An Effective Method for Combating Malicious Scripts Clickbots

Yanlin Peng, Linfeng Zhang, J. Morris Chang, and Yong Guan

Iowa State University, Ames IA 50011, USA
{kitap,zhanglf,morris,guan}@iastate.edu

Abstract. Online advertising has been suffering serious click fraud problem. Fraudulent publishers can generate false clicks using malicious scripts embedded in their web pages. Even widely-used security techniques like `iframe` cannot prevent such attack. In this paper, we propose a framework and associated methodologies to automatically and quickly detect and filter false clicks generated by malicious scripts. We propose to create an impression-click identifier which is able to link corresponding impressions and clicks together with a predefined lifetime. The impression-click identifiers are stored in a special data structure and can be later validated upon a click is received. The framework has the nice features of constant-time inserting and querying, low false positive rate and low quantifiable false negative rate. From our experimental evaluation on a primitive PC machine, our approach can achieve a false negative rate 0.00008 using 120MB memory and average inserting and querying time is 3 and 1 microseconds, respectively.

Keywords: Online Advertising Networks, Click Fraud, Network Forensics, Attack Detection.

1 Introduction

Recent-year rapid development of the Internet has led to a new, billion-dollar online advertising market. Using new web technologies, online advertising has many appealing features. Firstly, online adverting has the capability to target potential customers more quickly and more accurately than traditional broadcast advertisements, which potentially improves return on investment (ROI). Besides, direct response from potential customers is available, thus the performance of advertising campaigns can be tracked more easily. Online advertising also requires much fewer efforts and costs to set up and maintain. Hence, more and more companies have invested on online advertising campaigns. In 2008, online advertising revenues in the United States totaled $23.4 billion, with a 10.6 percent increase from 2007 [1].

Online advertising typically involves three parties: advertisers, publishers and syndicators. An advertisers provides advertisement (we use *ad* for short) information and pays for advertising. A publisher displays ads on her web sites and gets paid. A syndicator acts as a commissioner who gets ads from advertisers and

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 523–538, 2009.
© Springer-Verlag Berlin Heidelberg 2009

distributes them to publishers, and earns commission fees. Some large publishers, e.g. ESPN.com, have their own advertising system and deal with advertisers directly. But many small advertisers and small publishers depend on syndicator's professional service for advertising and billing.

Advertisers may be charged per thousand displays of ads (pay per mille, PPM), per click on ads (pay per click, PPC), or per conversional action (pay per action, PPA). Of course, advertisers would prefer paying according to sales by using PPA model. But publishers would prefer paying according to their traffic load by using PPM model. As the result of balancing risks between advertisers and publishers, the PPC model has been the most prevalent payment model in the online advertising market [2].

However, PPC model has been suffering serious click fraud problem. Click fraud is a type of Internet crime that occurs in online advertisement models when an ad is being clicked for the purpose of generating a charge without having actual interest in the target of the ad's link. Typically, two types of motivations are behind click frauds. Malicious advertisers may click on competitors' ads in order to increase their advertising expense. Since current advertising systems usually use auction scheme, such attack may deplete competitors' daily advertising budgets and remove them from the competing list. Fraudulent publishers often inflate the number of clicks on ads displaying on their own web sites in order to get more commissions. A survey indicates that honest Internet advertisers paid $1.3 billion for click fraud in 2006 [3]. The overall industry average click fraud rate for Q4 2008 is estimated at 17.1% [4]. Because of large number of fraudulent clicks, some syndicator companies (e.g. Google and Yahoo!) have been facing lawsuits recently [5,6]. Hence, preventing click fraud is a critical task to keep the healthiness of the online advertising market.

Fraudulent clicks could be generated by different entities using different techniques. Human, such as cheap labors, could generate fraudulent clicks manually. Clickbots [7] could generate automatic and large amount of fraudulent clicks quickly. A clickbot can be a special program on a virus/Trojan infected computer or a malicious script embedded in a publisher's web page. The latter one does not even require breaking into someone's computers. Whenever an innocent user visits the web site, the malicious script, which exploits vulnerabilities of online advertising models, is executed in the visitor's browser and may click ads automatically and stealthily. An experiment using malicious scripts had been conducted and cumulated thousands of dollars in the publisher's account [8]. In this paper, we focus on fraudulent clicks generated by such malicious scripts.

Several existing solutions have the capabilities to address some types of fraudulent clicks. However, none of them is able to prevent fraudulent clicks generated by malicious scripts as effective as the solution proposed in this paper.

Anomaly-based methods are industry-wide solutions to detect fraudulent clicks by detecting abnormal features in clicking streams. As Tuzhilin, Daswani et al. discussed in [9, 10, 11], fraudulent clicks, whether committed by human beings or bots, will show anomalies if enough data are collected. For example, duplicate click is one well-known anomaly, which indicates that clicks with the

same identifier appearing within a short time period are likely to be fraudulent clicks. Efficient algorithms for detecting duplicate clicks are proposed by Metwally et al. in [12] and Zhang et al. in [13]. In online advertising systems, a number of such online or offline filters are applied to identify anomalies. These filters are trade secrets, hence the details are not disclosed. The primary limitation for anomaly-based detection is the data limitation. When too little data are available, it may be hard to identify anomalies. Another limitation is the hardness to distinguish meaningless (but non-fraudulent) clicks from fraudulent clicks. That's why syndicators such as Google claim that they detect *invalid* clicks.

Another solution proposed by Juels et al. tries to authenticate valid clicks. In [14], they propose a credential-based approach to identify *premium clicks* (i.e. *good* clicks) instead of excluding invalid clicks. If a user has committed legitimate behaviors (e.g. purchases), the clicks from her browser are marked as premium clicks and cryptographic credentials are stored in the browsing cache for authentication. This approach, however, is still subject to the attack presented in this paper, where click fraud may be committed in a browser used by a legitimate user. If credentials have been stored due to the legitimate behaviors from that user, fraudulent clicks will also be identified as premium clicks.

As the carrier of ads, the security of the advertising client is also very important. Many syndicators, like Google and Yahoo!, have wrapped their ads by `iframe`s and utilize the same-origin-policy to protect their advertising clients [15, 11]. Another approach to protect advertising client is to use spiders to visit publisher's web sites and try to discover misuse of advertising clients [15]. However, both approaches could be circumvented by malicious publishers, which will be further discussed in Section 2.

In this paper, we propose a framework and associated methodologies to detect and prevent fraudulent clicks that are generated by malicious scripts embedded in fraudulent publisher's web sites We propose to create an one-time impression-click identifier with a predefined lifetime for each impression. At the syndicator's server, the impression-click identifiers are stored in a special data structure and are later validated against received clicks. Compared to naïve data structures (e.g. linked list) which result in high costs to store and query items, the proposed data structure has the characteristics of constant-time query, low memory space requirement, low false negative, and low false positive. Compared to general Bloom Filters [17], the proposed data structure has the capability of automatically deleting the outdated identifiers and that have been clicked. Thus, the proposed framework can be used to detect click fraud effectively.

Click fraud detection may be performed using online or offline filters [9]. However, offline detections are often used to detect sophisticated click frauds which will appear only after some sort of data integration and are hard to be detected at runtime. On the contrary, simple and fast detections are more preferable to be implemented as online filters to filter invalid clicks quickly. Since the framework proposed in this paper can be executed efficiently, we propose to apply the detection method presented in this paper at runtime, Using a primitive PC machine

to process $3,328,587$ impressions and $277,633$ clicks, our approach achieved a false negative rate 0.00008 and average 3 microseconds for inserting an identifier, average 1 microsecond for validating an identifier.

Contributions of this research: (1) We propose a framework which has the capability to correlate *genuine* impressions and clicks thus prevents the fraudulent clicks that are generated by malicious scripts embedded in publisher's web pages. (2) The proposed framework has the capability of automatically deleting the outdated identifiers and the identifiers that have been clicked. (3) The proposed framework can achieve constant processing time, low false negative and low false positive.

Note that the solution proposed in this paper does not mean to be a complete solution for all types of click frauds. Rather, it provides client-side and server-side methods to prevent a type of click fraud that is committed by sophisticated malicious scripts in publisher's web pages. This solution can be seamlessly combined with other click fraud detection methods to provide better protection.

In this paper, we discuss the scenario that the publisher and the syndicator are from different origins only. In case that they are from the same origin, the publisher does not have the motivation to exploit the advertising clients.

The rest of the paper is organized as follows. We describe the malicious-script generated click fraud and define the problem in Section 2. In Section 3, we propose and analyze a framework to address the problem. Experimental results are discussed in Section 4. We conclude our paper in Section 5.

2 Problem Definition

In this section, we firstly present a general framework of online advertising. Then, we discuss how malicious scripts can be used to launch click fraud attacks even though `iframe` has been used. At the end of the section, we specify the objectives of this research.

2.1 A Framework for Advertising Networks

In general, a typical advertising network involves three parties: advertisers, syndicators and publishers. A *visitor* interacts with all of them. A visitor is an information consumer who visits web sites via a browser and may click on interested ads. In ad networks, visitors are the targets of advertising and visitor's browsers transfer ad handling messages between publishers, syndicators and advertisers.

Figure 1 shows a typical ad network working process (we call it *Ad Handling Process*) consisting of ten steps. In the following description, we assume that ads are wrapped with `iframe`s, which is a widely-adopted security technique to protected advertising clients. We provide a pseudo form of the messages that are exchanged between the visitor V, the publisher P, the syndicator S and the advertiser A at each step, and provide a corresponding brief description. In the description, $HTTPreq$ denotes an HTTP request and $HTTPresp$ denotes an HTTP response.

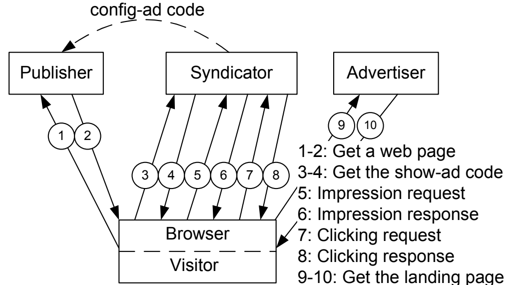


Fig. 1. A framework for advertising networks

Step 1: $V \rightarrow P : HTTPreq\{URL_{pub}\}$. A visitor requests a publisher's web page at URL_{pub} via her browser.

Step 2: $P \rightarrow V : HTTPresp\{Page_{pub}, Code_{conf}\}$. The publisher's web server sends back the content of the web page $Page_{pub}$, with the embedded config-ad code $Code_{conf}$. We call $Page_{pub}$ as a *referring page*, since it may refer the visitor to an advertiser's web site. The config-ad code contains configuration information about the publisher and a link URL_{show} to a show-ad code on the syndicator's server.

Step 3: $V \rightarrow S : HTTPreq\{URL_{show}\}$. The visitor's browser requests the show-ad code from the syndicator's server at URL_{show}.

Step 4: $P \rightarrow V : HTTPresp\{Code_{show}\}$. The syndicator's server returns the show-ad code $Code_{show}$, a snippet of script code, whose primary task is to construct an `iframe` which points to the real ad page URL_{imp}. For example, URL_{imp} may be like `http://syndicator.com/ads?client=publisher-id&referrer=http://publisher.com/`.The `iframe` may look like `<iframe src="`URL_{imp}`" id="ads_frame"></iframe>`.

Step 5: $V \rightarrow S : HTTPreq\{URL_{imp}\}$. The visitor's browser sends an HTTP request for the ad page to the syndicator's server at URL_{imp} (*impression request*).

Step 6: $S \rightarrow V : HTTPresp\{Page_{imp}\}$. The syndicator's server composes and returns an HTML document (*impression response*). The HTML document contains the descriptions and links for ads.

Step 7: $V \rightarrow S : HTTPreq\{URL_{click}\}$. If the visitor clicks an ad, an HTTP request is sent to the syndicator's server at URL_{click} (*click request*). The important parameters, such as the publisher's client ID and the URL of the advertiser's landing page, are embedded as parameters of URL_{click}. For example, URL_{click} may look like `http://syndicator.com/click?client=publisher-id&adurl=http://advertiser.com/&referrer=http://publisher.com/`.

Step 8: $S \rightarrow V : HTTPresp\{URL_{ad}\}$. The syndicator validates the click. If valid, the syndicator charges the advertiser and pays the publisher. Otherwise, the advertiser is not charged for an invalid click. For both validation results, the same HTTP response containing the URL of the advertiser's landing page URL_{ad} will be sent back (*click response*). The

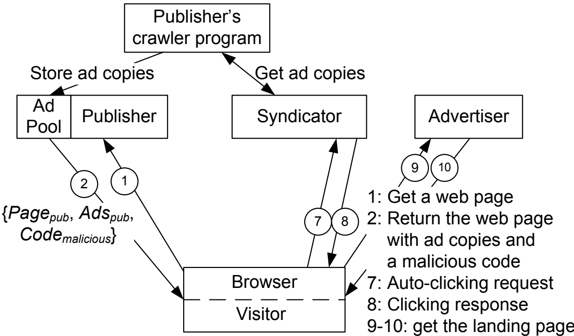


Fig. 2. A framework for malicious-script-generating click fraud

syndicator purposely makes no difference between valid response and invalid response to prevent attackers probing the click validation scheme.

Step 9: $V \rightarrow A$: $HTTPreq\{URL_{ad}\}$. Following the response in *Step* 8, the visitor's browser sends an HTTP request to advertiser's server at URL_{ad}.

Step 10: $A \rightarrow V$: $HTTPresp\{Page_{ad}\}$. The advertiser's server returns the landing page.

2.2 Threat Model

Many syndicators use `iframe` to wrap and protect their advertising clients [15, 11]. Using `iframe`, the same-origin-policy, which is enforced in all modern browsers, will prevent the script from one origin to read and change the web content from a different origin. The origin is defined by the protocol, port and host fields of a URL [18]. Since the publisher's web sites and the syndicator's web server are from different origins, the scripts on the publisher's web sites cannot click ads in the `iframe`. However, same-origin-policy can be circumvented. In this section, we present an attack to circumvent the same-origin policy. Such attack has been proved to be effective by the Think Digit Magazine [8].

This type of attack is launched by fraudulent publishers. As shown in Figure 2, before launching attacks, the publisher uses a crawler program to visit her own web site and downloads ads. The publisher may run this program iteratively and store all available ad copies into an ad pool on her web server and is ready for attacks. Compared with the typical ad handling process in Figure 1, the attack has the following different processing steps:

Step 2: After receiving an HTTP request from a visitor, the publisher's server returns $HTTPresp\{Page_{pub}, Ads_{pub}, Code_{mal}\}$, where additional Ads_{pub} are the ad copies selected from the publisher's ad pool, additional $Code_{mal}$ is a malicious script to generate automatic clicks on Ads_{pub}. Note that $Code_{conf}$ in the normal process is missing.

Step 3-6: These steps are skipped because $Code_{conf}$ is missing.

Step 7: The malicious code generates an automatic click on an ad copy in Ads_{pub}. Note that the ad copies in Ads_{pub} and the malicious code are

from the same origin – the publisher, hence the automatic false click will be generated successfully.

There is an obvious shortcoming of this attack: Steps 3-6 of the normal ad handling process are missing. Hence the syndicator's server can detect this attack simply by checking whether a corresponding impression request is received before a click request. A smarter script will download the genuine ads and provide fake ads at the same time. Specifically, the config-ad code $Code_{conf}$ is still embedded in the response of Step 2. Now, every click seems having a corresponding impression. Without specially designed mechanisms, it is hard for the syndicator to distinguish such false clicks from the genuine clicks.

The smarter script has a challenge to guess which ads are returned by a syndicator in the `iframe` so that it can click on the same copy from the publisher's ad pool. The challenge occurs because the publisher's script cannot read the content within an `iframe` and the ads in the `iframe` are often displayed dynamically due to the auction scheme. However, the smarter script still has good chances to guess by using special techniques, such as applying careful design to reduce the number of available ads for a web page or sending multiple impression requests for one visit.

2.3 Naïve Solutions

PPA model could be used to address general click fraud problem. However, PPA model is less preferred by publishers, since each display of ads will increase the traffic load of the publisher's web site and publishers take the risk that visitors do not convert on their web sites.

A syndicator can also place a nonce into browser cookies each time an ad is requested, then check that nonce when a click request is received. The problem of this solution is that users may not click on ads right away and browser cookies may be deleted before clicking ads. For example, Firefox has an option to let a user delete cookies when closing the browser. Thus, a valid click may be sent without a cookie. Deleting cookies is not unusual among users. A study of 2,337 users found that 10 percent of the users has the habit to delete cookies daily and more delete cookies in a longer regular period [19]. If a click without a cookie is not counted as valid, publishers are not fairly paid.

Another possible solution is to encode a time window, an IP address (or a cookie) and other related information into the clicking URL, using a secret key known by the syndicator only [20]. The encoded information is checked when a click request is received. The problem is similar: users may change IP addresses or delete cookies before clicking ads, thus validation of encoded information will fail. There are considerably many scenarios that IP address will changed, such as a user in a DHCP domain or roaming to a different subnet. If we classify those clicks as invalid, publishers are not fairly paid.

A syndicator may have human investigators to check publisher's website for misusing of advertising clients and malicious scripts. If malicious scripts that manipulate ads are found, the publisher's account will be suspended. However, manual investigation is impossible to monitor all publisher's websites when the publisher network becomes large. Hence, automatic spidering programs are often

used to investigate publisher's website. However, a cloaking-type attack can circumvent the spidering investigation effectively [15]. In the attack, the publisher serves a bad version with malicious scripts to normal visitors and a good version with benign scripts to the advertising system's spiders. A hidden forwarder is used to distinguish normal users from investigation spiders. The forwarder's URL is distributed to normal visitors via methods like spam email and redirect them to real publisher's websites. The publisher checks the `referer` field in the HTTP header to distinguish normal visitors from investigation spiders. For visits whose referrer is not the hidden forwarder, the good version is returned.

Smart investigation spiders or honeyclients may be able to get the bad version with the malicious script finally. However, the challenge remain to discover the malicious intension of the script from all kinds of obfuscation techniques [16].

Objective of the Research. By analyzing the threat model and naïve solutions, we realize that embedding a nonce into the clicking URL and then validate the nonce is a viable solution. However, considering millions or billions of impression requests received by a large syndicator like Google, querying and validating the nonce is not an easy task. In this research, our goal is to develop effective solutions to combat malicious script-based click fraud attacks, which can (1) distinguish false clicks generated by malicious scripts and the clicks generated by authentic advertising clients; (2) resist replay attacks; (3) to be efficient enough to be deployed and run on heavily-loaded advertising system servers; (4) achieve low false positive and low quantifiable false negative.

3 The Proposed Approach

We propose a framework to combat malicious script-generating click frauds. The proposed framework assumes that `iframe` is already used to enforce the same-origin-policy. For simplicity, we assume that the proposed framework runs on syndicator's server, but it can also run on advertiser's or third-party's server.

On the syndicator's server, we proposed to add four operations: *creating, storing, validating* and *deleting* impression-click identifiers, where an impression-click identifier is a one-time identifier that is assigned to an impression and the following clicks on it. After an impression request is received, the *creating* operation is executed to generate an impression-click identifier and embed it into ad links that are returned to a visitor. After being created, the identifier is *stored* into a special data structure for later validation. The data structure used in this framework can serve every query in constant time and with low false negative and low false positive. This is crucial to the success of processing billions of ad-clicking requests received every day. The data structure used in this framework also has new properties to handle time-based sliding windows and remember clicked impressions. After a click request is received, the *validating* operation is executed to validate the click. If the impression-click identifier of the click is missing, or cannot be found, or has been expired, the click is classified as *invalid*. Otherwise, it is classified as *valid*. The *deleting* operation is executed periodically to delete the expired impression-click identifiers.

The proposed framework only modifies the ad handling process by additional creating and storing operations between Step 5 and Step 6 and validating operation between Step 7 and Step 8. The deleting operation is executed periodically on the syndicator's server. The modification requires only small changes on the syndicator's server, and no changes on other involved parties (visitors, publishers, advertisers). Hence, it is easy to implement and deploy.

3.1 Definition and Terminology

We present several definitions and terms used in this paper here.

Definition 1. *An* impression-click identifier *is assigned for each authentic impression and the authentic clicks on it. We define the impression-click identifier as an one-time identification vector* $\langle ID_{pub}, URL_R, IP_v, S\rangle$, *where* ID_{pub} *is the publisher's ID,* URL_R *is the URL of the referring page (described in Step 2 of Figure 1) which displays the ad content generated by the syndicator,* IP_v *is the visitor's IP address,* S *is a one-time random identifier generated by cryptographically secure pseudo-random number generator.*

Definition 2. *The* lifetime *of an impression-click identifier is defined as a time period* T. *If an ad impression were not clicked within* T, *the syndicator should expect to receive no more meaningful clicks on that impression.*

Definition 3. *A* time-based sliding window *is defined as a window which contains the impression-click identifiers that have arrived in the last* T *time units. For any time* t, *the impression-click identifiers that arrived within* $(t-T, t]$ *are valid, while all identifiers arriving before* $t-T$ *are expired (i.e., invalid).*

Definition 4. *A* timestamp *used in our framework is defined as a finite, wraparound integer that is associated with a time point. The timestamp starts at* 0 *and is increased by 1 at each new time point (clock tick). When the timestamp reaches the wraparound value* W, *it returns to* 0. *Hence, a timestamp is an integer between* $[0, W-1]$. *We assume that a sliding window with length* T *contains* N *time points. Then,* W *must satisfy* $W \geq N$.

Definition 5. *An* active timestamp *in our framework is defined as a timestamp which is not* N *older than the current timestamp. Let* ts *denote the current timestamp,* ts' *denote the timestamp to be checked. If* $(ts - ts') \mod W < N$, ts' *is active. Similarly, an* expired timestamp *is defined as a timestamp which is* N *older than the current timestamp. If* $(ts - ts') \mod W \geq N$, ts' *is expired.*

3.2 Creating Impression-Click Identifers

When a syndicator's server receives an impression request, an impression-click identifer is created. ID_{pub}, URL_R, IP_v are firstly extracted from the HTTP header and the IP header. Then, the syndicator's server generates a one-time random identifier S which is, for example, a random number generated by a

cryptography-secure random number generator. Now, the syndicator's server has constructed an impression-click identifier $\langle ID_{pub}, URL_R, IP_v, S\rangle$. The random number S is embedded into ad links of the ad page that will be returned to the visitor. In a legitimate clicking scenario, the ad link will be clicked by the same visitor at the same web page, hence S will be sent back to the syndicator with the same ID_{pub}, URL_R, IP_v as the corresponding impression request. Hence, a valid click request must have the same impression-click identifier as the corresponding impression request. In this way, we connect an impression and the following valid clicks on it together.

3.3 Storing Impression-Click Identifers

After the impression-click identifer is created, it must be stored for later validation purpose. In a large ad network, it is a challenge to store and validate the impression-click identifiers efficiently due to billions of impression and click requests may be received each day. We proposed to use a special data structure to accomplish the tasks. We also proposed to use a time-based sliding window to maintain active and expiration statuses of impression-click identifiers.

The data structure is represented as an array of m entries $P[0], P[1], \cdots, P[m-1]$, where each entry of the array contains an E-bit integer (called timestamp-integer and denoted as $E[i]$) and a bit (called click-bit and denoted as $B[i]$), where $E = \lceil \log_2(N+C+1) \rceil$. Parameters N and C will be described later. All timestamp-integers are initialized to invalid timestamps (all 1s) and all click-bits are initialized to 1s. The data structure also has k hash functions which are used to assist inserting and querying operations.

Our framework uses a sliding window to contain the items arrived within the last T time periods. The period contains N timestamps. We let the wraparound value W for the timestamps equal to $N + C$, where $C \geq 0$ is a parameter to adjust the overhead of the deleting operation and will be further explained when we present the deleting operation. Simply saying, the array may have $N + C$ different timestamps and the sliding window contains N most recent timestamps. The timestamps in the sliding window are active, and that out of the sliding window are expired. A timestamp-integer of the data structure must contain one invalid timestamp (all 1s) and $N + C$ active or expired timestamps. Hence, a timestamp-integer must have at least $E = \lceil \log_2(N + C + 1) \rceil$ bits.

Assume that the impression request arrives at time t, with corresponding timestamp $ts \in [0, N + C - 1]$. To store an impression-click identifier, the syndicator's server hashes the impression-click identifier ID by k hash functions and gets k hash results $h_i(ID) (1 \leq i \leq k)$. The corresponding k timestamp-integers, whose indices are the same as the k hash results, are set to the current timestamp ts, and the corresponding k click-bits are set to 1.

3.4 Validating Impression-Click Identifer

When a click request is received, the syndicator's server validates the impression-click identifer of the request. The syndicator's server tries to extract ID_{pub},

URL_R, IP_v, S from the HTTP header and the IP header. If S is missing, the click is marked as invalid immediately. Otherwise, we construct an impression-click identifier $ID = \langle R, ID_{pub}, IP_v, S \rangle$ for the click request.

Then, the syndicator's server queries ID in the data structure. Assume that the click request arrives at time t, with corresponding timestamp $ts \in [0, N + C - 1]$. The syndicator's server hashes ID by k hash functions and check k corresponding entries $E[h_i(ID)]$ and $B[h_i(ID)]$. If any of the k timestamp-integers is invalid (all 1s) or expired $((ts - E[h_i(ID)]) \mod (N + C) \geq N)$, undoubtedly the corresponding impression request has never been received or has been expired already. If all of the k click-bits are 0, the corresponding impression has been clicked with a very high probability. In either case, the click is classified as *invalid*. Otherwise, the click is classified as *valid*.

3.5 Deleting Expired Impression-Click Identifers

The deleting operation firstly starts at the beginning of the $(N + 1)$th time point (the timestamp is N), and then is invoked once at the beginning of each successive time point (after the timestamp is updated). Each time, the operation scans $\lceil \frac{m}{(C+1)} \rceil$ continuous entries. If an entry contains an expired timestamp, the timestamp-integer is reset to invalid (all 1s) and the click-bit is reset to 1.

We denote the starting entry of a deleting operation as $P[i]$ and the ending entry as $P[j]$. The first deleting operation starts from the head of the array and has $P[i] = P[0]$. Other deleting operations start from the next entry of the last scanned entry and have $P[i] = P[(j + 1) \mod m]$. Whenever the operation reaches the bottom of the array, it will go around to the head $P[0]$.

The proposed framework uses the parameter C to adjust the number of entries that are scanned by a deleting operation. If $C = 0$, the whole array is scanned at the beginning of each time point and the expired timestamps are cleaned. Each operation must scan m entries. By using $C > 0$, the number of scanned entries for each deleting operation is reduced to $\lceil \frac{m}{(C+1)} \rceil$. For example, when $C = 1$, only half of the entries are scanned in a deleting operation.

Compared with the traditional sliding window technique, our framework delays the deleting of an expired timestamp for at most C time points. The benefit is that we reduce the number of scanned entries, thus the running time overhead, for each deleting operation. Note that the wraparound value is $N + C$, while a sliding window contains only N timestamps. Hence, the expired timestamps that are not cleaned yet will be temporarily stored in the array. If a validating operation reads an expired timestamp, it will immediately recognize the expiration, hence will not introduce any error.

The analysis of false negative and false positive rates is simple. To save space, we do not show the proof here. More details can be found in [21].

3.6 Security Analysis

Effectiveness of the Proposed Approach Against Script-generating Click Fraud Attacks. In our proposed framework, we use a special data

structure to validate the impression-click identifier sent along with the visitor's ad click request. The unique feature of this approach is that we have very low false positive. That is, we will not likely say that a valid impression-click identifier is "invalid". From the theoretical analysis, we can also show that the false negative can be controlled to be a low and acceptable value by carefully determining the system parameters such as the size of the space and the number of hash functions. This means that the possibility that an invalid impression-click identifier is regarded as "valid" can be controlled to be low enough to be acceptable for both the advertisers and other online advertising business parties. Under our proposed framework, there is only one way for the attacks to be able to succeed: Correctly guessing and generating an active and valid impression-click identifier. However, it is practically infeasible to do so since it is hard for a malicious script to read the impression-click identifier embedded in an `iframe` without more sophisticated attacks, and we use cryptographically secure pseudo-random number generator to generate impression-click identifiers.

Effectiveness of the Proposed Approach Against Cross-Site Scripting Attack. Cross-site scripting attack, a popular attack on web applications, cannot work under our proposed framework. Such attack requires that malicious scripts are injected into the ad page that are generated by a syndicator and viewed by visitors. By doing so, the malicious script, and the malicious publisher, could be able to get access to the impression-click identifiers. But this is infeasible, because the syndicator will not accept inputs from the publishers and add it into the ad page. Hence, it is impossible to inject malicious scripts into the ad page, because the syndicator would not do it and no other parties could do it.

Effectiveness of the Proposed Approach Against Replay Attack. If an attacker is able to sniff or retrieve the impression-identifers that are embedded in an `iframe`, it is possible to replay the identifiers and generate false clicks. However, the proposed framework deletes an identifier once it is clicked. Hence, the replay attack on each identifier is restricted to once.

Effectiveness of the Proposed Approach Against Man-in-the-middle Attack. It is possible to launch sophisticated man-in-the-middle attack to intercept valid impression-click identifiers such that the malicious publisher could be able to generate malicious automatic ad clicks with the intercepted valid impression-click identifiers. But a very simple solution can effectively defend against such man-in-the-middle attacks, which is to use HTTPS instead of HTTP. With it, the man-in-the-middle attacker cannot read valid impression-click identifiers from the ad page sent by the syndicator any more.

Limitation of the Proposed Approach. Although the proposed framework is able to prevent malicious-script generating fraudulent clicks effectively, it is limited to address this type of click fraud only. The framework is not able to prevent click fraud generated by human or bot machines.

4 Experimental Evaluation

We evaluate the performance of the proposed framework using two data sets: an HTTP data set and a synthetic data set. The HTTP data set is transformed from a data set of publicly available HTTP traffic[1] during 2 weeks in 1995, which contains $3,326,797$ impression requests and $277,633$ clicking requests. The synthetic data set is generated by us according to general rules of web traffic and ad clicks, which contains $20,971,520$ impression requests and $2,023,813$ click requests. Although real clicking data are not available for evaluation, these two data sets are still able to testing performance of the proposed framework. The HTTP data set captures characteristics of real web traffic, while the synthetic data set contains much more data to test the scalability of the framework.

4.1 Experimental Setup

The original HTTP data set contains total $3,328,587$ HTTP requests. Each HTTP request has a host that made the request, a time when the request was received, and other information. We transform each HTTP request in the data set to one impression request. The impression-click identifier of an impression request simply consists of a host of the request and a random number. Such simplification will not affect the evaluation of the performance. The arriving time of the impression request is the same as the HTTP request. In the total, we have $3,326,797$ impression requests after removing the disordered requests.

We generate clicks using a typical click-through rate 0.1. That is, for each impression request generated above, there is a probability 0.1 to generate a click request for it. All clicks are generated as invalid clicks. Hence, in our evaluation, the false negative rate is approximate to the fraction of total clicks that are classified as valid clicks. In order to evaluate the capability of the proposed framework to handle different fraudulent clicks, we have purposely generated three types of invalid clicks. The first type of invalid clicks have the same identifiers as impressions, but arriving T time later (i.e. expired). The second type of invalid clicks are generated with invalid hosts (but not expired). The third type of invalid clicks are generated with different random numbers (also not expired). Different fractions of the three types of invalid clicks actually have undetectable impact on the evaluation results. In the following description, we imply that the fraction of the three types of invalid clicks is 0.2, 0.3, 0.5.

We run our evaluations on a PC with a 3GHz Pentium-4 CPU and 1GB memory. Other parameters for the HTTP data set are: $T = 1$ week ($604,800$ seconds), $N = C = 604,800$, $E = 32$ bits.

The synthetic data set is generated as follows. We generate impression requests which arrive in random time intervals. Clicks requests are generated using the similar methods as that is used to generate clicks for HTTP data set. The data totally contains $20,971,520$ impression requests and $2,023,813$

[1] ClarkNet HTTP traffic,
`http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html`

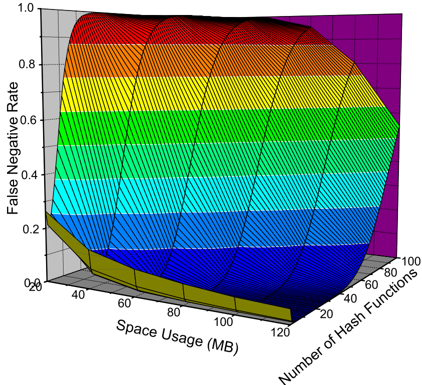


Fig. 3. False negative rate vs. Space usages and Number of hash functions

click requests. Other parameters for the HTTP data set are: $T = 4$ months , $N = C = 1,048,576$, $E = 32$ bits.

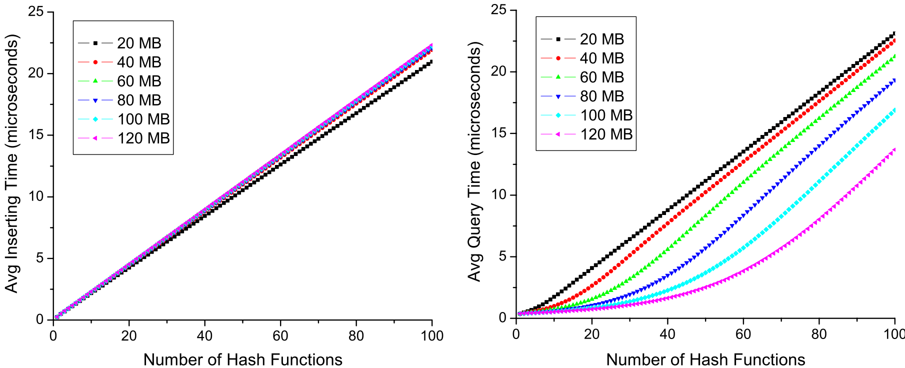
4.2 Experimental Results

We have evaluated the proposed framework using both the HTTP data set and the synthetic data set. Their results are similar, hence we show and discuss the results for HTTP data set only. More details about the synthetic results can be found in [21].

At first, we evaluate the false negative rates for different space usage and number of hash functions. The result is shown in Figure 3. We observe a shape like a gorge. The bottom of the gorge is the minimum values of m under specific space usages. Under specific number of hash functions, the false negative rate decreases when the space usage increases, because a larger memory will reduce the *collisions* between the hash results, hence can reduce the false negative rate. In this experiment, we are able to achieve a low false negative rate 0.00008 when the space usage is 120MB and $k = 13$.

The second experiment is to evaluate the time used for an inserting operation. In Figure 4(a), we observe that the inserting time increases linearly with k. We also observe that the inserting time is similar for different size of memory, i.e. the inserting overhead is almost not affected by the space usage. When we use 120MB memory and 13 hash functions, the average inserting time is as small as less than 3 microseconds.

The third experiment is to evaluate the time used for a querying operation. In Figure 4(b), we observe an interesting result that the querying time increases non-linearly when k is small, and then increases linearly when k is large enough. The reason for this observation is as below. When k is relatively small, *active* timestamps occupies a small portion of the entries. A querying operation likely meets an invalid or expired timestamp before checking all k entries and stops. A small increase of k will cause a large increase of *active* timestamps. Hence, a lot



(a) Average inserting time vs. Number of hash functions (b) Average querying time vs. Number of hash functions

Fig. 4. Average inserting or querying time vs. Number of hash functions

more entries have to be checked and the querying time increase in an exponential-like speed. When k is large enough, most of the entries are occupied by *active* timestamps. A querying operation has to check almost all k entries. Hence, the querying time increases linearly with k. We also observe that a querying operation costs less time when a larger size of memory is used. The reason is that when using a larger space, more entries have invalid or expired timestamps, hence a query operation checks less entries in average. When we use 120MB memory and 13 hash functions, the average querying time is as small as less than 1 microseconds.

5 Conclusions

In this paper, we propose an effective solution to validate and filter click frauds generated by malicious scripts from fraudulent publishers. We propose a set of operations that can create an one-time impression-click identifier for each ad impression request and validate it later. Our proposed solution has been proved to be able to achieve constant-time inserting and querying, low false positive rate and low quantifiable false negative rate.

Acknowledgments

This work was partially supported by NSF under grants No. CNS-0644238, CNS-0626822, and CNS-0831470. We appreciate anonymous reviewers for their valuable suggestions and comments.

References

1. PricewaterhouseCoopers, Iab internet advertising revenue report, 2008 full-year results, `http://www.iab.net/media/file/IAB_PwC_2008_full_year.pdf`

2. Mitchell, S.P., Linden, J.: Click fraud: What is it and how do we make it go away (December 2006), http://www.kowabunga.com/white-papers.aspx
3. Survey, O.: Hot topics: Click Fraud Reaches $1.3 Billion, Dictates End of "Don't ask, Don't Tell" Era, http://www.outsellinc.com/store/products/243
4. Click Forensics, Inc., Industry Click Fraud Rate Higher Than Ever Reaching 17.1% in Q4 (2008), http://www.clickforensics.com/newsroom/press-releases/120-click-fraud-index.html
5. Mills, E.: Google Click Fraud Settlement Given Go-Ahead (July 2006), http://news.cnet.com/Google-click-fraud-settlement-given-go-ahead/2100-1024_3-6099368.html
6. Liedtke, M.: Yahoo Settles Click Fraud Lawsuit (June 2006), http://www.msnbc.msn.com/id/13601951/
7. Daswani, N., Stoppelman, M.: The Anatomy of Clickbot.A. In: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets, p. 11 (2007)
8. Think Digit Magazine, Clickety-clack: Googlewhack! (November 2007), http://www.thinkdigit.com/details.php?article_id=1983
9. Tuzhilin, A.: The Lane's Gifts v. Google Report. Tech. Rep. (2006), http://googleblog.blogspot.com/pdf/Tuzhilin_Report.pdf
10. Metwally, A., Agrawal, D., Abbad, A.E., Zheng, Q.: On Hit Inflation Techniques and Detection in Streams of Web Advertising Networks. In: ICDCS 2007, p. 52 (2007)
11. Daswani, N., Mysen, C., Rao, V., Weis, S., Gharachorloo, K., Ghosemajumder, S.: Crimeware: Understanding New Attacks and Defenses, 1st edn., vol. 11, pp. 325–354. Addison-Wesley, Reading (2008)
12. Metwally, A., Agrawal, D., Abbadi, A.E.: Duplicate Detection in Click Streams. In: WWW 2005, pp. 12–21 (2005)
13. Zhang, L., Guan, Y.: Detecting Click Fraud in Pay-Per-Click Streams of Online Advertising Networks. In: ICDCS 2008 (June 2008)
14. Juels, A., Stamm, S., Jakobsson, M.: Combating Click Fraud via Premium Clicks. In: 16th USENIX Security Symposium, pp. 17–26 (2007)
15. Gandhi, M., Jakobsson, M., Ratkiewicz, J.: Badvertisements: Stealthy Click-Fraud with Unwitting Accessories. Journal of Digital Forensic Practice 1(2), 131–142 (2006)
16. Chellapilla, K., Maykov, A.: A taxonomy of JavaScript redirection spam. In: AIRWeb 2007: Proceedings of the 3rd international workshop on Adversarial information retrieval on the web, pp. 81–88 (2007)
17. Broder, A., Mitzenmacher, M.: Network Applications of Bloom Filters: A Survey. Internet Mathematics 1, 485–509 (2004)
18. The Same Origin Policy, http://www.mozilla.org/projects/security/components/same-origin.html
19. McGann, R.: Study: Consumers delete cookies at surprising rate (March 2005), http://www.clickz.com/3489636
20. Daswani, N., Kern, C., Kesavan, A.: Foundations of Security: What Every Programmer Needs to Know. Apress (February 2007)
21. Peng, Y., Zhang, L., Chang, J.M., Guan, Y.: An Effective Method for Combating Malicious Scripts Clickbots, Tech Report, http://www.ece.iastate.edu/~kitap/docs/clickfraud.pdf