dollars from Google. www.theregister.co.uk/2009/05/14/viral_web_infection/.

[14] Click Fraud Index. *ClickForennsics Inc.* (2009).

[15] 'psyb0t' worm infects Linksys, Netgear home routers, modems. *ZDNet* (2009).

[16] Comcast reports fourth quarter and year end 2009 results. *Comcast Corporation* (2010).

[17] IAB Internet Advertising Revenue Report, 2009 full-year results. *Interactive Advertising Bureau* (2010).

[18] 2008 Year-in-Review Benchmarks. *DoubleClick Research Report* (June, 2009).

[19] COARFA, C., DRUSCHEL, P., AND WALLACH, D. Performance analysis of tls web server. In *TOCS* (2006).

[20] COPPERSMITH, D., AND JAKOBSSON, M. Almost optimal hash sequence traversal.

[21] DHARMAPURIKAR, S., KRISHNAMURTHY, P., SPROULL, T., AND LOCKWOOD, J. W. Dpi using parallel bloom filters. In *IEEE Micro* (2004).

[22] DORASWAMY, N., AND HARKINS, D. *IPSec: The New Security Standard for the Internet, Intranets, and Virtual Private Networks*. Prentice Hall PTR, USA, 1999.

[23] FRIEDMAN, B., HURLEY, D., HOWE, D. C., FELTEN, E., AND NISSENBAUM, H. Users' conceptions of web security: a comparative study.

[24] GASPARD, C., GOLDBERG, S., ITANI, W., BERTINO, E., AND NITA-ROTARU, C. Sine: Cache-friendly integrity for the web. *ICNP 2009* (July 2009).

[25] HEARNE, R. Click through rate of google search results. http://www.redcardinal.ie/search-engine-optimisation/12-08-2006/clickthrough-analysis-of-aol-datatgz/.

[26] KRISHNAMURTHY, B., MALANDRINO, D., AND WILLS, C. E. Measuring privacy loss and the impact of privacy protection in web browsing. In *SOUPS* (2007).

[27] KRISHNAMURTHY, B., AND WILLS, C. Cat and mouse: Content delivery tradeoffs in web access. In *WWW* (2006).

[28] LANGLEY, A. Opportunistic encryption everywhere. In *W2SP* (2009).

[29] REIS, C., GRIBBLE, S. D., KOHNO, T., AND WEAVER, N. C. Detecting in-flight page changes with web tripwires. In *NSDI* (2008).

[30] RESCORLA, E. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.

[31] SOGHOIAN, C., AND STAMM, S. Certified lies: Detecting and defeating government interception attacks against ssl. *Available at SSRN* (2010).

[32] THEFREESITE.COM. Free internet access providers. http://www.thefreesite.com/Free_Internet_Access.

[33] TSOW, A., JAKOBSSON, M., YANG, L., AND WETZEL, S. Warkitting: the drive-by subversion of wireless home routers. *Journal of Digital Forensic Practice 1*, 3 (2006).

[34] VRATONJIC, N., RAYA, M., HUBAUX, J.-P., AND PARKES, D. Security Games in Online Advertising: Can Ads Help Secure the Web? In *WEIS* (2010).

[35] WEINSTEIN, L. Google hijacked – major ISP to intercept and modify web pages. http://lauren.vortex.com.

[36] WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Perspectives: Improving ssh-style host authentication with multi-path probing. In *USENIX ATC* (2008).