# Detecting Hit Shaving in Click-Through Payment Schemes

Michael K. Reiter
*AT&T Labs Research*
Vinod Anupam and Alain Mayer
*Bell Labs, Lucent Technologies*

# Detecting Hit Shaving in Click-Through Payment Schemes

Michael K. Reiter
*AT&T Labs Research*
*Florham Park, NJ, USA*
reiter@research.att.com

Vinod Anupam Alain Mayer
*Bell Labs, Lucent Technologies*
*Murray Hill, NJ, USA*
{anupam,alain}@research.bell-labs.com

## Abstract

A web user "clicks through" one web site, the referrer, to another web site, the target, if the user follows a hypertext link to the target's site contained in a web page served from the referrer's site. Numerous click-through payment programs have been established on the web, by which (the webmaster of) a target site pays a referrer site for each click through that referrer to the target. However, typically the referrer has no ability to verify that it is paid for every click-through to the target for which it is responsible. Thus, targets can undetectably omit to pay referrers for some number of click-throughs, a practice called *hit shaving*. In this paper, we explore simple and immediately useful approaches to enable referrers to monitor the number of click-throughs for which they should be paid.

## 1 Introduction

Though the emergence of full-scale electronic commerce on the World-Wide-Web is proceeding slowly, the web has been quickly and aggressively realized as an effective advertising medium. Indeed, advertising itself has become an important commodity on the web. The latest evidence of this fact is the growth of *click-through payments*, in which the webmaster of one web site $B$ pays the webmaster of another site $A$ for every referral that $B$'s pages receive from $A$'s. In other words, if a web user, when viewing one of $A$'s pages, clicks on a link in that page to one of $B$'s pages (in this sense, the user has "clicked through" $A$ to reach $B$), then $A$ is entitled to payment from $B$. $B$ runs a click-through payment program to motivate others to prominently display links to site $B$ and thus to increase the traffic that $B$ receives.

Due to the structure of the HTTP protocol, click-through payment schemes as implemented on the web today hold many opportunities for fraud. The referrer $A$ is given no way to verify that it is paid for every referral its pages give $B$. This allows $B$ to undetectably "forget" some referrals that it receives from $A$, a practice called *hit shaving*. Moreover, even if $A$ were able to detect that its referrals were shaved by $B$, it has no evidence to present to a third party to argue this fact. The attention paid to hit shaving in discussions of web advertising (e.g., [Kle98]), often in advertisements for the click-through programs themselves, suggests that hit shaving is a recognized and prominent problem in the click-through industry today. Moreover, the stakes suggest that fraud is likely: some click-through programs advertise surprisingly large payments (e.g., up to \$6 per click-through) and prizes based on click-throughs (e.g., one web site advertised a click-through contest in which first prize was a 1998 Corvette).

The purpose of this paper is to bring the problem of hit shaving to the attention of the technical community and to explore remedies to the problem. We first focus on solutions that can be immediately useful on the web today: we offer web constructions (i.e., ways to construct web pages and CGI scripts) that enable a webmaster to monitor how often users click through her pages to others, and to which pages they click. Moreover, these techniques require no cooperation or awareness by the sites to which the referrals are made, making them very effective. Though only heuristic in nature and not foolproof, these techniques can immediately offer webmasters greater ability to detect hit shaving by click-through payment programs. We then explore more ambitious approaches that, with the cooperation of click-through program providers, enables webmasters to monitor more precisely the number of click-throughs for which they should be paid, and to even obtain nonrepudiable evidence of these click-throughs from the target site. Even though this second set of approaches requires cooperation by the webmasters of the click-through payment programs, these webmasters need not be trusted, in the sense that their failure to cooperate is quickly detectable. All of the techniques that we propose are largely invisible to the web user, in that the user experiences nothing out of the ordinary as a consequence of these techniques. Moreover, these techniques work with off-the-shelf browsers today.

## 1.1 The problem

We begin with a brief overview of how click-through payment programs work today, focusing on their susceptibilities to fraud and the various concerns that shape our solutions. We draw this description from several prominent examples of click-through programs on the web. In the rest of this paper, we often speak of a web site and its webmaster (i.e., the person that controls the content it serves) synonymously.

A click-through agreement is set up when (the webmaster of) one site $A$, the *referrer*, applies for an account at the *target* site $B$ that is running a click-through payment program. This typically takes place by $A$ filling out a form for $B$ in which $A$ provides, for example, its address to which payment checks should be mailed. After establishing the account, $A$ is given advertising material in the form of Hypertext Markup Language (HTML) commands to include in its web pages, possibly along with accompanying images ("banners") to display on its web pages. These HTML commands include a hypertext link to the target site $B$; i.e., when a user views $A$'s page and clicks on this link, then the user's browser retrieves the referred-to page from $B$. In this sense, the user has "clicked through" $A$ to get to $B$. Typically $B$ maintains an account statement for $A$, so that $A$ can periodically visit site $B$ to see how many referrals $A$'s pages have made to $B$ (and thus the amount of money to which $A$ is entitled).

To understand the risks for fraud in this mechanism, we need to review the actual Hypertext Transfer Protocol (HTTP) messages exchanged during a click-through. This exchange is shown in Figure 1. The exchange begins when the user's browser retrieves a web page from site $A$, say `http://siteA.com/pageA.html`. This page contains a hypertext link to a page served by $B$, say

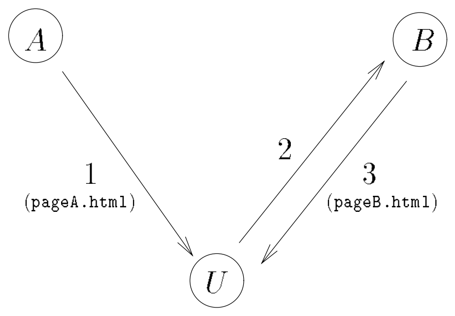

Figure 1: **A click-through**: User $U$ retrieves `pageA.html` from $A$ (message 1) and clicks on a link in it, causing `pageB.html` on site $B$ to be requested (message 2) and loaded (message 3).

`http://siteB.com/pageB.html`. $A$ included this link in `pageA.html` when it registered to participate in $B$'s click-through payment program. When the user clicks on that link, her browser retrieves `pageB.html` from $B$. $B$ can use a header of the HTTP protocol, the `Referrer` header, to determine the URL of the page that referred the user to site $B$. In this case, the `Referrer` field will indicate the URL of `pageA.html`, and so $A$ will be credited with the referral.

The possibilities for abuse in this system should be evident. Since there is no communication to $A$ after the user clicks on the link to `pageB.html`, there is no way for $A$ to know how many referrals its pages give $B$. So, $B$ can just ignore the `Referrer` field of requests and thus fail to give proper credit to $A$; as described earlier, this is called *hit shaving* in the click-through payment industry. $B$ is also subject to abuses by $A$, e.g., if $A$ generates false requests to $B$ with `Referrer` fields naming `pageA.html`. In this way, $A$ can unfairly inflate the payment that it receives from $B$, and we will henceforth refer to such practices by $A$ as *hit inflation.* Like shaving, hit inflation is a recognized problem in the click-through payment industry. Many providers of click-through payment programs threaten cancellation of a referrer's account if hit inflation by an account holder is detected, but for obvious reasons providers do not typically reveal their methods for detecting hit inflation. It is likely that these methods are at least partially based on monitoring user IP addresses, and in particular detecting multiple requests from the same IP address or domain. Indeed, many click-through programs agree to pay only for "unique" referrals, i.e., referred requests from users at different addresses.

In this paper we treat only the problem of hit shaving. Nevertheless, the threat of hit inflation shapes the class of solutions that we are willing to consider. For example, one approach to detect hit shaving would be for $A$ to craft `pageA.html` so that its link purportedly to `pageB.html` is actually a link to a URL on site $A$; then, when the user clicks on that link, $A$ retrieves `pageB.html` from $B$ and serves it to the user. This enables $A$ to precisely know how many referrals it gives to $B$. However, this exacerbates the problem of detecting hit inflation, because it establishes a norm in which $A$ directly issues to $B$ all requests for which it should be credited. This hampers $B$'s ability to detect hit inflation based on user IP addresses. For this reason, we require that our solutions do not change the fact that the user's browser requests $B$'s pages directly from $B$.

### 1.2 Goals and assumptions

In the remainder of this paper, our goal is to enable site $A$ to monitor how many legitimate referrals it gives to site $B$, or more specifically, how many times $B$ receives an HTTP request from a user's browser for `pageB.html` with a `Referrer` header naming some URL on site $A$ (i.e., message 2 in Figure 1); we take this as the number of click-throughs for which $A$ should be paid. Note that this number includes any such request received by $B$, regardless of how $B$ responded to it (with `pageB.html` or with an error message), but it does not include requests that $B$ did not receive, e.g., because $B$ was down.[1] However, because $A$ cannot monitor exactly which messages $B$ receives, we must settle for solutions that enable $A$ to approximate this number. For reasons discussed in Section 1.1, it is difficult to monitor this number given the way that click-throughs presently work. Thus, our solutions modify how the click-through happens, but we allow them to do so only in a way as to impact traffic patterns, server load, and users' experiences as little as possible.

In forming our solutions, we assume that the user's browser is a frame-enabled and JavaScript-enabled off-the-shelf browser. We view the user's browser as a trusted-but-oblivious third party: we assume that it faithfully interprets the web pages (including JavaScript commands, when necessary) fed to it, but we do not assume that it has been modified in any way to support our approaches. In the JavaScript code segments included in this paper, we have allowed ourselves the full expressiveness of JavaScript 1.2, but versions for JavaScript 1.1, and in some cases JavaScript 1.0, can be formulated. The effectiveness of our JavaScript code segments has been verified using both Netscape Communicator 4.03 and Internet Explorer 4.0 over Windows 95 as the user's browser (subsequently abbreviated "NC4" and "IE4", respectively) and two Apache web servers on different hosts in different domains as sites $A$ and $B$.

[1] Other definitions are possible; e.g., the number of click-throughs for which $A$ should be paid might not include those in which $B$ responded with an error message. Nevertheless, in most cases the claims we make for our schemes continue to hold even under other reasonable definitions.

## 2 Upper bounds on referrals

A first approach to detect hit shaving is to modify the click-through sequence to elicit a notification from the user's browser to the referring site $A$ when the user clicks the link to `pageB.html` in `pageA.html`. In this way, $A$ can monitor how many times users have clicked through `pageA.html` to `pageB.html` by monitoring how many such notifications it receives. In this section we show two approaches for achieving this, or more specifically for turning the exchange of Figure 1 into one that looks like Figure 2. As Figure 2 shows, once the user clicks on the link to `pageB.html`, a notification is sent back to $A$ (message 2) and then `pageB.html` is retrieved (messages 3,4). Neither of the methods we propose requires cooperation from site $B$, and both are invisible to the user.

Though effective, the methods of this section enable site $A$ only to record an *upper bound* on the number of referrals for which $A$ should be credited, not an exact count. The reason for this is that the notification sent to $A$ (message 2 in Figure 2) is an indication only that the user's browser will request `pageB.html`, not that $B$ has received this request. To see the importance of this distinction, the webmaster of site $B$ could plausibly claim that site $B$ was down or heavily overloaded for some significant period of time (causing requests to be dropped), and thus no referrals were completed (or thus credited to $A$'s account) during that time. An approach that enables $A$ to additionally record a lower bound on its number of referrals is the topic of Section 3.

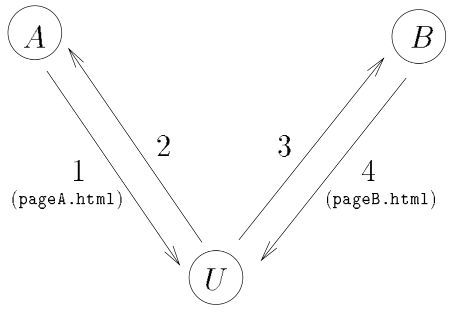

Figure 2: **Bounding referrals from above**: User $U$ retrieves `pageA.html` from $A$ (message 1). When the user clicks on the link to `pageB.html`, the user's browser sends a notification to $A$ (message 2) and then retrieves `pageB.html` (messages 3,4).

## 2.1 Using HTTP redirection

The first approach we describe for achieving the interaction of Figure 2 uses the "redirect" feature of the HTTP protocol. When a browser requests a URL from a web server, the server can return an HTTP redirection status code in the range 300–399 (e.g., `301 Moved Permanently`), which indicates to the browser that it should look for the page at another URL. This other URL is specified in the `Location` HTTP header. Upon receiving a response with status code `301 Moved Permanently`, the browser finds the `Location` header and immediately issues a request for the URL specified in that field.

Given this mechanism, one way for site $A$ to monitor the number of clicks through `pageA.html` to `pageB.html` is to craft `pageA.html` so that its link purportedly to `pageB.html` is really a link to a "dummy" URL on site $A$. If site $A$ is configured to redirect requests for this dummy URL to `pageB.html` on site $B$, then $A$ can easily monitor clicks through `pageA.html` to `pageB.html` by monitoring the number of requests for the dummy URL. The resulting web transaction proceeds as shown in Figure 2: after receiving `pageA.html`, the user clicks on the link purportedly to `pageB.html`, which causes the dummy URL on site $A$ to be requested (message 2). $A$ returns an HTTP redirect header with the `Location` field set to the URL of `pageB.html`, which causes the browser to retrieve `pageB.html` (messages 3,4). The number of requests for the dummy URL is an indicator of the number of clicks through `pageA.html` to `pageB.html`.

One practical obstacle to this approach as described so far is that it employs a reconfiguration of the web server on site $A$, which may not be possible if the participant in the click-through program does not have the authority to reconfigure the web server on site $A$. It is possible to effect this redirection without reconfiguring the web server by using CGI programming. On many web servers, a CGI script that returns a properly formatted `Location` header will cause a redirection to the URL named in that header. So, if the dummy URL on site $A$ is the URL for a CGI script that outputs a `Location` header set to the URL of `pageB.html`, then this achieves the exchange of Figure 2. This exchange can also be achieved by using a *no parse header* (NPH) script, which is a CGI script that is allowed to entirely control the HTTP headers in the response sent back to the browser. An NPH script that explicitly returns a redirection status code and `Location` header can also be used to effect the desired redirection.

It is worth noting that some obvious HTML-only approaches to effecting this redirection do not suffice because they cause the HTTP `Referrer` field to be blanked in the request to $B$, thereby precluding $A$ from getting credit for the click-through. One such approach is to craft `pageA.html` so that its link purportedly to `pageB.html` is really a link to an HTML page on site $A$ that immediately "refreshes" the user's browser to `pageB.html` using HTML's `<meta>` tag (see [MK97, Section 14.2]). Using this approach with NC4 and IE4, the `Referrer` field that $B$ received was empty.

## 2.2 Using JavaScript

In this section, we describe a second way of achieving the message exchange shown in Figure 2. The main difference of this approach from that of the previous section is that the message exchange is achieved by embedding JavaScript commands in `pageA.html`, rather than employing HTTP redirection. It is also instructive for introducing techniques that will be useful in Section 3.

Suppose that when signing up for $B$'s click-through payment program, $A$ is instructed by $B$ to place the following link to $B$ in its page:

```
<a href="http://siteB.com/pageB.html">
Click here for site B.
</a>
```

In this approach, the webmaster of $A$ constructs its `pageA.html` as follows. First, she makes a file `pageAcontents.html` that contains the (HTML commands to generate the) actual contents that she wants to display to the user, including the link to site $B$. This file looks as shown in Figure 3. The important aspect of `pageAcontents.html` is the `onClick` attribute added to the link to $B$. When the user clicks this link, the browser first executes the JavaScript code in the `onClick` attribute before retrieving `pageB.html`. In this case, the `onClick` attribute invokes a function called `notify`, defined in `pageA.html` as shown in Figure 4.

The page `pageA.html` consists of a header containing a JavaScript function `notify`, and a body consisting of two frames: one named `visible` and displaying `pageAcontents.html` (the file in Figure 3), and the other named `invisible` and initially blank. As their names suggest, the `visible` frame consumes 100% of the browser window (see the `<frameset>` tag); the `invisible` frame is truly invisible to the user. When the user clicks the link to `pageB.html` in `pageAcontents.html`, this invokes the `notify` function of `pageA.html` with the URL of `pageB.html`. The `notify` function requests a URL on site $A$. This URL serves the same purpose as the dummy URL of Section 2.1, i.e., monitoring requests for this URL is the means by which $A$ keeps track of the clicks through `pageA.html` to `pageB.html`. For example, here this URL is the URL of a CGI script on site $A$ (`record.cgi`), which is provided the URL of `pageB.html` as input (following the `?`) for recording. Because JavaScript does not support general networking but does support fetching URLs, the `notify` function invokes `record.cgi` in a roundabout way, namely by fetching the output of the CGI script and "displaying" it to the user in the `invisible` frame. It does this by assigning the `location` property of the `invisible` frame to be the URL of the CGI script. When invoked, `record.cgi` simply records the referral to `pageB.html` and returns.

Using this simple trick, the click-through sequence has been transformed from that in Figure 1 to that in Figure 2. The browser invokes `record.cgi` on site $A$ (message 2) before retrieving `pageB.html` from site $B$ (messages 3,4). By using logs kept by `record.cgi`, site $A$ can monitor an upper bound on how many referrals its pages have made to $B$.

The main risk to the claim that $A$ maintains an upper bound on its referrals to $B$ with this scheme is that the browser's connection back to $A$ (message 2) might be preempted by the retrieval of `pageB.html`. This is conceivable if (i) the setup of the connection back to $A$ is delayed, e.g., due to network congestion, and (ii) `pageB.html` is a page that overtakes the top-level browser window (as opposed to displaying in the `visible` frame only), thereby overwriting `pageA.html`. If this is deemed a significant risk, then `pageB.html` can be displayed in a separate browser window (so that `pageA.html` is not overwritten) by including a `target` attribute in the link to `pageB.html` in `pageAcontents.html`.

## 3 A lower bound on referrals

In this section we describe a somewhat different approach to recording the number of referrals that $A$ gives to $B$. The method of this section addresses one limitation of those in Section 2, namely that the referral count recorded by $A$ is only an upper bound on the number of referrals it gives $B$. The solution in this section enables $A$ to infer a *lower bound*, i.e., a number of referrals for which $A$ has confidence that $B$ actually received the referred request. To achieve this, we modify our strategy so that $A$ is notified by the user's browser *only if* $B$ responds to the browser's request for `pageB.html`. That is, our goal is a protocol like that shown in Figure 5, where the browser first retrieves `pageB.html` (messages 2,3) and then informs $A$ of this afterwards (message 4).

Achieving the interaction of Figure 5 is more complicated than the simple tricks of Section 2. The general strategy that we take is as follows. When the link to `pageB.html` in `pageA.html` is clicked by the user, `pageA.html` opens a new browser window, named `nextpage`, and directs `pageB.html` to be displayed there. This enables JavaScript embedded in `pageA.html` to continue to run in the original window while `pageB.html` is being loaded. The goal then is for the `pageA.html` script to detect when the `nextpage` window has received a response from site $B$ (i.e., message 3 in Figure 5), indicating that $B$ has received the HTTP request for `pageB.html` including the `Referrer` field crediting $A$ for the refer-

```
<html>
<!-- File: pageAcontents.html -->
...
<a href="http://siteB.com/pageB.html"
   onClick="parent.notify('http://siteB.com/pageB.html')">
Click here for site B.
</a>
...
</html>
```

Figure 3: File `pageAcontents.html` for scheme of Section 2.2

```
<html>
<!-- File: pageA.html -->
<head>
<script language="JavaScript">
function notify(url) {
    invisible.location="http://siteA.com/cgi-bin/record.cgi?refer=" + url;
}
</script>
</head>

<frameset rows="100%,*">
<frame src="pageAcontents.html" name="visible">
<frame src="about:blank" name="invisible">
</frameset>

</html>
```

Figure 4: File `pageA.html` for scheme of Section 2.2

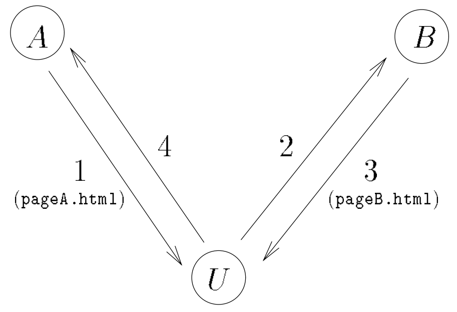

Figure 5: **Bounding referrals from below**: User $U$ retrieves `pageA.html` (message 1) and clicks on a link in it, causing `pageB.html` to be requested (message 2). Only if $B$ responds (message 3), the browser notifies $A$ of the referral (message 4).

ral. When it detects this, the script in `pageA.html` causes a URL on site $A$ to be requested, thereby notifying $A$ of the referral (message 4 in Figure 5).

The complexity in this approach is in the means by which the `pageA.html` script detects that site $B$ has responded. A first attempt might be for the script to set the `onload` event handler for the `nextpage` window when that window is created. The `onload` event handler is invoked when `pageB.html` finishes loading into the `nextpage` window (see [Fla98]). Thus, if `pageA.html` sets the `onload` event handler to be a function that notifies site $A$ of the referral, this would achieve the exchange of Figure 5. While this works with NC4, it does not work with IE4: presumably for security reasons, IE4 clears the `nextpage` window's `onload` event handler before loading `pageB.html`, and so $A$ is not notified when `pageB.html` has loaded. Moreover, NC4's failure to clear the `onload` event handler is arguably a weakness in its security model that should disappear in future versions of the browser (see [AM98]).

Fortunately, security mechanisms similar to those that cause this approach to fail for IE4 can be exploited in both browsers to achieve the effect we desire. The approach we take is for the script in `pageA.html` to periodically probe the JavaScript namespace of the `nextpage` window. Before $B$ has responded, these probes will be allowed by the browser. After $B$ has responded, however, these probes will be disallowed by the browser's security mechanisms (see [Fla98, Chapter 21]), causing a JavaScript error. By specifying an appropriate error handler for this error, the script in `pageA.html` can notify site $A$ of the referral. In the remainder of this section, we present an implementation of these ideas. For simplicity, our implementation here is not fully general; in particular, it suffices only for the case in which `pageA.html` offers a link to only one target site $B$. However, it can be generalized so that `pageA.html` can offer multiple target sites.

As in Section 2.2, our solution here is structured using a file `pageAcontents.html` to hold the actual contents of the page that $A$ wants to display to the user (including the link to $B$'s page), which is served to the user within `pageA.html` in a `frameset`. The file `pageAcontents.html` now looks as shown in Figure 6. The two differences from the previous `pageAcontents.html` (Figure 3) are the addition of a `target` attribute in the link and the invocation of `setup` (vs. `notify`) in the `onClick` event handler. Due to the latter, when the link is clicked, now the `setup` function is invoked, which is defined in `pageA.html` as shown in Figure 7.

When invoked, the `setup` function opens a new browser window named `nextpage` (as specified in the second argument of the `window.open` method call). This name is the value of the `target` attribute of the link to `pageB.html` in `pageAcontents.html` (see Figure 6), which means that `pageB.html` will be displayed in this new window when it is eventually retrieved. To ensure that the script in `pageA.html` is allowed to probe the namespace of `nextpage` until $B$ has responded, `setup` initially writes a simple HTML page into `nextpage`, using the `document.open`, `write`, and `close` methods.

The probes into the namespace of the `nextpage` window are performed by the `probe` function. The `probe` function attempts to read a portion of the namespace of the `nextpage` window (in this case, its `location.href` property) that will cause an error after $B$ has responded but will be allowed beforehand. If its read is allowed, then the `probe` function sets a timer so that it is invoked again 100 milliseconds later. Otherwise the error handler for the window in which this script is running, i.e., the window displaying `pageA.html`, is invoked. This error handler schedules an invocation of the `notify` function (see the line before the `</script>` tag in Figure 7).[2] As in Section 2.2, this function invokes the `record.cgi` CGI script on site $A$ with the URL

[2]The scheduled delay (in Figure 7, of one second) before invoking `notify` avoids various race conditions in IE4, yielding a more robust implementation for this platform.

```
<html>
<!-- File: pageAcontents.html -->
...
<a target="nextpage"
   href="http://siteB.com/pageB.html"
   onClick="parent.setup('http://siteB.com/pageB.html')">
Click here for site B.
</a>
...
</html>
```

Figure 6: File `pageAcontents.html` for scheme of Section 3

of `pageB.html`, which was stored in the `retrieved` variable during the execution of `setup`. Again, the trick of assigning to the `location` property of an invisible frame is used to invoke `record.cgi`.

To summarize, this achieves the mechanism shown in Figure 5: when the link to `pageB.html` in `pageAcontents.html` is clicked by the user, (i) the `setup` function is invoked to open a new browser window; (ii) `pageB.html` is retrieved and displayed in that window (messages 2,3); (iii) an error is encountered in `probe`; (iv) the error handler is invoked to schedule `notify`, which (v) invokes `record.cgi` on site $A$ with the URL of `pageB.html` (message 4). Moreover, if $B$ does not receive message 2, then neither message 3 nor message 4 will be sent. Like the mechanisms of Section 2, this technique requires no cooperation from $B$ for $A$ to track the referrals its pages give to $B$. And again, this technique presents nothing out of the ordinary to the user; spawning new windows is not uncommon while following links to other sites.

The main factor limiting the accuracy of $A$'s referral counting with this scheme appears to be the risk that the user closes the window containing `pageA.html` before `notify` is invoked. In this case, the script in `pageA.html` will be halted before site $A$ is notified of the referral. Thus, at best we can claim that this mechanism reports a lower bound on the number of referrals that $A$ gives to $B$. While there are other potential sources of inaccuracy, we believe that those of which we are aware can be discounted or virtually eliminated. For example, there is a risk that the user aborts the loading of `pageB.html` before receiving a response from site $B$ and instead loads a different page in the `nextpage` window, in which case the referral notification to $A$ would be erroneous. However, this risk is mitigated if the window is created with no `location` line or toolbar,[3] as is achieved by the third argument of the `open` call in Figure 7. Another risk is that some party invokes `record.cgi` arbitrarily, and in particular when no referral has taken place. However, there seems to be little practical motivation for such "attacks".

Because this scheme reports only a lower bound on referrals, perhaps the most prudent use of it is in combination with that of Section 2. Combined in the obvious way, these mechanisms enable site $A$ to retain both an upper and lower bound on the number of referrals that it has given to $B$, and typically these numbers should be very close to one another. A large gap in these bounds indicates to the webmaster of site $A$ that she should examine the availability of site $B$. Even without further evidence, large discrepancies between these upper and lower bounds may be good cause to no longer advertise $B$'s pages.

## 4 Cooperative approaches

Though the schemes of Sections 2 and 3 enable participants in a click-through program to approximate the number of click-throughs for which they should

[3] This does not completely prevent a user from loading a different page into the `nextpage` window before `pageB.html` loads: e.g., the user may still use "drag and drop" features or keyboard shortcuts to load a different URL into the window. However, we expect that the percentage of users employing these mechanisms is small and thus that users loading different pages into the `nextpage` window will yield insignificant error in the lower bound, especially since there is typically a very limited time frame (i.e., before site $B$ responds) in which the user would need to load the different page.

```
<html>
<!-- File: pageA.html -->
<head>
<script language="JavaScript">
var retrieved = null;
var w = null;

function setup(url) {
        retrieved = url;
        w = window.open("", "nextpage", "scrollbars,resizable,status");
        w.document.open("text/html");
        w.document.write("<html></html>");
        w.document.close();
        probe();
}

function probe() {
        if (w.closed) return;
        var temp = w.location.href;
        setTimeout("probe()", 100);
}

function notify() {
        if (w.closed) return;
        invisible.location="http://siteA.com/cgi-bin/record.cgi?refer=" + retrieved;
}

window.onerror = function() { setTimeout("notify()", 1000); return true; };
</script>
</head>

<frameset rows="100%,*">
<frame src="pageAcontents.html" name="visible">
<frame src="about:blank" name="invisible">
</frameset>

</html>
```

Figure 7: File `pageA.html` for scheme of Section 3

be paid, there is still some room for error in these approaches. In this section, we show that even greater accuracy can be achieved if site $B$ cooperates with site $A$ to enable $A$ to more effectively monitor $B$'s behavior. While the schemes of this section require cooperation by site $B$, this does not imply that $A$ must fully trust $B$. Rather, if $B$ misbehaves, then it risks detection by site $A$, with high probability if $A$ combines the approaches of this section with those of Sections 2 and 3. Site $B$ might be willing to cooperate in these schemes to instill trust in its referrers, in the hopes of obtaining more clients for its click-through program.

### 4.1 Click-through acknowledgements

In the first approach that we propose, site $B$ effectively "acknowledges" each referral from $A$ as the click-through happens. $B$ could send an acknowledgement to $A$ directly, i.e., by sending it in a message to $A$, but this requires $B$ to incur more costs (e.g., connection setups) than is necessary. Rather, here we review a simple way in which $B$ can piggyback the acknowledgement on its reply to the user, so that the user's browser will forward the acknowledgement to $A$.

Transferring an acknowledgement from $B$ to $A$ via the user's browser can be achieved easily with the addition of a CGI script to site $B$ and some modifications to `pageB.html`. To begin with, $B$ sets up a CGI script that serves $B$'s web pages (possibly only for referrals from click-through program participants). Let's call this script `siteB.com/cgi-bin/serve.cgi`. This CGI script accepts as input the name of a page to produce (e.g., `pageB.html`) and emits a version of `pageB.html` that is slightly different for each referred request. Specifically, if $B$ receives a request for `pageB.html` referred by $A$, then the version of `pageB.html` served by `serve.cgi` requests a dummy URL on site $A$ when it loads. Each retrieval of this dummy URL is an implicit "acknowledgement" from $B$.

A more explicit acknowledgement can be achieved if the dummy URL on site $A$ is a CGI script that `pageB.html` can invoke with $B$'s site name and the time of the referral. For example, `pageB.html` emitted from `serve.cgi` might look as shown in Figure 8. Notice that the visible contents of the page, `pageBcontents.html`, are served within a `frameset` with one visible frame and one invisible frame (analogous to how $A$ served `pageA.html` in Sections 2.2 and 3). As the `frameset` loads, the browser invokes $A$'s `record.cgi` with the arguments provided by $B$. Again, the trick of invoking `record.cgi` by writing its output to an invisible frame is used, but this time it is done by `pageB.html` (vs. `pageA.html`). Alternatively, `record.cgi` could be invoked from an `<img>` tag, for example. The `record.cgi` CGI script on site $A$, upon being invoked, can verify that the arguments properly acknowledge $A$'s referral.

Because $B$ could serve a `pageB.html` that does not invoke $A$'s `record.cgi`, it is advisable for $A$ to construct `pageA.html` as in Section 2, i.e., so that $A$ is informed whenever the user clicks on the link to `pageB.html`. This will alert $A$ if $B$ routinely serves a `pageB.html` that does not invoke the appropriate callback to $A$'s `record.cgi`. $A$ could further employ the mechanism of Section 3, providing $A$ with the full detection capabilities offered by both approaches. In this light, the technique of this section can be viewed as a way for $B$ to help $A$ improve its click-through monitoring over what $A$ can achieve without $B$'s help using the schemes of Section 2 and Section 3. In particular, $B$'s acknowledgement may reach $A$ even if the notification from the scheme of Section 3 does not (e.g., because the user prematurely closes the window containing `pageA.html`). If $A$ couples the detection techniques of Sections 2 and 3 with random inspections of `pageB.html` as served by $B$ on a referral, $B$ stands a high probability of being caught if it fails to acknowledge $A$ a significant portion of the time.

### 4.2 Click-through nonrepudiation

One drawback of all our previous schemes is that while they enable a referrer to detect hit shaving, they do not arm the referrer with any evidence to present to a third party in the case of a dispute. So, in the extreme, the webmaster of site $B$ can repudiate some or all referrals, including any acknowledgements it sent, and refuse to pay certain referrers. While these referrers are thus likely to leave $B$'s click-through program, there is nothing that they can do to bring third-party leverage on the dispute. In this section we extend the technique in Section 4.1 to enable $B$ to pass nonrepudiable acknowledgements to the referrer. Again, any failure of $B$ to cooperate is quickly detectable by the referring site $A$ (if combined with the techniques of

```
<html>
<!-- pageB.html, dynamically generated by serve.cgi -->

<frameset rows="100%,*">
<frame src="pageBcontents.html">
<frame src="http://siteA.com/cgi-bin/record.cgi?refer=siteB.com&when=Feb2_13:04_EST_1998">
</frameset>
</html>
```

Figure 8: File `pageB.html` in scheme of Section 4.1

Sections 2 and 3), and so the webmaster of $A$ can immediately take action to avert a dispute, rather than wait until, say, the end of the month to find out that $B$ will not pay her.

### 4.2.1 Using digital signatures

If there is a well-known public key for authenticating site $B$ via digital signatures (e.g., [RSA78]), then one approach for $B$ to provide nonrepudiable acknowledgements to $A$ is for $B$ to pass a digital signature to $A$ as part of the click-through protocol. This signature could sign a tuple containing the IP address of the user, the time and date of the referral, the page to which the referral was made, and the referring page. $A$ can then retain this signed tuple for use in a dispute with $B$ later, if necessary. Like in Section 4.1, $B$ can create this signature in `serve.cgi` and include it within `pageB.html`, to be passed as an argument to a CGI script on site $A$ by the user's browser when `pageB.html` loads.

A drawback of this approach is that it requires $B$ to compute a digital signature per referral, which must be done on its critical path for servicing the user's request. Because digital signatures, particularly RSA signatures [RSA78], tend to be computationally intensive, the additional computational load imposed by these signatures may be prohibitive if $B$ is a very busy server.

### 4.2.2 Using hash chains

In order to lessen the computational burden on $B$, in this section we sketch an approach that requires far less from $B$ computationally but that still provides some degree of nonrepudiable evidence to $A$. It employs the well-known idea of *hash chaining*, which has been used in the past for efficient user authentication [Hal94] and micropayments [RS95], among other things.

Again we assume that there is a well-known (i.e., authenticated) public key for $B$. When $A$ signs up for $B$'s click-through payment program, $B$ generates a large, unpredictable number $s$, applies a one-way hash function (e.g., [SHA95]) $f$ to it $k$ times to produce $\ell = f^k(s)$, digitally signs the pair $<k, \ell>$, and sends $<k, \ell>$ and the signature to $A$. Note that all of this takes place when $A$ registers for the click-through program, not on the critical path of referrals. Once this is set up, rather than passing a digital signature back to $A$ during a referral, $B$ simply passes back the pair $<i, \ell'>$ to $A$, where $\ell' = f^{k-i}(s)$, for the $i$-th referral ($1 \leq i \leq k$) that $A$ gives it. $B$ can pass the pair $<i, \ell'>$ to $A$ using techniques like those of Section 4.1. $A$ can verify the correctness of this pair by verifying that $\ell = f^i(\ell')$. In the event of a dispute, $A$ need only present the digitally signed pair $<k, \ell>$ and a pair $<i, \ell'>$ where $\ell = f^i(\ell')$ to convince a third party that $B$ received at least $i$ referrals from $A$.

This scheme has some disadvantages in comparison to that of Section 4.2.1. The main disadvantage is that $B$ can later repudiate the user's IP address in this scheme. In payment programs that pay only for "unique" referrals, $A$'s inability to record the user IP address in a way that prevents $B$ from later repudiating it could leave $A$ at a disadvantage in a dispute. An agreement that $B$ returns a referral record (i.e., a new pair $<i, f^{k-i}(s)>$) only for unique referrals may restore the balance, but only if $A$ is prepared to verify, when $B$ refuses to return a referral record, that the referred user was a repeat user. A second disadvantage of this scheme is that it must periodically be "refreshed"; i.e., once $k$ referrals from $A$ to $B$ have been made, then $A$ must obtain a new signed pair $<k', f^{k'}(s')>$ from $B$.

## 5 Discussion

As mentioned in Section 4.2.2, our use of hash-chaining is similar to its use in certain micropayment schemes, specifically the PayWord scheme due to Rivest and Shamir [RS95]. This similarity is perhaps not coincidental, in that the deployment of a micropayment scheme, or more generally any digital cash scheme, could be a useful tool to counter hit shaving. In this case, the target of a referral could be required to pass a digital coin back to the referrer in the referral protocol, e.g., using the techniques of Section 4. The referrer could thus collect immediate payment for referrals it gives, and detect when payment is not being received.

Other potential developments that could expand our options for countering hit shaving include the adoption of a richer security model for JavaScript. For example, Anupam and Mayer [AM98] propose a JavaScript security model in which a script can selectively allow other scripts to access portions of its namespace by configuring access control lists accordingly. If adopted, this could enable other cooperative solutions in which `pageB.html` allows a script in `pageA.html` to access some portion of its namespace, so that the script in `pageA.html` can confirm when `pageB.html` has loaded and notify site $A$. Such solutions have the advantage of allowing `pageB.html` to be a static page (as opposed to one dynamically generated by a CGI script on site $B$), though they also place requirements on `pageA.html` that the solutions of Section 4 do not.

Although the techniques proposed in this paper are effective for detecting hit shaving, they do have the adverse effect of eroding user privacy further than the web already does today. That is, the web today, via the `Referrer` HTTP header, often reveals to a site the page that a user visited previously. Our techniques further enable the referring site to learn the page that the user visits next. Mechanisms for anonymously surfing the web, such as the Anonymizer, the Lucent Personalized Web Assistant [GGMM97], and Crowds [RR98][4] are generally incompatible with click-through payment programs on two counts: they strip out the `Referrer` field, and they preclude monitoring of user IP addresses for the purposes of detecting hit inflation. The former can be remedied by configuring these systems to let the `Referrer` field remain; the latter obstacle appears more difficult to overcome.

[4] See `www.anonymizer.com`, `lpwa.com`, and `www.research.att.com/projects/crowds`, respectively.

This work leaves several open problems. In particular, we have not attempted to address the problem of hit inflation, but have only attempted to not exacerbate it. More robust approaches for detecting or preventing hit shaving should also be explored.

**Acknowledgments** We are grateful to Avi Rubin of AT&T Labs and the anonymous referees for providing encouragement and helpful comments.

## References

[AM98] V. Anupam and A. Mayer. Security of web browser scripting languages: Vulnerabilities, attacks, and remedies. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[Fla98] D. Flanagan. *JavaScript: The Definitive Guide*. 3rd edition, O'Reilly & Associates, 1998.

[GGMM97] E. Gabber, P. Gibbons, Y. Matias, and A. Mayer. How to make personalized web browsing simple, secure, and anonymous. In *Proceedings of Financial Cryptography '97*, 1997.

[Hal94] N. M. Haller. The S/Key$^{TM}$ one-time password system. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems*, 1994.

[Kle98] D. Klein. *Succumbing to the dark side of the force: The Internet as seen from an adult web site*. Invited talk at the 1998 USENIX Annual Technical Conference, June 17, 1998.

[MK97] C. Musciano and B. Kennedy. *HTML: The Definitive Guide*. 2nd Edition, O'Reilly & Associates, 1997.

[RR98] M. K. Reiter and A. D. Rubin. Crowds: Anonymous web transactions. *ACM Transactions on Information and System Security* 1(1), June 1998.

[RS95] R. Rivest and A. Shamir. PayWord and MicroMint: Two simple micropayment schemes. Manuscript, 1995.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21(2):120–126, February 1978.

[SHA95] FIPS 180-1, Secure hash standard. Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 17, 1995.