Noninvasive Clickfraud Detection through Decoy Link Design Adaptation

***Abstract*—Online advertising is prone to various types of fraudulent activity due to vulnerabilities. Regardless of the advertising business model, fraudulent clicks are capable of exhausting advertisers' budgets as well as decreasing publishers' client-base. In this paper we focus on click fraud caused by Clickbots, automated programs or scripts that click on an advertiser's link with no interest in the advertiser's product or intent to buy. We also present a methodology for click fraud detection that is unobtrusive to human users and differentiates Clickbot accesses from human accesses to a website through the use of Decoy Link Design Adaptation, the process of adapting a web page by creating decoy links. The methodology maximizes the percentage of bot sessions detected while minimizing the percentage of false positives by clustering the sessions based on their navigational patterns. The results of our investigation quantify the ability of web design principles that have been adapted using Decoy Link Design Adaptations (DLDA) to successfully identify bot sessions. We finally take the most effective DLDA, Shadow Links, and create a programmitic implementation. The programmtic implementation besides providing easy integration into websites for developers proves to be very fast and able to modify the pages on the fly. ÷**

I. Introduction

Online advertising has become a large source of revenue for countless businesses, but is prone to fraudulent activity due to vulnerabilities in many of the online advertising business models. Pay-per-click is a popular business model that allows exposure of advertisements to web users by agreeing to pay the publisher each time a user clicks on the advertisement. Fraudulent clicks on these advertisements can exhaust the advertisers' budgets, potentially upsetting the advertiser by diluting the value of their ad campaign while also upsetting the publisher if the attacks decrease their client-base. These fraudulent clicks, also known as click fraud, are often made by programs called Clickbots. Many methodologies have been, and are currently being, created to help combat against click fraud, but as these methodologies are created and evolve, the programs become more sophisticated at avoiding detection [1].

There are various methods for combating click fraud currently available. Some known methods utilize various statistical measures to lessen the amount of money lost by the advertiser. An example of this is the Pay-Per-Percentage (PPP) online advertising business model [2], an alternative to the Pay-Per-Click (PPC) business model that allows the advertiser to bid on a percentage of impressions, or page loads. However, the advertiser must still pay regardless of the outcome of the impression. Another well-known method is duplicate click detection where duplicate clicks on an advertisement are detected and discarded. Examples of this method are presented in Metwally's and Zhang's work in [3] and [4] respectively. Unfortunately, bots already exist that attempt to mimic human browsing with random clicks, and a detection methodology that takes access behavior into account is needed.

We present a methodology for differentiating human accesses from web robot accesses to a website using decoy links. Decoy Link Design Adaption(DLDA) adapts a web page by instantiating several decoy links that redirect to an identifying page. DLDA is unobtrusive to the human users, allowing them to seamlessly pass through by intuitively identifying and selecting the sole valid link.

In our evaluation of the effectiveness of various web design principles that have been adapted by DLDA, we first test for the best DLDA ased on its ability to avoid falsely identifying human users while still identifying the bots. We selected various web design principles for navigational structures such as Tag Clouds, Tabbed Toolbars and Shadow Links and applied DLDA to create an adapted web design principles capable of thwarting fraudulant activity while remaining unobtrusive to all human users. Our results indicate that DLDA applied to the Shadow Links design principle provides the best bot identification to human false positive ratio.

After determining that DLDA applied to the Shadow Links design principle provided the best results, we designed a programmitic implementation, in the form of an Apache module, to automatically modify the page with Shadow Link DLDA. The module was tested for its ability to detect bots, and the overhead it adds to modify a webpage. We found the Apache module was able to detect all clickbots when it output at least 10 links (1 valid, 9 invalid). The overhead of adapting a web page within the Apache module was determined by analyzing the amount of time it takes to adapt a page and the new page download size (including HTML, CSS, Javascript, and media elements). While the overhead proved to be negligible in most cases, the modified HTML increases considerably in size based on the number of decoy links required for each link.

II. Targets

We pit these design principles which have been adapted using DLDA against a specific type of web robot that does not have the full capabilities of a web browser. We believe this is reasonable based on the characteristics of bots currently in the wild such as ClickBot.A[1], publicly available requests for bots[5], and the time and effort necessary to create a bot that has the same capabilities of a web browser.

ClickBots have a specific set of requirements to fullfill to avoid detection. They must not produce an unreasonable Click-Through Rate(CTR), they must convince the advertiser they were human, and they must avoid clicking the same link continuosly. Cick Through Rate is defined as the percentage of clicks on an advertisement to the number of times it is

displayed. The average CTR for allowed web bots, such as crawlers, is less than 1%[6], and being consistantly above the average CTR helps advertising service providers such as Google identify fraudulent activity. Most bots currently being created will try to mimic basic human activity to fool an advertiser. For instance, if there are a substantial number of instances of user(s) clicking on a particular advertisement and not traversing past the landing page, an advertiser is likely to believe that their advertisement is being clicked on fraudulently. The majority of bots also spread their clicks to avoid detection of duplicate actions. A bot performing the same action too frequently is quickly identified by both the advertising service provider as well as the advertiser. Clickbot.A, for instance, is only allowed to click on a finite number of advertisements per instance.

III. Significance and Importance of Methodology

We believe our methodology is effective because of its ability to provide an unobtrusive user experience while thwarting fraudulant activity of any bot that does not have the same capabilities of a browser. To fully comprehend the difficulties bot creators face when attempting to provide the bot with the same capabilities of a browser, we will describe the most notable functions of a web browser emphasising the complexity of each.

Web browsers are intricate pieces of technology whose primary purpose is to display content from the web. While there are minute differences between the variety web browsers that are currently available, they all provide similar functionality such as parsing Hyper Text Markup Language (HTML), parsing and applying Cascading Style Sheets (CSS), parsing and running JavaScript with a JavaScript engine, the ability to traverse the Document Object Model (DOM) with JavaScript, and managing the layout with a layout engine. For the purposes of our research we programmed a variety of bots whose functionality ranges from having the ability to simply parse HTML to having the ability to translate CSS and execute JavaScript. The majority of the bots currently found online only have the capability to parse HTML which, as briefly stated before, is the largest factor behind the effectiveness of our methodology.

Hyper Text Markup Language (HTML) is the predominant markup language for web pages. Parsing HTML is the easiest capability to achieve because of the simple nature and structure of the language. A programmer does not have provide full parsing capabilities to a bot as all forms of HTML have the same syntax for anchor tags, the HTML tags that represent links. Providing the capability to parse and use anchor tags from HTML is the only necessary requirement for any bot as it allows for the traversal through a website where each click is technically a valid request.

Cascading Style Sheets (CSS) is a language used to describe the presentation of a web page written in HTML. CSS contains a complex selection syntax and several definitions for styles which can be applied to HTML elements. Bots must include functionality to properly translate the entirety of the CSS standard in order to reliably parse CSS, a task even most browsers are currently incapable of accomplishing[7]. Bot creators will also have to go above and beyond properly translating the CSS standard to fully understand all possible CSS variations for a website due to countless browser-specific extensions and hacks. While the majority of websites apply simplistic CSS that can be easily parsed by a cleverly programmed bot, the slightest tweak may upset the design and disrupt the bots attempt at fraudulent behavior [ref to how many people use CSS, and to what degree]. If and when the bot successfully parses the CSS it must then be applied to the HTML document, a difficult process due to the complexity of the selection rules within the current CSS selector syntax style.

JavaScript is the default scripting language primarily used client-side for the development of dynamic web pages. A programmer wishing to provide a bot with the ability to parse and execute JavaScript must not only be able to decipher the JavaScript, but also infer the significance of the location of any JavaScript embedded within the HTML due to JavaScript sequential nature. In other words, it is possible to parse and display HTML before executing JavaScript allowing any scripts to run at different stages of page load. If and when the bot successfully parses and understands the web page's JavaScript, it must then execute the JavaScript using a JavaScript engine. JavaScript engines are publicly available and easily obtainable, and most, such as Mozilla's Rhino, are provided for free with sufficient documentation. While these JavaScript engines allow bot creators to provide JavaScript execution capabilities to their bots, the engines themselves provide no functionality to correctly execute JavaScript in accordance to its location within the HTML. The bot creators must also provide for a JavaScript accessible Document Object Model (DOM) that allows JavaScript to modify the web page directly.

If and when the bot creators have provided all the functionality previously described they must then implement a layout engine, a library which decides how to display the HTML and CSS styles on screen. A bot creator may not need a full layout engine, but in order to have a chance at defeating our technique they will be forced to provide basic layout functionality as well as capabilities for parsing HTML, parsing and applying CSS to HTML, parsing and applying JavaScript around and within the HTML, and executing the Javascript. We believe our methodology is effective against the majority of the bots currently available because of this.

A. *Our Bots*

We created three different web bots based on the characteristics of bots currently found in the wild and thus are representative of the types of bots we are trying to identify and combat. The bots function as follows:

- **Simple Bot** — has the capability to select either all links on a page or random links on a page.
- **Keyword Bot** — has the capability to select all links that contain predefined keywords in the anchor text of the HTML.
- **CSSJS Bot** — has the capability to execute JavaScript, translate CSS, and select links based on their visibility to the user.

TABLE I
ABILITY OF BOTS AND BROWSER

	Replay Bot	Simple	Keyword	CSSJS	Browser
HTML Parsing		x	x	x	x
CSS Parsing				x	x
CSS Application				1/2	x
JS Parsing				x	x
JS Engine				x	x
JS DOM				x	x
Layout Engine					x

IV. User Interaction

DLDA is an almost entirely unobtrusive methodology. When properly installed, the developer is not required to manually alte any HTML, CSS, or JavaScript to have the protection provided by DLDA, and the visitors to the developer's website have little trouble navigating through the modified website.

A. Website User

A website user, or visitor to the website, must not be aware of the fact that they are constantly being tested by DLDA. We took the visitor's perspective into consideration in order to accomplish this goal. As we stated before, web browsers rely on libraries and functionality such as CSS parsing and layout engines. Because of this we are provided with a great deal of flexibility with how we present a modified web page since websites are most commonly viewed through graphical web browsers.

There are two particular scenarios to consider when providing webpages altered by DLDA to the website user. The first is when DLDA is applied and the decoy links are visible on the web page. By making the decoy links visible to the human user it is necessary to create visual cues that the human user will intuitively perceive and understand so that the valid link is obviously the valid link and the decoy links are obviously bad links. By applying CSS we can make the valid link larger, a different color, or be placed in a special position within the web page. While these options help create intuitive designs, and the results have shown that human users do select the most obvious link, the decoy links are still available and thus are sometimes selected regardless of how obvious it may be to avoid them. The second scenario involves applying DLDA to a web page where the decoy links exist within the web page but are not visible to the human user. The decoy links can easily be made invisible by generating CSS that will position the link off the page, set the decoy links' opacity to zero, or set the decoy links' color to that of the background.

B. Website Developer

A website developer, or a person who creates and modifies the web pages within a website, are already preoccupied with creating and managing visually asthetic, cross-browser compatible designs and typically do not have the time or knowledge to create security modules to protect against fraudulent activity. If necessary, developer's are able to include third party security modules, such as reCaptcha, to protect against such behavior, but these modules are obtrusive to the human user and are becoming increasingly more complicated as bots evolve and learn to compromise the security offered by third party applications [8], [9].

We created a programmatic application of DLDA so that the process of generating decoy links occurs dynamically without requiring human approval or support. The DLDA Apache module parses each request and generates a predefined number of decoy links per valid link on every page. The need for the developer to constantly modify each individual page to conform to some DLDA no longer exists since the dynamic nature of the module makes the methodology unobtrusive to the developer as well.

V. Decoy Link Design Adaptation

We adapted several popular navigational web design principles that we believe lend themselves to intuitive user interaction. That is to say, these well known and often used web design principles, such as Tag Clouds, create a clear path through the website to the human user due to the link structure's easily identifiable nature. We also adapted simple navigational link structures because of their ability to direct the user's selection. Most web design principles require no special consideration from the developer when designing a web page, but certain design principles, such as Tag Clouds, are ultimately infeasible to implement in a programmatic application because of their hefty space requirements on any given page.

We applied DLDA to each web design principle by changing the link which was most easily recognizable by a human user to the valid link, and the remaining links to decoy links. The desired result being that the decoy links would lead the bots to visit an identifying page while the valid link would lead the human users to continue forward through the site. Below is a list of the web design principles and Decoy Link Design Adaptations used in our study.

- **Basic** — The Basic web design principle leaves any and all links unmodified with no DLDA used.
- **CSS** — The CSS web design principle styles all links on a page with CSS. This may interfere with bots incapable of parsing CSS.
- **Multiple Link** — The Multiple Link web design principle positions multiple decoy links off the visible page and valid links remain in their original, visible position. The decoy links contained text mimicing that of the valid links.
- **Tag Cloud** — The Tag Cloud web design principle created a Tag Cloud of decoy links each with identical text where the largest link was the only valid link.
- **Tab Links** — The Tabbed Menu web design principle created a tabbed menu where decoy links belonged to tabs containing identical text and color, and the valid link belonged to a tab with a different color.
- **Shadow Link** — The Shadow Link web design principle created multiple, invisible decoy links containing identical text to the valid link. These decoy links, however, were positioned with the same x, y coordinates as the valid link, but below the valid link itself.

A. Determing the Effectiveness of each DLDA

For the purposes of our research, effectiveness is measured as the percentage of sessions identified as those created by bots as compared to the percentage of sessions created by human users that were falsely identified as bots. To measure the effectiveness of each DLDA, we created a mock webstore using OSCommerce as well as six landing pages each containing content adapted using the different DLDA. We requested that a group of approximately twenty people traverse past each of the six landing pages independently and into the webstore to perform a series of actions, such as selecting a specific product, browsing or searching for various products, or purchasing an available product. We then programmed the three Clickbots previously described to mimic the behavior of the human users to the best of their available capabilities. By creating our own landing pages adapated with DLDA and our own mock webstore, we have a controlled environment where it was known beforehand which sessions wer created by Clickbots and which were created by human users.

To obtain the effectiveness measures for each DLDA, we analyzed the web server access log, a file containing the HTTP requests made by both the Clickbots and the humna users. We sessionized the access log, isolating the requests belonging to each particular IP into groups. These groups of requests were then further isolated based on a 30 minute interval, where a new session begins only if 30 minutes have elapsed between the IPs last two requests. Sessions containing a request for the identifying page were considered to be comrpised of requests made by a bot, and sessions not containing a request for the identifying page were considered to be comprised of requests made by a human users.

Once the access log was sessionized, we measured the effectiveness of each DLDA by isolating sessions created by both the bots and the human users that originated from each of the six different landing pages. Since we knew which IPs belonged to our bots prior to our investigation, we were able to determine which sessions were properly and improperly classified as those created by one of our bots, and those which were properly and improperly classified as those created by a human user.

Figure 1 shows the percentage of sessions containing a request for the identifying page per user type per landing page. We can infer from the information in Figure 1 that the landing page adapted using the Shadow Link DLDA has the highest effectiveness measure (100% to 0%). Due to its ability to successfully identify all sessions created by the bots while having a relatively low percentage of false positives, we believe that the Shadow Link DLDA is the most effective DLDA.

The effectiveness measures of the Shadow Link DLDA (100% to 0%) and the Multiple Link DLDA (75% to 0%) leads us to believe that having visible or invisible multiple links will dramatically increase the chances of successfully identifying sessions created by Clickbots. A desirable side-effect of the Shadow Link DLDA is that the human users were never aware of the fact that the landing page they visited had been adapted. From their perspective, the page adapted using the Shadow Link DLDA appeared no different than the Basic landing page.

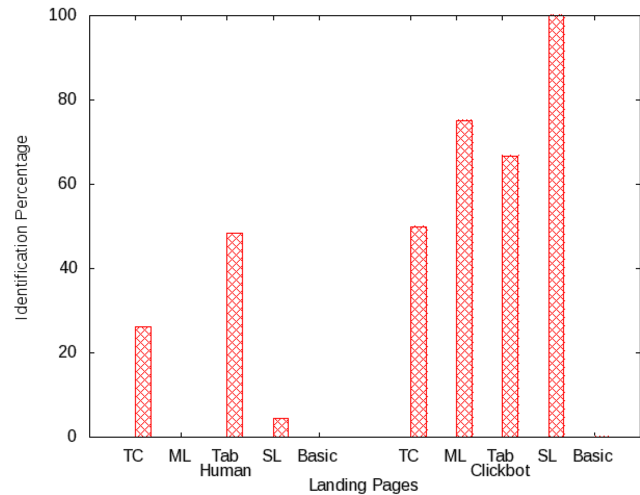

Fig. 1. Percentage of sessions created by both human users and clickbots containing the identifying page, per landing page (TC - Tag Cloud, ML - Multiple Link, SL - Shadow Link).

Furthermore, the low effectiveness measure of the landing page adapted using the Tab DLDA (x% to x%) leads us to believe that placing both the valid link and the decoy links on equal footing makes it more difficult for the human users to determine which link is actually valid. While the tab containing the valid link on the landing page modified using the Tab DLDA was a different color, the visual cue was not strong enough to direct 100% of the human users through to the webstore. We believe that while the Tab DLDA produced a decent effectiveness measure, placing the tabs containing the valid link and decoy links on equal footing confused an unacceptable percentage of human users for the Tab DLDA to be considered feasible for general use.

VI. Programmatic Implementation of the Shadow Links DLDA

We created a programmatic implementation of the most effective DLDA, the Shadow Links DLDA, to allow developers to easily integrate DLDA into their own website. Due to the infeasibility of programming a compiler capable of parsing the plethora web programming languages currently available, we decided to create a program that modifies the resultant HTML generated by a web server after managing a user's request.

While altering the web server's resultant HTML can be accomplished by either implementing a proxy or a webserver module, programming and intergrating a webserver module using the webserver's API is arguably more advantageous than using a proxy. Implementing DLDA with a proxy comes with the disadvantages of managing network sockets and potentially appearing like a man-in-the-middle, conspicuous proxy server. A proxy would also be more troublesome for a developer to download, configure, deploy and manage, while webserver modules are relatively easy to install, configure, and manage. One of the most common webservers deployed today is the

Apache HTTP Web Server. Given the advantages and disadvantges of proxy servers and webserver module, we decided to program a webserver module for Apache that adapted the webserver's resultant HTML. The webserver module contains the following components:

- **Shadow Links Filter** — In the Apache webserver, a filter is a process that is applied to data that is sent or received by the server. Data sent by clients to the server is processed by input filters while data sent by the server to the client is processed by output filters. Multiple filters can be applied to the data, and the order of the filters can be explicitly specified [10]. The Shadow Links filter is an output filter explicitely placed at the end of the filter list. It modifies the page by parsing the resultant HTML and adding a variable number of decoy links to every valid link within the HTML. Unlike the Shadow Links DLDA previously described, however, the Shadow Links filter makes all valid and invalid links visible. The valid link is not placed at the beginning of the list of links within the HTML, but rather randomly between two of the variable number of decoy links generated by the filter. The valid link is moved to the beginning of the list by obfuscated JavaScript that has also been placed within the HTML by the filter. This ensures that when a human user views the information through a web browser, the web browser's layout manager will place the valid link on top of the variable number of decoy links allowing for an unobtrusive user experience. The decoy links are randomly chosen from a supplied file of decoy links allowing the developer to configure decoy URLs that are indistinguishable from the URLs found within the website.
- **Logging** — Similar to the web server's capability to log user requests in the web server access log, this component logs any incoming requests from a page modified by the filter.
- **Log Sessionizer** — The Log Sessionizer parses the output from the Logging component and creates a sessionized form of the log. The sessionized log is programmatically examined to determine if the session was created by a bot, or if the session was created by a human who managed to request information from the server using a decoy link. This determination is based on the number of valid links and decoy links selected by the user. If the user is above a certain number of selected decoy links, they are identified as a bot. This provides a separate, succint log of requests made entirely by bots for the developer to use as evidence when confronting their online advertsing agency about undetected fraudulent activity, or as information to supply to the server's black list or firewall.

The module also contains a few configuration options.

- **Number of Shadow Links** — This configuration option determines how many decoy links are placed within the resultant HTML as per each valid link.
- **Decoy Links File** — This configuration options determines which file contains the decoy links to be inserted into the resultant HTML.
- **Shadow Links Log File** — This specifies the log file that contains the requests made by human users and bots through a page modified by the filter.

The programmatic implementation of DLDA as a web server filter offers developers the advantages of the Shadow Links DLDA while remaining unobtrusive, configurable, and manageable. The developer need only include the filter directive within the web server's configuration file, provide the configuration options, and restart the web server.

VII. Evaluation

Here we evaluate the effect of different settings of the module to determine their effect on false positives and false negatives. We also evaluate the effect of settings and web page characteristics on output time and size.

In testing our module, we used sessions as in the preliminary evaluation. The importance of sessions cannot be overstated. Sessions allow us to make up for possibilites of single runs of bots avoiding decoy links. They also allow us to group differentiate whether a human made a mistake in visiting a decoy link. Using sessions, we can vary how many decoy links must be clicked to be considered a bot access. This allows a greater flexibility in decreasing the false positive percentages for humans. It also increases the praticality of the clustering out bots that may have avoided decoy links.

Our evaluation was carried out on a single machine running Fedora 9 with an Intel Core 2 Duo, containing 2 cores sharing 5MB of cache, and 3GB of memory. The machine was running the Apache HTTP server version 2.2.9. The DLDA module was compiled using GCC with no optimization flags.

A. *Number of DLDA Links*

The number of links output by DLDA has an affect on the chance of a bot clicking the identifying link. We ran each of our three bots 50 times each against different numbers of output DLDA links. Figure 2 shows how well they faired against the varying number of links.

We would expect to see that the more links on the page the more likely a bot is to identify itself. Looking at Figure 2(a) and Figure 2(b), we can see this is usually the case. Some bots went undetected with only 2 links, but as we increase the number of links that dropped to 0% or close 0%. However, at around 50 links something counter intuitive happens, and the number of false negatives goes up. Investigating why this occured leads to understand that the more links on the page the more time a bot must spend parsing all the links. In the cases of the Keyword and CSSJS bots, they have to actively look at each link to decide whether they should visit. This takes them a great deal of time as more links are put on the page.

We next investigated ways to correct for inconsitencies in the ability of a bot process links at a constant pace. We introduced sessions into the mix to smooth out bot inconsitencies. A session is set of accesses from a user, in our case an IP Address, that starts a certain period of time after another session ends or with no other session on record. We manipulated the time difference necessary for session to be

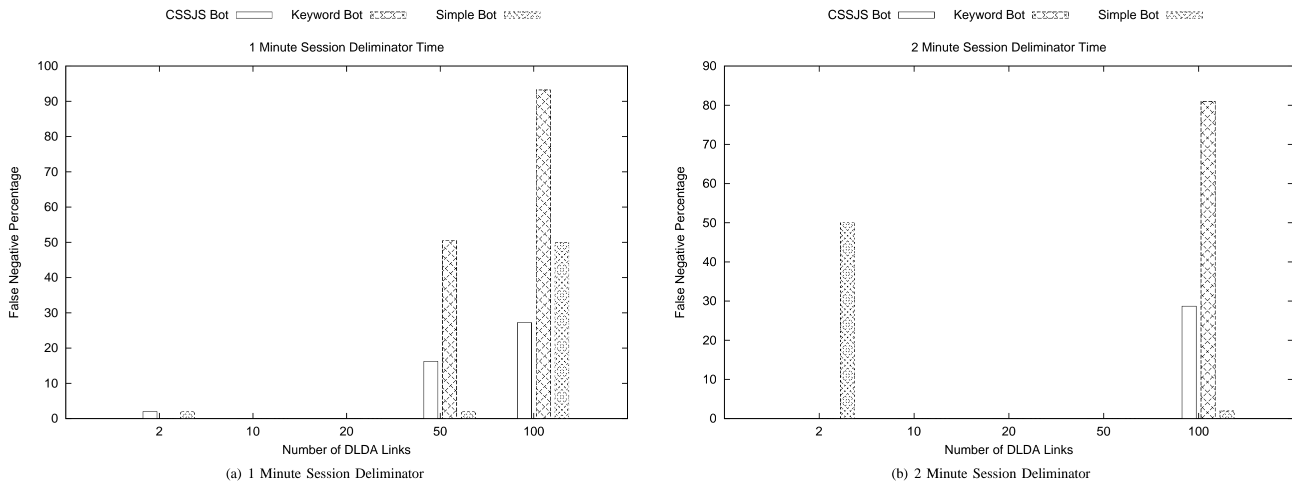

Fig. 2. Percentage of undetected bots (False Negatives) based on the number of links and session deliminator time.

considered terminated and a new session to start. Looking back at Figure 2, we can see that as we increase the time necessary for session termination the false negatives go down. At five minutes, we were unable to get any false positives. This reason behind this is because increasing the time to terminate a session means that there are fewer sessions.

The number of links output by DLDA has an on affect how fast the page can be modified and the final output size. The page modified was the main OSCommerce mock store page which was dymanically generated but always contained the same number of links. We varied the number of links output to 10, 20, 50, and 100 links per original link on the page.

Figure 3(a) presents the effect of this on total generation time of a dynamically generated page. We can see that the overall time to generate a page increases linearly with the number of links output. Looking at 10 decoy links output for each real link on a page shows only a 11ms overhead. In this case, the HTML generation is a majority of time to generate the final DLDA page. The situation is similar with 20 decoy links only adding close 20ms overhead. Looking at the case of adding 100 links for each original link, we see the time to generate the page nearly doubles making it inadvisable to use this many links.

The change in the final size of the output can be seen in Figure 3(b). The increase is page size is linear, but with a greater slope than the total download time. Given the speed of most Internet connections today, adding 10 or 20 links for every link on a page is not a hinderence for a dynamically generated page. Increasing the number of output links past 20 can cause dramatic increases in the size of the file. It is therefore prudent to investigate and narrow the range of output DLDA links based on targeted user connection speeds.

When using DLDA it is important that an attacker be unable to distinguish a decoy link from a real link by an easy means. One way to distinguish a decoy link is to have it appear too many times on a single page. We prevent this by allowing the specification of any number of links in the Decoy Links file. Given that any number of links can be specified, we must know how the DLDA module handles different numbers of links.

We tested how size and generation speed of a page were affected by varying the number of links in the Decoy Links file. The page modified was the same OSCommerce mock store page modified testing the effects of the number of output links. This time, however, the number of links was held constant.

Figure 4 shows the effect of varying the number links in the Decoy Links file with respect to the generation speed of the page. We can see that generation is fairly constant. The biggest block of generation time for any number of links in the DLDA Links file is the time to generate the overall page. Figure 4(b) shows the page size with varying numbers of links in the Decoy Links file. The page maintains a fairly constant size as expected because the number of output links is still the same. The slight size variation, only by a few 100 bytes, based on the length of a decoy link chosen at the time to output on the page.

Given our tests, we can recommend that 10 or 20 decoy links be used for each original link on a page. Since the Decoy Links file has no bearing the operation of the module, a site administrator should use as many links as makes sense for the site being protected. If the website is being targeted to users of slower connections, 2 decoy links can be used, but at a cost of increased false negatives.

B. *Number of Page Links*

In the last section, DLDA was evaluated based on the modification of its configuration. Here we evaluate how well DLDA performs by varying of the page to be modified and leaving the configuration the same. DLDA was configured to output 10 decoy links for each individual link and use a Decoy Links file with 100 decoy links.

Since DLDA only modifies the links of a page, anchor tags with href attributes, we based our evualtion on this. We created static HTML pages with a varying number of links starting at 10 links up to 100 links increasing the number of links for each page by 10 each time. DLDA was tested for how well it performed on each of these different pages in terms of generation speed and size output.

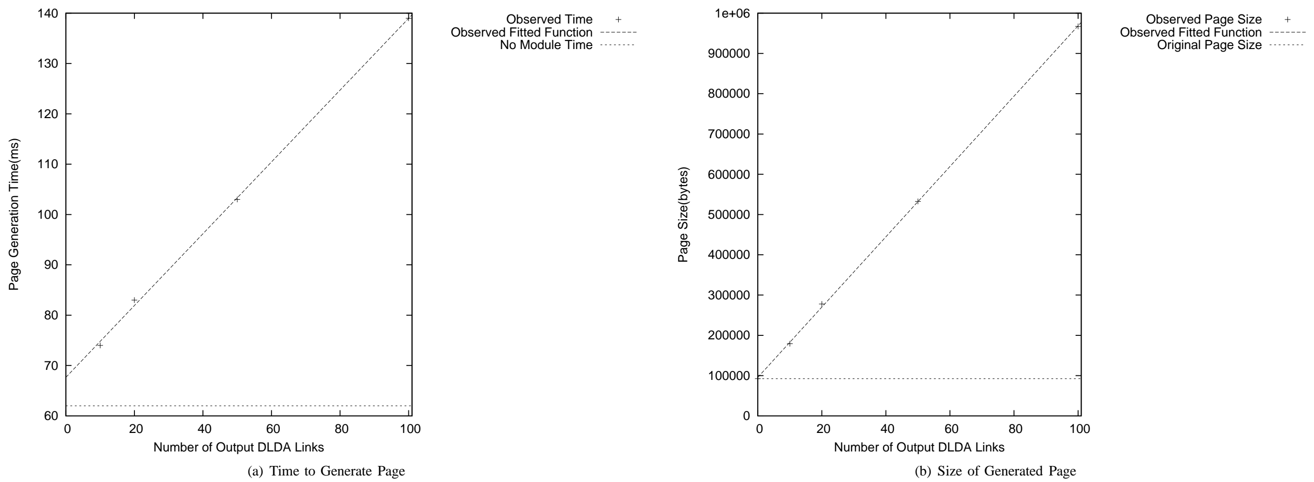

Fig. 3. Effect of varying the number of Shadow Links on page generation time and page size.

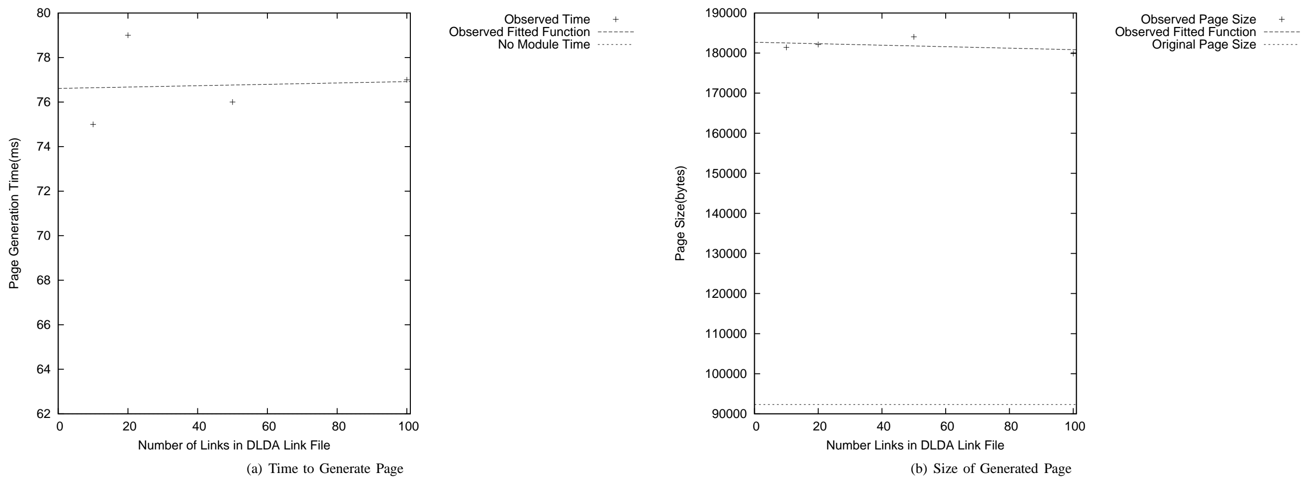

Fig. 4. Effect of changing the number of links in the Link File on page generation time and size.

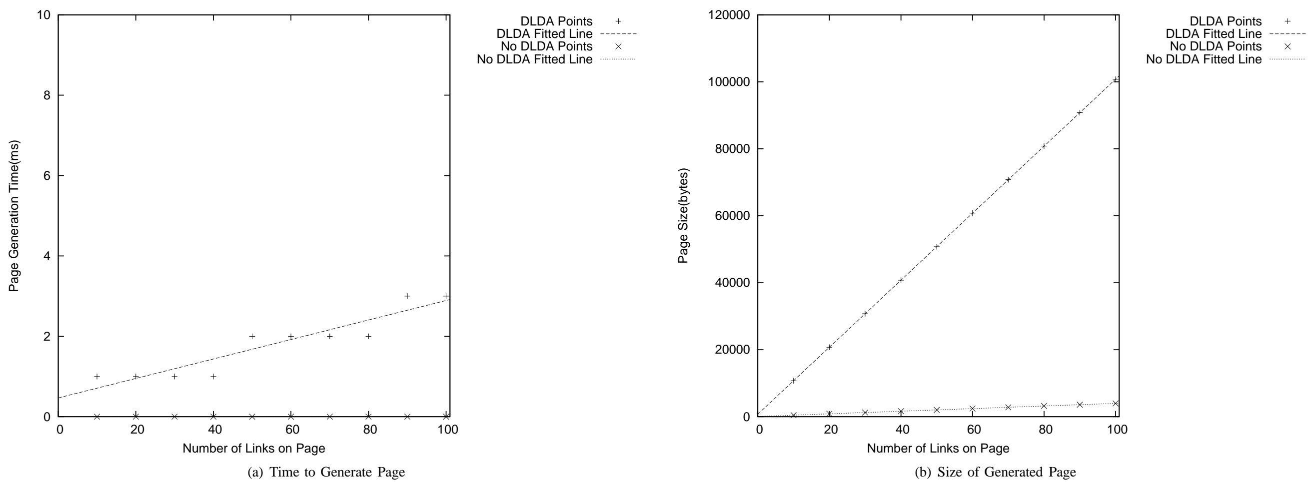

Fig. 5. Effect the number of links on a web page has on page generation time and page size. Ten DLDA Links were output for each original link and 100 links were in the DLDA Links file.

Figure 5(a) shows the variation of DLDA generation time with respect to the number of links on the static HTML page. We can see that server can generate the HTML page, send the static HTML to the client, in between 0 and 1ms. Adding DLDA into the mix produces overhead. The total overhead produced is not more than 2ms for any page generation.

Figure 5(b) shows how the original page size is changed by DLDA based on the number links in the page. We can see a linear growth in the page size with DLDA enabled. This again begs the consideration of the person configuring DLDA. The number of links output by DLDA should be adjusted to meet their users needs in terms of download speed.

VIII. Discussion

Fraudlent activity on websites, particularly click fraud on web based advertisments, is becoming an increasing problem. The problem is compounded with webrobots conducting a significant amount of click fraud making it scalable and economically viable. We introduced a new technology, Decoy Link Design Adapatation, to combat fraud caused by webrobots that parse and visit links found in HTML. DLDA adapts the pages of a website by replacing the original link with a set of links one of which is valid with the rest invalid. Once DLDA has been applied to a page, Clickbots follow the invalid decoy links identifying themselves as committing fraudulent activity.

Investigating our proposed DLDA techniques, we found the Shadow Link DLDA was the most effective. It produced few false positives while detecting all the bot accesses. We implemented the Shadow Link DLDA as an Apache Web Server module to provide simple, configurable way to add DLDA protection to any website.

The Shadow Links server module showed great performance at detecting the targeted webrobots. With the module replacing each link on the page with 2 links, one valid and one invalid link, it was able to detect most bot accesses correctly. Increasing the number of output links to 10 or 20, allowed the module to detect all the bots. The module proved quite capable of realtime web page modification; adding 10 and 20 links only increases the page generation time by small amounts 11ms and 20ms, respectively.

The developers ability to provide the decoy links through the Decoy Links file provides a way stop a simple frequency analysis break of DLDA. If we had 10 links on a page and 10 links in the Decoy Links file while outputting 10 links for each original link, the final output would contain enough duplicate decoy links to be noticable. Countering this requires the developer to put a large number of links in the Decoy Links file. Putting a large number of valid looking links in a file can be time consuming; as such, we plan to provide a URL re-writing implementation that produces random URLs for every URL on an output page.

The current version of Shadow Links raises the bar for webrobot writers requiring them to implement some of the web browser's ability to provide interaction of the JavaScript and CSS code of a webpage. If this time consuming undertaking is completed, the Shadow Link DLDA can be broken. The Shadow Link DLDA can be extended to mitigate this through the random output of CAPTCHA like test pages using a form of DLDA. Outputing these types of test pages introduces a level of obtrusiveness to the user, but strengthens the protection provided by the Shadow Link DLDA because webbots will require artificial intelligence capabilities as well.

The Shadow Link DLDA module can be modified to output a randomly generated DLDA CAPTCHA test page at specific times, randomly or on some predifined signal. A human user visiting a test page can see all links, valid and invalid, and is asked to complete a simple problem that allows them to select the valid link. A simple version of this problem would be to click the image with the green dots instead of the images with dots of another color.

The DLDA CAPTCHA test page requires original URL protected by the test page is retrievable in the URL leading to the test page, nor can the original URL or invalid URLs retrievable on the test page be determined without answering the posed question. However, to make the test page generic, it must be able to deduce the a valid or invalid from information in itself alone. This can accomplished by encrypting the original using a random salt, preventing correlation of real or fake URLS, and appending it as a query string. When a human clicks the correct link, a decryption page decrypts the original URL and redirects the user to that URL. However, when a bot selects the incorrect URL, the decryption page will mark that request as a bot access.

The Shadow Link DLDA module while usuable and secure modifies webpages differently each time it processes the page for output. This could have a serious effect on webpage caching. Analyzing the modifications made by the module shows the effect is minimal at best.

The Shadow Link DLDA only modifies the links on the HTML generated from a script or read from a static HTML page. Therefore, caching of media related items (pictures, videos, flash content), CSS, and Javascript included in the page are not affected. These being the largest parts in terms of size of a webpage a large portion of content remains easily cacheable.

While it easy to see how DLDA can affect caching, caching will affect how well DLDA work as well. DLDA relys on modification of the webpage to provide its protection. Getting the optimal protection requires running the modification each time a page is requested. This helps to keep someone from analyzing which link is actually the valid link in the group of links containing the valid and invalid links. While it is best to modify each for each request, a small amount of time cached for a DLDA should not be a problem.

IX. Related Work

CAPTCHA is a common method for blocking web bots from using an online server like forums or webmail. It's success at these tasks suggest it could also be applied to click fraud much the same way as DLDA. With CAPTCHA, a user is presented with a picture of text and responds by entering that text into a text field on the web page. This is a challenge-response test most humans can pass and current computer

programs cannot. With computers unable to solve CAPTCHA, it provides click fraud deterrence by barring bots beyond the landing page of a websites ad campaign due to their lack of AI [11]. In [12] Chow, et. al. propose a Clickable CAPTCHA that is more friendly for mobile computing devices like cellphones. In Clickable CAPTCHAs, the user clicks on the pictures of words in set of pictures filled with characters. The significant difference between DLDA and CAPTCHA is in the perception of the challenge. The user of a CAPTCHA is presented with a challenge they must actively solve. The DLDA user passively solves the challenge by inherently understanding which link is valid. Thus, CAPTCHA creates an obtrusive inconvenience for the user, especially those users who are human and cannot decipher the CAPTCHA thus barring them from passing as well. As bots become better at compromising challenge-response tests, CAPTCHA evolves and becomes even more inconvenient and harder to solve to the human user.

In a duplicate detection approach to click fraud[4], Zhang and Guan present a space and time efficient algorithm to look through Clickstreams, which are similar to web server access logs except in real time, and is agnostic to the advertising business model. Their implementation of duplicate detection, called timing Bloom filters (TBF), allows for the inspection of clicks over a specified time for duplicates making it harder for bots to cause monetary damage by discarding clicks that occur in rapid succession. However, duplicate detection does not differentiate between a bot and a human. With sophisticated Clickbots that act like humans, publishers may very well start discarding valid clicks on advertisements using this model.

References

[1] N. Daswani and M. Stoppelman, "The anatomy of clickbot.a," in *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*. Berkeley, CA, USA: USENIX Association, 2007, pp. 11–11.

[2] J. Goodman, "Pay-per-percentage of impressions: An advertising model that is highly robust to fraud," in *ACM E-Commerce Workshop on Sponsered Search Auctions*, 2005.

[3] A. Metwally, D. Agrawal, and A. E. Abbadi, "Duplicate detection in click streams," in *In Proceedings of the 14th WWW International World Wide Web Conference*. ACM Press, 2005, pp. 12–21.

[4] L. Zhang and Y. Guan, "Detecting click fraud in pay-per-click streams of online advertising networks," in *Distributed Computing Systems, 2008. ICDCS '08.*, 2008, pp. 77–84.

[5] "Adsense click-bot asap," http://www.scriptlance.com/projects/1236408116.shtml.

[6] eMarketer, "The latest ad click count," http://www.emarketer.com/Article.aspx?R=1006969.

[7] QuirksMode, "Css - contents and compatibility," http://www.quirksmode.org/css/contents.html.

[8] P. Lamere, "moot wins, time inc. loses," http://musicmachinery.com/2009/04/27/moot-wins-time-inc-loses/.

[9] *CAPTCHA Killer - Automated CAPTCHA Bypass*, CAPTCHA Killer, http://www.captchakiller.com/.

[10] *Filters*, Apache Software Foundation, http://httpd.apache.org/docs/2.0/filter.html.

[11] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," in *In Proceedings of Eurocrypt*. Springer-Verlag, 2003, pp. 294–311.

[12] R. Chow, P. Golle, M. Jakobsson, L. Wang, and X. Wang, "Making captchas clickable," in *HotMobile '08: Proceedings of the 9th workshop on Mobile computing systems and applications*. New York, NY, USA: ACM, 2008, pp. 91–94.