Computer Networks 54 (2010) 2182–2198

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

An automatic HTTP cookie management system

Chuan Yue, Mengjun Xie, Haining Wang *

Department of Computer Science, The College of William and Mary, Williamsburg, VA 23187, United States

ARTICLE INFO

Article history:
Received 2 September 2009
Received in revised form 25 February 2010
Accepted 15 March 2010
Available online 20 March 2010
Responsible Editor: Neuman de Souza

Keywords:
Web
HTTP cookie
Security
Privacy
Web browsing

ABSTRACT

HTTP cookies have been widely used for maintaining session states, personalizing, authenticating, and tracking user behaviors. Despite their importance and usefulness, cookies have raised public concerns on Internet privacy because they can be exploited by third-parties to track user behaviors and build user profiles. In addition, stolen cookies may also incur severe security problems. However, current Web browsers lack secure and convenient mechanisms for cookie management. A cookie management scheme, which is easy-to-use and has minimal privacy risk, is in great demand; but designing such a scheme is a challenge. In this paper, we conduct a large scale HTTP cookie measurement and introduce CookiePicker, a system that can automatically validate the usefulness of cookies from a Web site and set the cookie usage permission on behalf of users. CookiePicker helps users achieve the maximum benefit brought by cookies, while minimizing the possible privacy and security risks. We implement CookiePicker as an extension to Firefox Web browser, and obtain promising results in the experiments.

1. Introduction

HTTP Cookies, also known as Web cookies or just cookies, are small parcels of text sent by a server to a web browser and then sent back unchanged by the browser if it accesses that server again [1]. Cookies are originally designed to carry information between servers and browsers so that a stateful session can be maintained within the stateless HTTP protocol. For example, online shopping web sites use cookies to keep track of a user's shopping basket. Cookies make web applications much easier to write, and thereby have gained a wide range of usage since debut in 1995. In addition to maintaining session states, cookies have also been widely used for personalizing, authenticating, and tracking user behaviors.

Despite their importance and usefulness, cookies have been of major concern for privacy. As pointed out by Kristol in [2], the ability to monitor browsing habits, and possibly to associate what you have looked at with who you are, is the heart of the privacy concern that cookies raise. For example, a lawsuit alleged that DoubleClick Inc. used cookies to collect web users' personal information without their consent [3]. Moreover, vulnerabilities of web applications or web browsers can be exploited by attackers to steal cookies directly, leading to severe security and privacy problems [4–7].

As the general public has become more aware of cookie privacy issues, a few privacy options have been introduced into Web browsers to allow users to define detailed policies for cookie usage either before or during visiting a Web site. However, these privacy options are far from enough for users to fully utilize the convenience brought by cookies while limiting the possible privacy and security risks. What makes it even worse is that most users do not have a good understanding of cookies and often misuse or ignore these privacy options [8].

Using cookies can be both beneficial and harmful. The ideal cookie-usage decision for a user is to enable and store useful cookies, but disable and delete harmful cookies. It has long been a challenge to design effective cookie management schemes that can help users make the ideal cookie-usage decision. On one hand, determining whether

* Corresponding author. Tel.: +1 757 221 3457.
E-mail addresses: cyue@cs.wm.edu (C. Yue), mjxie@cs.wm.edu (M. Xie), hnw@cs.wm.edu (H. Wang).

1389-1286/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2010.03.006

some cookies are harmful is almost impossible, because very few web sites inform users how they use cookies. Platform for Privacy Preferences Project (P3P) [9] enables web sites to express their privacy practices but its usage is too low to be a practical solution. On the other hand, determining whether some cookies are useful is possible, because a user can perceive inconvenience or web page differences if some useful cookies are disabled. For instance, if some cookies are disabled, online shopping may be blocked or preference setting cannot take into effect. However, current web browsers only provide a method, which asks questions and prompts options to users, for making decision on each incoming cookie. Such a method is costly [10] and very inconvenient to users.

In this paper, we first conduct a large scale cookie measurement for investigating the current cookie usage on various web sites. Our major measurement findings show that the pervasive usage of persistent cookies and their very long lifetimes clearly highlight the demand for removing useless persistent cookies to reduce the potential privacy and security risks. Then we present *CookiePicker*, a system that automatically makes cookie usage decisions on behalf of a web user. CookiePicker enhances the cookie management for a web site by using two processes: a training process to mark cookie usefulness and a tuning process to recover possible errors. CookiePicker uses two complementary algorithms to effectively detect HTML page difference online, and we believe that these two algorithms have the potential to be used by other online tools and applications. Based on the two HTML page difference detection algorithms, CookiePicker identifies those cookies that cause perceivable changes on a web page as useful, while simply classifying the rest as useless. Subsequently, CookiePicker enables useful cookies but disables useless cookies. All the tasks are performed without user involvement or even notice.

Although it is debatable whether defining useful cookies as those that lead to perceivable changes in web pages retrieved is the best choice, so far this definition is the most reasonable measure at the browser side and it is also used in [11]. The reasons mainly lie in that very few web sites tell users the intention of their cookie usage, P3P usage is still very low, and many web sites use cookies indiscriminately [12].

We implement CookiePicker as a Firefox web browser extension, and validate its efficacy through live experiments over a variety of web sites. Our experimental results demonstrate the distinct features of CookiePicker, including (1) fully automatic decision making, (2) high accuracy on decision making, and (3) very low running overhead.

The remainder of this paper is structured as follows. Section 2 presents the background of cookies. Section 3 shows our cookie measurement results. Section 4 describes the design of CookiePicker. Section 5 details the two HTML page difference detection algorithms used by CookiePicker. Section 6 presents the implementation of CookiePicker and its performance evaluation. Section 7 discusses the potential evasions against CookiePicker as well as some concerns about using CookiePicker. Section 8 surveys related work, and finally, Section 9 concludes the paper.

2. Background

HTTP cookies allow an HTTP-based service to create stateful sessions that persist across multiple HTTP transactions [13]. When a server receives an HTTP request from a client, the server may include one or more `Set-Cookie` headers in its response to the client. The client interprets the `Set-Cookie` response headers and accepts those cookies that do not violate its privacy and security rules. Later on, when the client sends a new request to the original server, it will use the `Cookie` header to carry the cookies with the request [14].

In the `Set-Cookie` response header, each cookie begins with a `NAME=VALUE` pair, followed by zero or more semicolon-separated attribute-value pairs. The `NAME=VALUE` pair contains the state information that a server attempts to store at the client side. The optional attributes `Domain` and `Path` specify the destination domain and the targeted URL path for a cookie. The optional attribute `Max-Age` determines the lifetime of a cookie and a client should discard the cookie after its lifetime expires.

In general, there are two different ways to classify cookies. Based on the origin and destination, cookies can be classified into first-party cookies, which are created by the web site we are currently visiting; and third-party cookies, which are created by a web site other than the one we are currently visiting. Based on lifetime, cookies can be classified into session cookies, which have zero lifetime and are stored in memory and deleted after the close of the web browser; and persistent cookies, which have non-zero lifetime and are stored on a hard disk until they expire or are deleted by a user. A recent extensive investigation of the use of first-party, third-party, session, and persistent cookies was carried out by Tappenden and Miller [15].

Third-party cookies bring almost no benefit to web users and have long been recognized as a major threat to user privacy since 1996 [2]. Therefore, almost all the popular web browsers, such as Microsoft Internet Explorer and Mozilla Firefox, provide users with the privacy options to disable third-party cookies. Although disabling third-party cookies is a very good start to address privacy concerns, it only limits the profiling of users from third-party cookies [2], but cannot prevent the profiling of users from first-party cookies.

First-party cookies can be either session cookies or persistent cookies. First-party session cookies are widely used for maintaining session states, and pose relatively low privacy or security threats to users due to their short lifetime. Therefore, it is quite reasonable for a user to enable first-party session cookies.

First-party persistent cookies, however, are double-edged swords. As we will show in Section 3, first-party persistent cookies could stay on a user's disk for a few years if not removed. Some cookies perform useful roles such as setting preferences. Some cookies, however, provide no benefit but pose serious privacy and security risks to users. For instance, first-party persistent cookies can be used to track the user activity over time by the original web site, proxies, or even third-party services. Moreover, first-party persistent cookies could be stolen or manipulated by three

Fig. 1. Internet Explorer 7 advanced privacy settings.

kinds of long-standing attacks: (1) cross-site scripting (XSS) attacks that exploit web applications' vulnerabilities [16,6,17], (2) attacks that exploit web browser vulnerabilities [18,19,7], and (3) attacks that could directly steal and control cookies launched by various malware and browser hijackers such as malicious browser extensions [20–22]. These attacks can easily bypass the *same origin policy* [23] of modern web browsers, which protects cookies stored by one origin from accessing by a different origin. As an example, recently cookie-stolen related XSS vulnerabilities were even found in Google's hosting services [24,25]; and the flaws in Internet Explorer 7 and Firefox could enable attackers to steal and manipulate cookies stored on a PC's hard drive [26].

Disabling third-party cookies (both session and persistent) and enabling first-party session cookies have been supported by most web browsers. The hardest problem in cookie management is how to handle first-party persistent cookies. Currently web browsers only have very limited functions such as automatically accepting or blocking all first-party persistent cookies, or prompting each of them to let users manually make decisions. Fig. 1 shows the advanced privacy settings of Internet Explorer 7, in which the functions provided to control first-party persistent cookies are very cumbersome and impractical to use. Therefore, the focus of this paper is on first-party persistent cookies and how to automatically manage the usage of first-party persistent cookies on behalf of a user[1]. Instead of directly addressing XSS and various web browser vulnerabilities, CookiePicker reduces cookie privacy and security risks by removing useless first-party cookies from a user's hard disk. Here we assume that the hosting web site is legitimate, since it is worthless to protect the cookies of a malicious site.

[1] The design of CookiePicker and its decision algorithms can be easily applied to other kinds of cookies as well if needed.

3. HTTP cookie measurement

To comprehensively understand the usage of HTTP cookies over the Internet, we conduct a large scale measurement study in December 2009. We choose over five thousand web sites from *directory.google.com*, an online web site directory containing millions of web sites categorized by topic. Then, we use a customized web page retrieval tool `wget` [27] to identify those web sites that set cookies at client side. We only consider first-party cookies, including both session cookies and persistent cookies, in this measurement study.

3.1. Web site selection and crawling

Since there are numerous web sites on the Internet, we select the web sites for measurement study based on the following two requirements: diversity and representativeness. As with [28], the selection pool we use is *directory.google.com*, in which web sites are organized into hierarchical categories and listed in Google page-rank order indicating the relevance to the classified category. There are fifteen top-level categories in *directory.google.com*. Our selection covers thirteen of them, except categories "world" and "regional". These two are avoided due to the existence of many non-English web sites. Since a top-level category has multiple sub-categories which also have further sub-categories, each top-level category may consist of hundreds of thousands of web sites. Thus, we only choose the web sites listed in the top-level categories and their immediate sub-categories. To select representative web sites, we filter out those web sites with page-rank less than 0.25[2]. There may exist duplicate web sites inside a category or between two categories. We first remove the duplicate intra-category web

[2] Page-rank value is between 0 and 1. The bigger, the more relevant.

Table 1
Statistics of the selected web sites and their usage of HTTP cookies.

Category	Web sites	Cookies	Session only	Persistent only	Both
Arts	342	113 (33%)	55	21	37
Business	602	265 (44%)	181	28	56
Computers	512	190 (37%)	108	33	49
Games	151	51 (34%)	19	12	20
Health	303	148 (49%)	82	17	49
Home	324	157 (48%)	83	23	51
News	483	176 (36%)	86	51	39
Recreation	351	173 (49%)	86	36	51
Reference	308	134 (44%)	69	17	48
Science	537	215 (40%)	118	48	49
Shopping	461	290 (63%)	93	67	130
Society	214	78 (36%)	35	17	26
Sports	673	287 (43%)	152	51	84
Multi-category	132	64 (48%)	23	24	17
Total	5,393	2,341 (43%)	1,190	445	706

sites and then move all the duplicate inter-category web sites into a new category—"Multi-category". After filtering and removing, the total number of unique web sites chosen for our study is 5,393. The number of web sites in each category is listed in Table 1, where the second column lists the total number of web sites in the corresponding category and the third column shows the number and percentage of the web sites that set cookies at client side. The fourth, fifth, and sixth columns of Table 1 show the number of the web sites that set only session cookies, only persistent cookies, and both session and persistent cookies, respectively.

We instruct the customized `wget` to crawl the selected web sites and save the corresponding cookies carried in the HTTP response headers. To simulate a user surfing the web, we turn on the recursive option in `wget` so that `wget` can recursively retrieve web pages. The maximum depth of recursion level is set to two. We instruct `wget` to only save first-party cookies. To avoid crawling being blocked or running too long, we use a random wait time varied from zero to two seconds between consecutive retrievals, and limit the crawling time on a single web site within six minutes.

3.2. Measurement results

After web crawling, we find that 2,341 (43%) web sites in total have set at least one first-party cookie at client side. The average number of retrieved unique URLs per web site for these 2,341 web sites is 538. The percentage of web sites setting cookies in each category varies from 33% to 63%, as shown in Table 1. These numbers are conservative since `wget` cannot obtain the cookies set by the web pages that require a user login or carried by the client-side JavaScript. Even so, the results clearly show the pervasive cookie usage among various types of web sites.

For those web sites that use cookies, we compute the CDFs (Cumulative Distribution Functions) of the number of cookies, the number of session cookies, and the number of persistent cookies per web site, respectively, and draw them in Fig. 2. Note that the *X*-axis is in logarithmic scale. We are interested in these numbers because web browsers usually set limits on the total number of cookies and the number of cookies per domain, in order to save memory

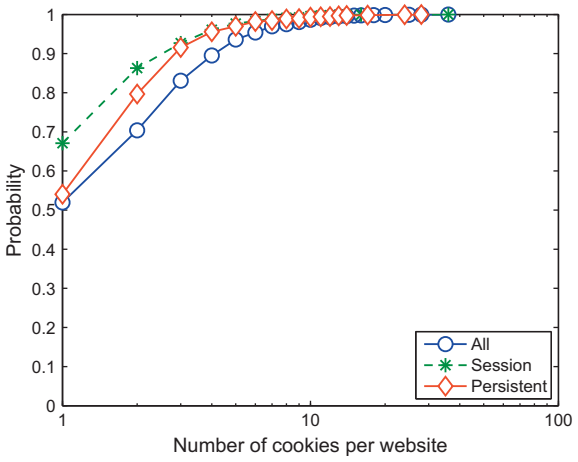

Fig. 2. CDFs of the number of all cookies, the number of session cookies, the number of persistent cookies per web site, respectively.

and disk space on a user's computer. For example, the maximum default number of cookies per domain and maximum number of total cookies are set to 50 and 1,000 respectively in Firefox. From Fig. 2, we can see that about 99% of the web sites set less than 10 cookies at client side and none of them sets more than 40 cookies. This indicates that all web sites under study follow the basic rule that a web site should not set too many cookies to a client.

Web browsers usually also set limit on the size of a cookie, which is mainly determined by the length of cookie value. For example, Firefox limits the length of a cookie `NAME/VALUE` pair up to 4,096 bytes. Fig. 3 depicts the CDFs of cookie size in bytes for all cookies, session cookies, and persistent cookies, respectively. Around 94% of cookies are less than 100 bytes and none of them is greater than 4,096 bytes.

If cookie is enabled during browsing a web site, the path of URL will determine what cookies to be transmitted together with the HTTP request to server. If the `Path` attribute of a cookie matches the prefix of the URL path, the cookie previously set will be sent back. We examine the

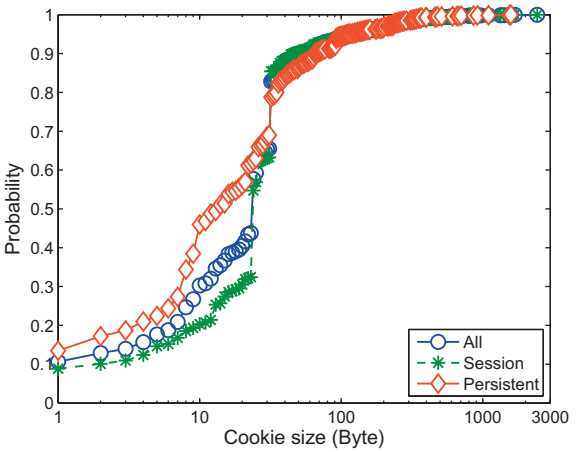

Fig. 3. CDFs of cookie size in bytes for all cookies, session cookies, and persistent cookies, respectively.

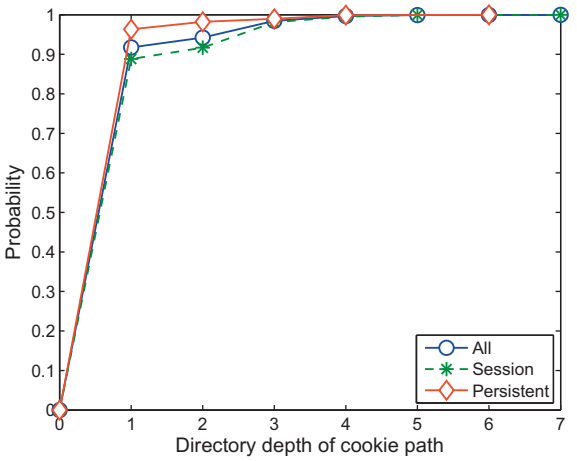

Fig. 4. CDFs of cookie PATH depth for all cookies, session cookies, and persistent cookies, respectively.

`Path` depth for cookies, session cookies, and persistent cookies, respectively, and draw their CDFs in Fig. 4. Surprisingly, near 90% of cookies have root "/" as their `Path` attribute values, which implies that these cookies will be sent back to the original web sites for any URL requests to those sites. Although some cookies may need to be sent back to the web servers for any URL on the servers, the abnormally high percentage suggests that the `Path` attributes of many cookies may not be appropriately set. As a consequence of this inappropriate setting, many requests carry cookies that are functionally unnecessary: the cookies neither affect responses nor make difference to server states. Wide-spread indiscriminate usage of cookies not only impedes many optimizations such as content delivery optimizations studied in [12], but also increases the risk of cookie stealing.

We are especially interested in the lifetime of persistent cookies, which can be calculated from the cookie attribute `Max-Age` specified in the `Set-Cookie` HTTP response header. We compute the lifetime for all persistent cookies and draw its CDF in Fig. 5 (The *X*-axis is log-scaled). There are a few hikes in the curve, among which the lifetimes corresponding to one year (365 days) and 30 years are most evident. Clearly, over 60% of persistent cookies are set to expire after one year or longer. We also find that seven web sites set their cookies to expire in year 9999, nearly eight thousand years from now! The mean and median lifetimes of persistent cookies are 38 years and one year, respectively.

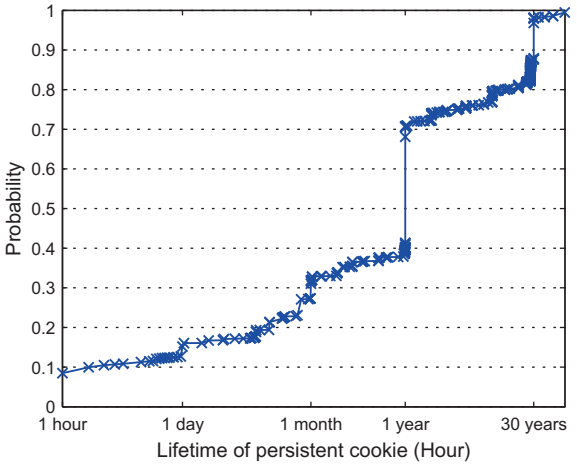

Fig. 5. CDF of persistent cookie lifetime.

In summary, our major measurement findings show that cookie has been widely used by web sites. Although the simple cookie collection tool we use cannot retrieve the web pages that require login and cannot acquire the cookies set by JavaScript, we still find that about 43% of the total selected web sites use either session cookies, persistent cookies, or both. Moreover, around 21% of the total selected web sites use persistent cookies and the majority of persistent cookies have their lifetimes longer than one year. Therefore, the pervasive usage of persistent cookies and their very long lifetimes clearly highlight the demand for removing those useless persistent cookies to reduce potential privacy risks.

We did a similar measurement study in December 2006 [29]. The procedure and tool for selecting and crawling web sites used in that study are the same as our current study. Although we cannot directly compare the results of two studies due to the differences of web sites and crawling paths, we find that the findings mentioned above hold in the 2006 study. For example, in the 2006 study, about 37% of the total (5,211) selected web sites use cookies and about 18% of the total selected web sites use persistent cookies. Similarly, more than 60% of persistent cookies have lifetimes longer than one year in the 2006 study.

Recently Tappenden and Miller [15] did an extensive study on cookie deployment and usage. Some of their study results have the similar implications as ours. For example, their study shows that 67.4% (66,031) of the 98,006 web sites use cookies and 83.8% (55,130) of those 66,031 sites use first-party cookies. In addition, they ob-

served that approximately 50.4% of all persistent cookies surveyed are set to live for over one year. The discrepancy in numbers between their study and ours can be attributed to the differences in the sample set and study methods. For example, they chose web sites from Alexa while we chose web sites from Google directory; they instrumented Firefox browser while we instrumented `wget`.

4. CookiePicker design

The design goal of CookiePicker is to effectively identify the useful cookies of a web site, and then disable the return of those useless cookies back to the web site in the subsequent requests and finally remove them. A web page is automatically retrieved twice by enabling and disabling some cookies. If there are obvious differences between the two retrieved results, we classify the cookies as useful; otherwise, we classify them as useless. CookiePicker enhances the cookie management for a web site by two processes: forward cookie usefulness marking and backward error recovery. We define these two processes and detail the design of CookiePicker in the following.

Definition 1. FORward Cookie Usefulness Marking (FORCUM) is a training process, in which CookiePicker determines cookie usefulness and marks certain cookies as useful for a web site.

Definition 2. Backward error recovery is a tuning process, in which wrong decisions made by CookiePicker in the FORCUM process may be adjusted automatically or manually for a web site.

4.1. Regular and hidden requests

A typical web page consists of a container page that is an HTML text file, and a set of associated objects such as stylesheets, embedded images, scripts, and so on. When a user browses a web page, the HTTP request for the container page is first sent to the web server. Then, after receiving the corresponding HTTP response for the container page, the web browser analyzes the container page and issues a series of HTTP requests to the web server for downloading the objects associated with the container page. The HTTP requests and responses associated with a single web page view are depicted by the solid lines (1) and (2) in Fig. 6, respectively.

Web page contents coming with the HTTP responses are passed into the web browser layout engine for processing.

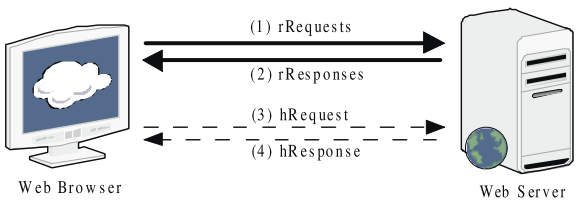

Fig. 6. HTTP requests/responses in a single web page view.

Fig. 7 depicts the data flow inside of Gecko [30], one of the most popular web browser layout engines used in all Mozilla-branded software and its derivatives. When a web page comes into Gecko, its container page is first parsed and built into a tree. The structure of the tree follows the W3C Document Object Model (DOM), thus the tree is known as DOM tree. Next, the data from the DOM tree are put into abstract boxes called frames by combining the information from stylesheets. Finally, these frames and the associated web objects are rendered and displayed on a user's screen.

In order to identify the cookie usefulness for a web page, CookiePicker compares two versions of the same web page: the first version is retrieved with cookies enabled and the second version is retrieved with cookies disabled. The first version is readily available to CookiePicker in the user's regular web browsing window. CookiePicker only needs to retrieve the second version of the container page. Similar to Doppelganger [11], CookiePicker utilizes the ever increasing client side spare bandwidth and computing power to run the second version. However, unlike Doppelganger, CookiePicker neither maintains a fork window nor mirrors the whole user session. CookiePicker only retrieves the second version of the container page by sending a single hidden HTTP request. As shown in Fig. 6, line (3) is the extra hidden HTTP request sent by CookiePicker for the second version of the container page, and line (4) represents the corresponding HTTP response. In the rest of the paper, we simply refer the requests and responses, represented by the solid lines (1) and (2) of Fig. 6, as regular requests and responses; and refer the extra request and response, represented by the dashed lines (3) and (4) of Fig. 6, as the hidden request and response.

4.2. Forward cookie usefulness marking

As shown in Fig. 8, the FORCUM process consists of five steps: regular request recording, hidden request sending, DOM tree extraction, cookie usefulness identification, and cookie record marking.

When visiting a web page, a user issues regular requests and then receives regular responses. At the first step, CookiePicker identifies the regular request for the container page and saves a copy of its URI and header information. CookiePicker needs to filter out the temporary redirection or replacement pages and locate the real initial container document page.

At the second step, CookiePicker takes advantage of user's think time [31] to retrieve the second copy of the container page, without causing any delay to the user's regular browsing. Specifically, right after all the regular responses are received and the web page is rendered on the screen for display, CookiePicker issues the single hidden request for the second copy of the container page. In the hidden request, CookiePicker uses the same URI as the saved in the first step. It only modifies the "Cookie" field of the request header by removing a group of cookies, whose usefulness will be tested. The hidden request can be transmitted in an asynchronous mode so that it will not block any regular browsing functions. Then, upon the arrival of the hidden response, an event handler will be

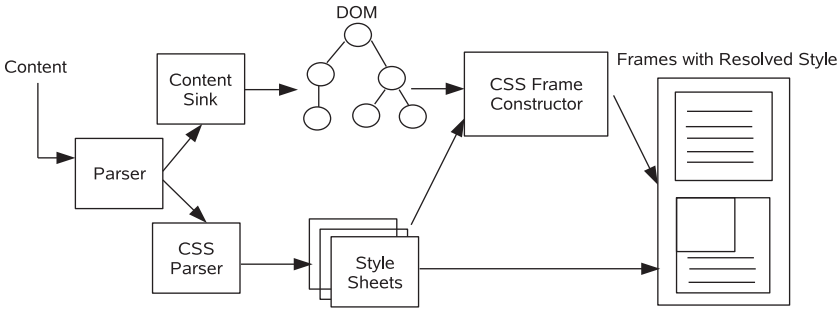

Fig. 7. Data flow inside Gecko.

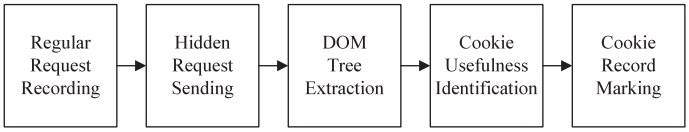

Fig. 8. The Five-Step FORCUM Process.

triggered to process it. Note that the hidden request is only used to retrieve the container page, and the received hidden response will not trigger any further requests for downloading the associated objects. Retrieving the container page only induces very low overhead to CookiePicker.

At the third step, CookiePicker extracts the two DOM trees from the two versions of the container page: one for the regular response and the other for the hidden response. We call these two DOM trees the regular DOM tree and the hidden DOM tree, respectively. The regular DOM tree has already been parsed by web browser layout engine and is ready for use by CookiePicker. The hidden DOM tree, however, needs to be built by CookiePicker; and CookiePicker should build the hidden DOM tree using the same HTML parser of the web browser. This is because in practice HTML pages are often malformed. Using the same HTML parser guarantees that the malformed HTML pages are treated as same as before, while the DOM tree is being constructed.

At the fourth step, CookiePicker identifies cookie usefulness by comparing the differences between the two versions of the container page, whose information is well represented in the two DOM trees. To make a right cookie usefulness decision, CookiePicker uses two complementary algorithms by considering both the internal structural difference and the external visual content difference between the two versions. Only when obvious structural difference and visual content difference are detected, will CookiePicker decide that the corresponding cookies that are disabled in the hidden request are useful. The two algorithms will be detailed in Section 5.

At the fifth step, CookiePicker will mark the cookies that are classified as useful in the web browser's cookie jar. An extra field "useful" is introduced to each cookie record. At the beginning of the FORCUM process, a *false* value is assigned to the "useful" field of each cookie. In addition, any newly-emerged cookies set by a web site are also assigned *false* values to their "useful" fields. During the FORCUM process, the value of the field "useful" can only be changed in one direction, that is, from "false" to "true" if some cookies are classified as useful. Later on, when the values of the "useful" field for the existing cookies are relatively stable for the web site, those cookies that still have "false" values in their "useful" fields will be treated as useless and will no longer be transmitted to the corresponding web site. Then, the FORCUM process can be turned off for a while; and it will be turned on automatically if CookiePicker finds new cookies appeared in the HTTP responses or manually by a user if the user wants to continue the training process.

4.3. Backward error recovery

In general, CookiePicker could make two kinds of errors in the FORCUM process. The first kind of error is that useless cookies are misclassified as useful, thereby being continuously sent out to a web site. The second kind of error is that useful cookies are never identified by CookiePicker during the training process, thereby being blocked from a web site.

The first kind of error is solely due to the inaccuracy of CookiePicker in usefulness identification. Such an error will not cause any immediate trouble to a user, but leave useless cookies increasing privacy risks. CookiePicker is required to make such errors as few as possible so that a user's privacy risk is lowered. CookiePicker meets this requirement via accurate decision algorithms.

The second kind of error is caused by either a wrong usefulness decision or the fact that some cookies are only useful to certain web pages but have not yet been visited during the FORCUM process. This kind of error will cause inconvenience to a user and must be fixed by marking the corresponding cookies as useful. CookiePicker attempts to achieve a very low rate on this kind of error, so that it does not cause any inconvenience to users. This requirement is achieved by two means. On one hand, for those visited pages, the decision algorithms of CookiePicker attempt

to make sure that each useful persistent cookie can be identified and marked as useful. On the other hand, since CookiePicker is designed with very low running cost, a longer running period (or periodically running) of the FORCUM process is affordable, thus training accuracy can be further improved.

CookiePicker provides a simple recovery button for backward error recovery in the tuning process. In case a user notices some malfunctions or some strange behaviors on a web page, the cookies disabled by CookiePicker in this particular web page view can be re-marked as useful via a simple button click. Note that once the cookie set of a web site becomes stable after the training and tuning processes, those disabled useless cookies will be removed from the web browser's cookie jar. CookiePicker also provides a user interface, allowing a user to view those useless cookies and confirm the deletion action. We will introduce this interface in Section 6.

5. HTML page difference detection

In this section, we present two complementary mechanisms for online detecting the HTML web page differences between the enabled and disabled cookie usages. In the first mechanism, we propose a restricted version of Simple Tree Matching algorithm [32] to detect the HTML document structure difference. In the second mechanism, we propose a context-aware visual content extraction algorithm to detect the HTML page visual content difference. We call these two mechanisms as Restricted Simple Tree Matching (RSTM) and Context-aware Visual Content Extraction (CVCE), respectively. Intuitively, RSTM focuses on detecting the internal HTML document structure difference, while CVCE focuses on detecting the external visual content difference perceived by a user. In the following, we present these two mechanisms and explain how they are complementarily used.

5.1. Restricted simple tree matching

As mentioned in Section 4, in a user's web browser, the content of an HTML web page is naturally parsed into a DOM tree before it is rendered on the screen for display. Therefore, we resort to the classical measure of *tree edit distance* introduced by Tai [33] to quantify the difference between two HTML web pages. Since the DOM tree parsed from the HTML web page is rooted (document node is the only root), labeled (each node has node name), and ordered (the left-to-right order of sibling nodes is significant), we only consider *rooted labeled ordered tree*. In the following, we will first review the tree edit distance problem; then we will explain why we choose *top-down distance* and detail the RSTM algorithm; and finally we will use Jaccard similarity coefficient to define the similarity metric of a normalized DOM tree.

5.1.1. Tree edit distance

For two rooted labeled ordered trees T and T', let $|T|$ and $|T'|$ denote the numbers of nodes in trees T and T', and let $T[i]$ and $T'[j]$ denote the ith and jth preorder traversal nodes in trees T and T', respectively. Tree edit distance is defined as the minimum cost sequence of edit operations to transform T into T' [33]. The three edit operations used in transformation include: inserting a node into a tree, deleting a node from a tree, and changing one node of a tree into another node. Disregarding the order of the edit operations being applied, the transformation from T to T' can be described by a mapping. The formal definition of a mapping [33] is as follows:

Definition 3. A mapping from T to T' is defined by a triple (M, T, T'), where M is any set of pairs of integers (i, j) satisfying:

1. $1 \leqslant i \leqslant |T|,\ 1 \leqslant j \leqslant |T'|$
2. For any two pairs (i_1, j_1) and (i_2, j_2) in M,
   (a) $i_1 = i_2$ iff $j_1 = j_2$;
   (b) $i_1 < i_2$ iff $j_1 < j_2$;
   (c) $T[i_1]$ is an ancestor (descendant) of $T[i_2]$ iff $T'[j_1]$ is an ancestor (descendant) of $T'[j_2]$.

Intuitively, condition (2a) ensures that each node of both trees appears at most once in the mapping, and condition (2b) and (2c) ensure that the structural order is preserved in the mapping.

The algorithm presented by Tai [33] solves the tree edit distance problem in time $O(|T| \cdot |T'| \cdot |D|^2 \cdot |D'|^2)$, where $|D|$ and $|D'|$ are the maximum depths, respectively, of T and T'. Zhang et al. [34] further improved the result via a simple fast dynamic programming algorithm in time $O(|T| \cdot |T'| \cdot min\{|D|, |L|\} \cdot min\{|D'|, |L'|\})$, where $|L|$ and $|L'|$ are the numbers of leaves in T and T', respectively.

Since the solution of the generic tree edit distance problem has high time complexity, researchers have investigated the constrained versions of the problem. By imposing conditions on the three edit operations mentioned above, a few different tree distance measures have been proposed and studied in the literature: *alignment distance* [35], *isolated subtree distance* [36], *top-down distance* [37,32], and *bottom-up distance* [38]. The description and comparison of these algorithms are beyond the scope of this paper, see [39] and [38] for details.

5.1.2. Top-down distance

Because RSTM belongs to the top-down distance approach, we review the definition of top-down distance and explain why we choose this measure for our study.

Definition 4. A mapping (M, T, T') from T to T', is top-down if it satisfies the condition that for all i, j such that $T[i]$ and $T'[j]$ are not the roots, respectively, of T and T':

if pair $(i, j) \in M$ *then* $(parent(i), parent(j)) \in M$.

The top-down distance problem was introduced by Selkow [37]. In [32], Yang presented a $O(|T| \cdot |T'|)$ time-complexity top-down dynamic programming algorithm, which is named as the Simple Tree Matching (STM) algorithm. As we mentioned earlier, our goal is to effectively detect noticeable HTML web page difference between the enabled and disabled cookie usages. The measure of top-down distance captures the key structure difference be-

tween DOM trees in an accurate and efficient manner, and fits well to our requirement. In fact, top-down distance has been successfully used in a few web-related projects. For example, Zhai and Liu [40] used it for extracting structured data from web pages; and Reis et al. [41] applied it for automatic web news extraction. In contrast, bottom-up distance [38], which although can be more efficient in time complexity ($O(|T| + |T'|)$), falls short of being an accurate metric [42] and may produce a far from optimal result [43] for HTML DOM tree comparison, in which most of the differences come from the leaf nodes.

5.1.3. Restricted simple tree matching

Based on the original STM algorithm [32], Fig. 9 illustrates RSTM, our restricted version of STM algorithm. Other than lines 4 to 8 and one new parameter *level*, our RSTM algorithm is similar to the original STM algorithm. Like the original STM algorithm, we first compare the roots of two trees A and B. If their roots contain different symbols, then A and B do not match at all. If their roots contain same symbols, we use dynamic programming to recursively compute the number of pairs in a maximum matching between trees A and B. Fig. 10 gives two trees, in which each node is represented as a circle with a single letter inside. According to the preorder traversal, the fourteen nodes in tree A are named from $N1$ to $N14$, and the eight nodes in tree B are named from $N15$ to $N22$. The final result returned by STM algorithm or RSTM algorithm is the number of matching pairs for a maximum matching. For example, STM algorithm will return "7" for the two trees in Fig. 10, and the seven matching pairs are $\{N1, N15\}, \{N2, N16\}, \{N6, N18\}, \{N7, N19\}, \{N5, N17\}, \{N11, N20\}$, and $\{N12, N22\}$.

There are two reasons why a new parameter *level* is introduced in RSTM. First, some web pages are very dynamic. From the same web site, even if a web page is retrieved twice in a short time, there may exist some differences between the retrieved contents. For example, if we refresh `Yahoo` home page twice in a short time, we can often see some different advertisements. For CookiePicker, such dynamics on a web page are just noises and should be differentiated from the web page changes caused by the enabled and disabled cookie usages. The advertisement replacements on a web page use different data items (e.g., images or texts) but they often stay at the same location of a web page's DOM tree. Data items are mainly represented by lower level nodes of a DOM tree [44]. In contrast, the web page changes caused by enabling/disabling cookies may introduce structural dissimilarities at the upper level of a DOM tree, especially when the theme of the page is changed. By using the new parameter *level*, the RSTM algorithm restricts the top-down comparison between the two trees to a certain maximum level. Therefore, equipped with the parameter *level*, RSTM not only captures the key structure dissimilarity between DOM trees, but also reduces leaf-level noises.

```
Algorithm: RSTM(A, B, level)
1.  if the roots of tree A and tree B contain different symbols then
2.      return(0);
3.  endif
4.  currentLevel = level + 1;
5.  if A and B are leaf or non-visible nodes or
6.      currentLevel > maxLevel then
7.      return(0);
8.  endif
9.  m = the number of first-level subtrees of A;
10. n = the number of first-level subtrees of B;
11. Initialization, M[i, 0] = 0 for i = 0, ..., m;
12.                 M[0, j] = 0 for j = 0, ..., n;
13. for i = 1 to m do
14.     for j = 1 to n do
15.         M[i, j] = max(M[i, j − 1], M[i − 1, j],
                          M[i − 1, j − 1] + W[i, j]);
16.         where W[i, j] = RSTM(A_i, B_j, currentLevel)
17.             where A_i and B_j are the ith and jth
                first-level subtrees of A and B, respectively
18.     endfor
19. endfor
20. return (M[m, n] + 1);
```

Fig. 9. The Restricted Simple Tree Matching Algorithm.

The second reason of introducing the new parameter *level* is that the $O(|T| \cdot |T'|)$ time complexity of STM is still too expensive to use online. Even with C++ implementation, STM will spend more than one second in difference detection for some large web pages. However, as shown in Section 6, the cost of the RSTM algorithm is low enough for online detection.

The newly-added conditions at line 5 of the RSTM algorithm restrict that the mapping counts only if the compared nodes are not leaves and have visual effects. More specifically, all the comment nodes are excluded in that they have no visual effect on the displayed web page. Script nodes are also ignored because normally they do not contain any visual elements either. Text content nodes, although very important, are also excluded due to the fact that they are leaf nodes (i.e., having no more structural information). Instead, text content will be analyzed in our Context-aware Visual Content Extraction (CVCE) mechanism.

5.1.4. Normalized top-down distance metric

Since the return result of RSTM (or STM) is the number of matching pairs for a maximum matching, based on the Jaccard similarity coefficient that is given in Formula (1), we define the normalized DOM tree similarity metric in Formula (2).

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}, \tag{1}$$

$$NTreeSim(A,B,l) = \frac{RSTM(A,B,l)}{N(A,l) + N(B,l) - RSTM(A,B,l)}. \tag{2}$$

The Jaccard similarity coefficient $J(A,B)$ is defined as the ratio between the size of the *intersection* and the size of the *union* of two sets. In the definition of our normalized DOM tree similarity metric $NTreeSim(A,B,l)$, $RSTM(A,B,l)$ is the returned number of matched pairs by calling RSTM on trees A and B for upper l levels. $N(A,l)$ and $N(B,l)$ are the numbers of non-leaf visible nodes at upper l levels of trees A and B, respectively. Actually $N(A,l) = RSTM(A,A,l)$ and $N(B,l) = RSTM(B,B,l)$, but $N(A,l)$ and $N(B,l)$ can be

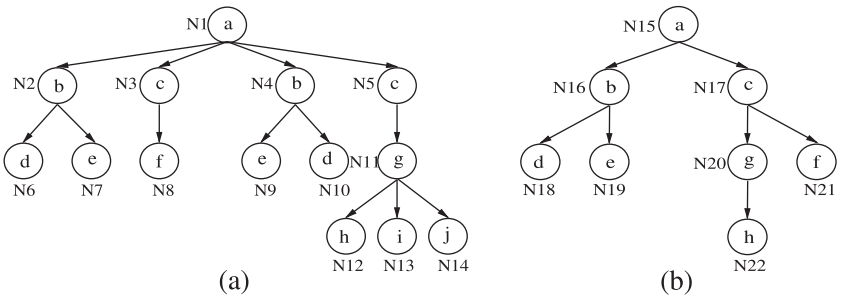

Fig. 10. (a) Tree A. (b) Tree B.

computed in $O(n)$ time by simply preorder traversing the upper l levels of trees A and B, respectively.

5.2. Context-aware visual content extraction

The visual contents on a web page can be generally classified into two groups: text contents and image contents. Text contents are often displayed as headings, paragraphs, lists, table items, links, and so on. Image contents are often embedded in a web page in the form of icons, buttons, backgrounds, flashes, video clips, and so on. Our second mechanism mainly uses text contents, instead of image contents, to detect the visual content difference perceived by users. Two reasons motivate us to use text contents rather than image contents. First, text contents provide the most important information on web pages. This is because HTML mainly describes the structure of text-based information in a document, while image contents often serve as supplements to text contents [45]. In practice, users can block the loading of various images and browse web pages in text mode only. Second, the similarity between images cannot be trivially compared, while text contents can be extracted and compared easily as shown below.

On a web page, each text content exists in a special context. Corresponding to the DOM tree, the text content is a leaf node and its context is the path from the root to this leaf node. For two web pages, by extracting and comparing their context-aware text contents that are essential to users, we can effectively detect the noticeable HTML web page difference between the enabled and disabled cookie usages. Fig. 11 depicts the recursive algorithm to extract the text content.

The contentExtract algorithm traverses the whole DOM tree in preorder in time $O(n)$. During the preorder traversal, each non-noise text node is associated with its context, resulting in a context-content string; and then the context-content string is added into set S. The final return result is set S, which includes all the context-content strings. Note that in lines 2 to 4, only those non-noise text nodes are qualified for the addition to set S. Similar to [46], scripts, styles, obvious advertisement text, date and time strings, and option text in dropdown lists (such as country list or language list) are regarded as noises. Text nodes that contain no alphanumeric characters are also treated as noises. All these checkings guarantee that we can extract a relatively concise context-content string set from the DOM tree.

Algorithm: contentExtract(T, *context*)
1. Initialization, $S = \emptyset$; *node* = T.*root*;
2. **if** *node* is a non-noise text node **then**
3. $cText$ = *context*+SEPARATOR+*node.value*;
4. $S = S \cup \{cText\}$;
5. **elseif** node is an element node **then**
6. *currentContext* = *context*+":"+*node.name*;
7. n = the number of first-level subtrees of T;
8. **for** $j = 1$ **to** n **do**
9. $S = S \cup$contentExtract(T_i, *currentContext*);
where T_i is the ith first-level subtrees of T;
10. **endfor**
11. **endif**
12. **return** (S);

Fig. 11. The Text Content Extraction Algorithm.

Assume S_1 and S_2 are two context-content string sets extracted from two DOM trees A and B, respectively. To compare the difference between S_1 and S_2, again based on the Jaccard similarity coefficient, we define the normalized context-content string set similarity metric in Formula (3):

$$NTextSim(S_1, S_2) = \frac{|S_1 \cap S_2| + s}{|S_1 \cup S_2|}. \quad (3)$$

Formula (3) is a variation [47] of the original Jaccard similarity coefficient. The extra added s on the numerator stands for the number of those context-content strings that are not exactly same, while having the same context prefix, in S_1 and S_2. Intuitively, between two sets S_1 and S_2, Formula (3) disregards the difference caused by text content replacement occurred in the same context; it only considers the difference caused by text content appeared in each set's unique context. This minor modification is especially helpful in reducing the noises caused by advertisement text content and other dynamically changing text contents.

5.3. Making decision

As discussed above, to accurately identify useful cookies, CookiePicker has to differentiate the HTML web page differences caused by web page dynamics from those

Algorithm: decision(A, B, l)
1. **if** NTreeSim$(A, B, l) \leq$ Thresh1 **and**
2. NTextSim$(S_1, S_2) \leq$ Thresh2 **then**
3. **return** the difference is caused by cookies;
4. **else**
5. **return** the difference is caused by noises;
6. **endif**

Fig. 12. CookiePicker Decision Algorithm.

caused by disabling cookies. Assume that tree *A* is parsed from a web page retrieved with cookies enabled and tree *B* is parsed from the same web page with cookies disabled. CookiePicker examines these two trees by using both algorithms presented above. If the return results of *NTreeSim* and *NTextSim* are less than two tunable thresholds, *Thresh1* and *Thresh2*, respectively, CookiePicker will make a decision that the difference is due to cookie usage. Fig. 12 depicts the final decision algorithm.

Note that these two thresholds are internal to CookiePicker, so a regular user does not need to know them. In our experiments (Section 6.2), we set the values of both thresholds to 0.85, and we found that no backward error recover is needed. We would like to recommend this as a reference value for CookiePicker. However, it is possible to further tune these two thresholds. For example, one approach is to self-adaptively adjust the thresholds based on the number or frequency of a user's backward error recover actions. The bottom line is that backward error recovery should not cause too much inconvenience to a user. Another approach is to use site-specific thresholds so that the granularity of accuracy can be refined to the site-level. In addition, it is also possible to allow users to share fine-tuned site-specific thresholds.

6. System evaluation

In this section, we first briefly describe the implementation of CookiePicker, and then we validate its efficacy through two sets of live experiments.

6.1. Implementation

We implemented CookiePicker as a Firefox extension. Being one of the most popular web browsers, Firefox is very extensible and allows programmers to add new features or modify existing features. Our CookiePicker extension is implemented in about 200 lines of XML user interface definition code, 1,600 lines of JavaScript code, and 600 lines of C++ code. JavaScript code is used for HTTP request/response monitoring and processing, as well as cookies record management. The HTML page difference detection algorithms are implemented in C++, because JavaScript version runs very slow. C++ code is compiled into a shared library in the form of an XPCOM (Cross-Platform Component Object Mode) component, which is accessible to JavaScript code. CookiePicker is a pure Firefox extension and it does not make any change to the Firefox's source code.

We omit other details and only describe two key interfaces in CookiePicker's implementation: the interface to user and the XPCOM component interface. Fig. 13 shows CookiePicker's main user interface. We port the codes of a popular Firefox extension Cookie Culler [48] and merge them into CookiePicker. Cookie Culler allows a user to access cookie records and manually delete those cookies the user does not need to keep. By integrating this interface, CookiePicker provides a user with the capability to easily view the decisions made by CookiePicker and double check those useless cookies before they are finally removed from a browser's cookie jar. As shown in Fig. 13,

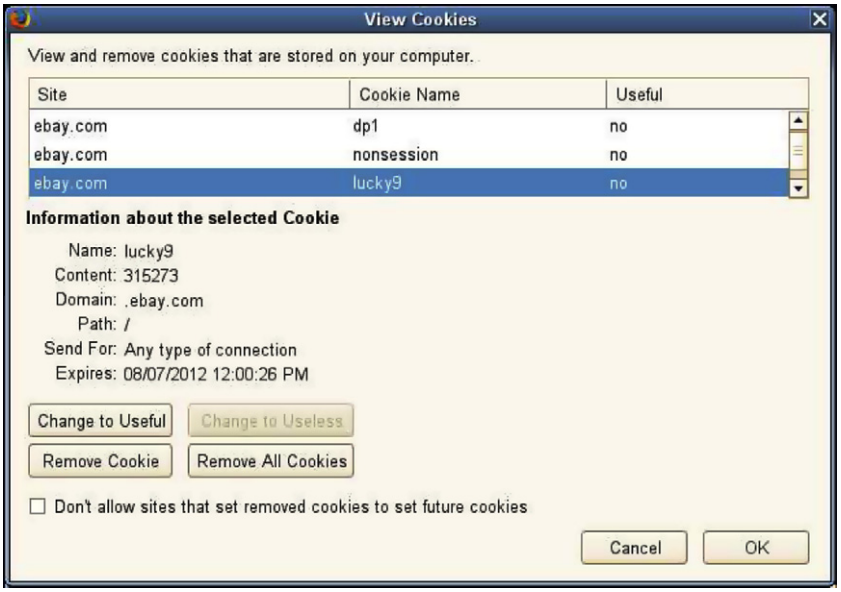

Fig. 13. CookiePicker user interface.

if a user wants, the user can know that these three first-party persistent cookies from ebay.com have been automatically marked as useless and will be deleted by CookiePicker. The user can also make corrections to this result if necessary.

In the XPCOM component interface, two functions are defined as follows and they correspond to CookiePicker's two HTML page difference detection algorithms, respectively:

```
interface ICookiePickerComponent : nsISupports{
    float nTreeSim(in nsIDOMNode rNode, in nsIDOMNode
    hNode, in long l);
    float nTextSim(in nsIDOMNode rNode, in nsIDOMNode
    hNode);
}
```

These two functions are implemented in C++, and can be accessed by JavaScript code via the ICookiePickerComponent component. For `nTreeSim`, its three input parameters match exactly with those in Fig. 12. For `nTextSim`, its definition here is a little bit different from that in Fig. 12, because the DOM trees are directly passed in and the corresponding context-content string sets are extracted internally.

6.2. Evaluation

We installed CookiePicker on a Firefox version 1.5.0.8 web browser[3] and designed two sets of experiments to validate the effectiveness of CookiePicker in identifying the useful first-party persistent cookies. The first set of experiments is to measure the overall effectiveness of CookiePicker and its running time in a generic environment; while the second set of experiments focuses on the web sites whose persistent cookies are useful only, and examines the identification accuracy of CookiePicker upon useful persistent cookies. For all the experiments, the regular browsing window enables the use of persistent cookies, while the hidden request disables the use of persistent cookies by filtering them out from HTTP request header. The two thresholds used in CookiePicker decision algorithm are both set to 0.85, i.e., $Thresh1 = Thresh2 = 0.85$. The parameter l for NTreeSim algorithm is set to 5, i.e., the top five level of DOM tree starting from the *body* HTML node will be compared by NTreeSim algorithm.

6.2.1. First set of experiments

From each of the 15 categories listed in Table 1, we randomly choose two web sites that use persistent cookies. Thus, in total there are 30 web sites in the first set of experiments. As listed in the first column of Table 2, these 30 web sites are represented as S1 to S30 for privacy concerns.

Inside each web site, we first visit over 25 web pages to stabilize its persistent cookies and the "useful" values of the persistence cookies, i.e, no more persistent cookies of the web site are marked as "useful" by CookiePicker afterwards. Then, we count the number of persistent cookies set by the web site and the number of persistent cookies marked as useful by CookiePicker. These two numbers are shown in the second and third columns of Table 2, respectively. Among the total 30 web sites, the persistent cookies from five web sites (S1, S6, S10, S16, S27) are marked as "useful" by CookiePicker, and the persistent cookies from the rest of 30 web sites are identified as "useless". In other words, CookiePicker indicates that we can disable the persistent cookies in about 83.3% (25 out of 30) of testing web sites. To further validate the testing result above, we check the uselessness of the persistent cookies for those 25 web sites through careful manual verification. We find that blocking the persistent cookies of those 25 web sites does not cause any problem to a user. Therefore, none of the classified "useless" persistent cookies is useful, and no backward error recovery is needed.

For those five web sites that have some persistent cookies marked as "useful", we verify the real usefulness of these cookies by blocking the use of them and then comparing the disabled version with a regular browsing window over 25 web pages in each web site. The result is shown in the fourth column of Table 2. We observe that three cookies from two web sites (S6, S16) are indeed useful. However, for the other three web sites (S1, S10, S27), their persistent cookies are useless but are wrongly marked as "useful" by CookiePicker. This is mainly due to the conservative threshold setting. Currently the values of both thresholds are set to 0.85, i.e., $Thresh1 = Thresh2 = 0.85$. The rationale behind the conservative threshold setting is that we prefer to have all useful persistent cookies be correctly identified, even at the cost of some useless cookies being misclassified as "useful". Thus, the number of backward error recovery is minimized.

In Table 2, the fifth and sixth columns show the average running time of the detection algorithms and the entire duration of CookiePicker, respectively. It is clear that the running time of the page difference detection is very short with an average of 14.6 ms over the 30 web sites. This detection time is roughly the extra CPU computation time introduced by running the CookiePicker browser extension. The extra memory resource consumption is also negligible because CookiePicker does not have a large memory requirement. The average identification duration is 2,683.3 ms, which is reasonable short considering the fact that the average think time of a user is about 10 s [31]. Note that web sites S4, S17, and S28 have abnormally high identification duration at about 10 s, which is mainly caused by the slow responses from these web sites.

6.2.2. Second set of experiments

Since only two web sites in the first set of experiments have useful persistent cookies, we attempt to further examine if CookiePicker can correctly identify each useful persistent cookie in the second set of experiments. Because the list of web sites whose persistent cookies are really useful to users does not exist, we have to locate such web sites manually. Again, we randomly choose 200 web sites that use persistent cookies from the 15 categories listed in Table 1. Note that the 30 web sites chosen in the first set of experiments are not included in these 200 web sites. We manually scrutinize these 200 web sites,

[3] Porting CookiePicker to recent Firefox versions is quite feasible because CookiePicker uses the standard XPCOM mechanism of Firefox.

Table 2
Online testing results for thirty web sites (S1 to S30).

Web site	Persistent	Marked useful	Real useful	Detection time (ms)	CookiePicker duration (ms)
S1	2	2	0	8.3	1,821.6
S2	4	0	0	9.3	5,020.2
S3	5	0	0	14.8	1,427.5
S4	4	0	0	36.1	9,066.2
S5	4	0	0	5.4	698.9
S6	2	2	2	5.7	1,437.5
S7	1	0	0	17.0	3,373.2
S8	3	0	0	7.4	2,624.4
S9	1	0	0	13.2	1,415.4
S10	1	1	0	5.7	1,141.2
S11	2	0	0	2.7	941.3
S12	4	0	0	21.7	2,309.9
S13	1	0	0	8.0	614.9
S14	9	0	0	11.9	1,122.4
S15	2	0	0	8.5	948.0
S16	25	1	1	5.8	455.9
S17	4	0	0	7.5	11,426.3
S18	1	0	0	23.1	4,056.9
S19	3	0	0	18.0	3,860.5
S20	6	0	0	8.9	3,841.6
S21	3	0	0	14.4	936.1
S22	1	0	0	13.1	993.3
S23	4	0	0	28.8	2,430.1
S24	1	0	0	23.6	2,381.1
S25	3	0	0	30.7	550.1
S26	1	0	0	5.03	611.6
S27	1	1	0	8.7	597.5
S28	1	0	0	10.7	10,104.1
S29	2	0	0	7.7	1,387.1
S30	2	0	0	57.6	2,905.6
Total	103	7	3	–	–
Average	–	–	–	14.6	2,683.3

and finally find six web sites whose persistent cookies are really useful to users, i.e., without cookies, users would encounter some problems. Because the manual scrutiny is tedious, we cannot afford more effort to locate more such web sites. The six web sites are listed in the first column of Table 3 and represented as P1 to P6 for privacy concerns.

In Table 3, the second column shows the number of the cookies marked as "useful" by CookiePicker and the third column shows the number of the real useful cookies via manual verification. We observe that for the six web sites, all of their useful persistent cookies are marked as "useful" by CookiePicker. This result indicates that CookiePicker seldom misses the identification of a real useful cookie. On the other hand, for web sites P5 and P6, some useless persistent cookies are also marked as "useful" because they are sent out in the same regular request with the real useful cookies. The fourth and fifth columns show the similarity score computed by $\mathrm{NTreeSim}(A, B, 5)$ and $\mathrm{NTextSim}(S_1, S_2)$, respectively, on the web pages that persistent cookies are useful. These similarity scores are far below 0.85, which is the current value used for the two thresholds *Thresh1* and *Thresh2* in Fig. 12. The usage of these useful persistent cookies on each web site is given at the sixth column. Web sites P1, P4, and P6 use persistent cookies for user's preference setting. Web sites P3 and P5 use persistent cookies to properly create and sign up a new user. Web site P2 uses persistent cookie in a very unique way. Each user's persistent cookie corresponds to a specific sub-directory on the web server, and the sub-directory stores the user's recent query results. Thus, if the user visits the web site again with the persistent cookie, recent query results can be reused to improve query performance.

In summary, the above two sets of experiments show that by conservatively setting *Thresh1* and *Thresh2* to 0.85, CookiePicker can safely disable and remove persistent

Table 3
Online testing results for six web sites (P1 to P6) that have useful persistent cookies.

Web site	Marked useful	Real useful	NTreeSim $(A, B, 5)$	NTextSim (S_1, S_2)	Usage
P1	1	1	0.311	0.609	Preference
P2	1	1	0.459	0.765	Performance
P3	1	1	0.667	0.623	Sign Up
P4	1	1	0.250	0.158	Preference
P5	9	1	0.226	0.253	Sign Up
P6	5	2	0.593	0.719	Preference
Average	–	–	0.418	0.521	–

cookies from about 83.3% of web sites (25 out of the 30 web sites that we intensively tested). Meanwhile, all the useful persistent cookies are correctly identified by CookiePicker and no backward error recovery is needed for all the 8 web sites (S6, S16, P1, P2, P3, P4, P5, P6) that have useful persistent cookies. Misclassification happens only in 10% web sites (3 out of 30), on which useless persistent cookies are wrongly identified as useful.

7. Discussions

In CookiePicker, useful cookies are defined as those that can cause perceivable changes on a web page, and as we discussed before this definition is probably the most reasonable measure right now at the browser side. We assume that this measure is known to anyone who wants to evade CookiePicker. In this section, we first identify possible evasion techniques, and then we analyze potential evasion sources and explain why those evasion techniques are not a serious concern to CookiePicker. Finally, we discuss some concerns about using CookiePicker.

7.1. Possible evasion techniques

Since CookiePicker makes its decision based on HTML page difference detection, we identify the following four possible techniques that could be used to evade CookiePicker:

- *random advertising*: A web site can serve more random advertisements to different retrieves of a web page in order to reduce the accuracy of CookiePicker's difference detection algorithms.
- *dynamic content rewriting*: A web site can identify CookiePicker's hidden request and use JavaScript to dynamically rewrite web page contents at the browser side.
- *dramatic structure changing*: A web site can identify CookiePicker's hidden request and intentionally generate structurally different response pages (even if with very similar visual content).
- *cookie root path abusing*: A web site can set attribute values of all its cookies as root "/" to let all its cookies, no matter useful or useless, return back for every web request.

For the *random advertising* evasion technique, indeed as shown in previous experimental results, CookiePicker's two difference detection algorithms can filter out the effects of advertisements and web page dynamics very well. Moreover, other advertisement removing techniques such as those used in [49] can be integrated into CookiePicker to further improve the detection accuracy. The other three kinds of evasion techniques can be employed by a web site operator. However, as we discuss below, they will not pose a serious threat to the usage of CookiePicker.

7.2. Evasion sources

The evasion of CookiePicker will most likely come from two sources: web site operators who want to track user activities, and attackers who want to steal cookies. As stated in Section 2, we assume that the hosting web site is legitimate, since it is pointless to provide cookie security and privacy services for a malicious web site. For legitimate web sites, if some operators strongly insist to use first-party persistent cookies for tracking long-term user behaviors, they can evade CookiePicker by detecting the hidden HTTP request and manipulating the hidden HTTP response using the evasion techniques mentioned above. However, we argue that many web site operators will not pay the effort and time to do so, either because of the lack of interest to track long-term user behaviors in the first place, or because of inaccuracy in cookie-based user behavior tracking, which has long been recognized [50]. These web site operators are either not aware of the possible privacy and security risks of stolen first-party persistent cookies, or simply not willing to pay the cost to renovate their systems. Our CookiePicker is a pure client-side solution that especially aims to protect the users of these web sites.

For third-party attackers, unless they compromise a legitimate web site, it is very difficult for them to use any of the above evasion techniques to manipulate the web pages sending back to a user's browser and circumvent CookiePicker. Therefore, CookiePicker can effectively identify and remove useless cookies stored by most legitimate web sites on a user's hard disk, and prevent them from being stolen by malicious attacks such as cross-site scripting attacks.

7.3. Concerns about using cookiePicker

One concern is that CookiePicker might reject third-party cookies in the case of HTTP redirection (see Section 5.1.2 of [15]). However, this concern is invalid because CookiePicker does not block (or even touch) third-party cookies by itself – decision regarding third-party cookies is made completely by the web browser based on users' preference setting. Similarly, whether a cookie is a first-party cookie of a web site is also decided by the web browser, and CookiePicker makes its decision based on the browser's decision. Therefore, it does not matter whether a web site is using CDN (Content Delivery Network) or not.

Another concern is that CookiePicker may not work well on web pages that use Ajax (Asynchronous JavaScript and XML). This concern is reasonable because Ajax requests may use cookies and may also change web page contents. Currently, CookiePicker only sends hidden requests to retrieve container pages, as described in Section 4. But it is possible to extend CookiePicker to evaluate the impact of cookies on Ajax requests, for example, by asking CookiePicker to send out two Ajax requests (one with cookies and the other without cookies) at the same time and then compare the response messages.

8. Related work

RFC 2109 [14] specifies the way of using cookies to create a stateful session with HTTP requests and responses. It is also the first document that raises the general public's awareness of cookie privacy problems. RFC 2965 [51] fol-

lows RFC 2109 by introducing two new headers, `Cookie2` request header and `Set-Cookie2` response header. However, these two new headers are not supported by the popular web browsers such as Internet Explorer and Firefox. RFC 2964 [13] focuses on the privacy and security of using HTTP cookies, and identifies the specific usages of cookies that are either not recommended by the IETF or believed to be harmful. Fu's study [4] suggests that setting authenticators in cookies should be very careful and especially persistent cookies should not be used to store authenticators.

Cookies not only can be retrieved and stored by the headers of HTTP requests and responses, but also can be read and written by client-side JavaScript. The *same origin policy* [23] introduced in Netscape Navigator 2.0 prevents cookies and JavaScript in different domains from interfering with each other. The successful fulfillment of the *same origin policy* on cookies and JavaScript further invokes the enforcement of this policy on browser cache and visited links [52]. Recently, in order to mitigate cross-site scripting attacks, Internet Explore also allows a cookie to be marked as "HttpOnly" in the `Set-Cookie` response header, indicating that a cookie is "non-scriptable" and should not be revealed to client applications [53].

Modern web browsers have provided users with refined cookie privacy options. A user can define detailed cookie policies for web sites either before or during visiting these sites. Commercial cookie management software such as Cookie Crusher [54] and CookiePal [55] mainly rely on pop-ups to notify incoming cookies. However, the studies in [8] show that such cookie privacy options and cookie management policies fail to be used in practice, due mainly to the following two reasons: (1) these options are very confusing and cumbersome, and (2) most users have no good understanding of the advantages and disadvantages of using cookies. A few Firefox extensions such as Cookie Culler [48] and Permit Cookies [56], although convenient to use, are just very simple add-ons for user to easily access privacy preference settings or view cookies. Acumen system [57] can inform a user how many other users accept certain cookies. However, the system does not protect the privacy of the user itself. Moreover, many users' decisions could be wrong, resulting in negative reference. Another interesting system is Privoxy [49] web proxy. It provides advanced filtering capabilities for protecting privacy, modifying web page data, managing cookies, controlling access, and removing advertisement, banners, pop-ups and other obnoxious Internet junk. However, Privoxy is more useful for those sophisticated users who have the ability to fine-tune their installation.

Recently, the most noticeable research work in cookie management is Doppelganger [11]. Doppelganger is a system for creating and enforcing fine-grained privacy-preserving cookie policies. Doppelganger leverages client-side parallelism and uses a twin window to mirror a user's web session. If any difference is detected, Doppelganger will ask the user to compare the main window and the fork window, and then, make a cookie policy decision. Although taking a big step towards automatic cookie management, Doppelganger still has a few obvious drawbacks. First, Doppelganger still heavily relies on the user's comparison between the main window and the fork window to make a decision. Second, the cost of its parallel mirroring mechanism is very high. This is because not only every user action needs to be mirrored, but every HTTP request also needs to be duplicately sent back to the web server. Third, due to the high cost, a user may not be patient enough to have a long training period, thus the policy decision accuracy cannot be guaranteed. Last but not the least, Doppelganger only achieves web site level cookie policy making. In contrast, our CookiePicker works fully automatically without user involvement or even notice. It has very low overhead, and hence, can be trained for a long period on a user's web browser to achieve high accuracy. CookiePicker achieves cookie-group level policy making, implying that usefulness is identified for a group of cookies used in a web page view.

9. Conclusions

In this paper, we have conducted a large scale cookie measurement, which highlights the demand for effective cookie management. Then, we have presented a system, called CookiePicker, to automatically managing cookie usage setting on behalf of a user. Only one additional HTTP request for the container page of a web site, the hidden request, is generated for CookiePicker to identify the usefulness of a cookie set. CookiePicker uses two complementary algorithms to accurately detect the HTML page differences caused by enabling and disabling cookies. CookiePicker classifies those cookies that cause perceivable changes on a web page as useful, and disable the rest as useless. We have implemented CookiePicker as an extension to Firefox and evaluated its efficacy through live experiments over various web sites. By automatically managing the usage of cookies, CookiePicker helps a user to strike an appropriate balance between easy usage and privacy risks.

Acknowledgments

We thank the anonymous reviewers for their careful and insightful comments. This work was partially supported by NSF Grant CNS-091537 and ONR Grant N00014-09-1-0746.

References

[1] HTTP Cookie, <http://en.wikipedia.org/wiki/HTTP_cookie>.
[2] D.M. Kristol, Http cookies: standards, privacy, and politics, ACM Trans. Inter. Tech. 1 (2) (2001) 151–198.
[3] S. Chapman, G. Dhillon, Privacy and the internet: the case of doubleclick, inc, 2002.
[4] K. Fu, E. Sit, K. Smith, N. Feamster, Do's and don'ts of client authentication on the web, in: Proceedings of the In 10th USENIX Security Symposium, Washington, DC, 2001.
[5] M. Jakobsson, S. Stamm, Invasive browser sniffing and countermeasures, in: Proceedings of the WWW'06, 2006, pp. 523–532.
[6] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Cross site scripting prevention with dynamic data tainting and static analysis, in: Proceedings of the NDSS'07, 2007.
[7] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, S.T. King, Automated web patrol with Strider HoneyMonkeys: finding web sites that exploit browser vulnerabilities, in: Proceedings of the NDSS'06, 2006.
[8] V. Ha, K. Inkpen, F.A. Shaar, L. Hdeib, An examination of user perception and misconception of internet cookies, in: CHI'06

extended abstracts on human factors in computing systems, Montreal, Canada, 2006, pp. 833–838.
[9] Platform for privacy preferences (P3P) project, <http://www.w3.org/P3P/>.
[10] L.I. Millett, B. Friedman, E. Felten, Cookies and web browser design: toward realizing informed consent online, in: Proceedings of the CHI'01, 2001, pp. 46–52.
[11] U. Shankar, C. Karlof, Doppelganger: better browser privacy without the bother, in: Proceedings of the ACM CCS'06, Alexandria, VA, 2006.
[12] L. Bent, M. Rabinovich, G.M. Voelker, Z. Xiao, Characterization of a large web site population with implications for content delivery, in: Proceedings of the WWW'04, 2004, pp. 522–533.
[13] K. Moore, N. Freed, Use of http state management, RFC 2964 (2000).
[14] D. Kristol, L. Montulli, Http state management mechanism, RFC 2109 (1997).
[15] A.F. Tappenden, J. Miller, Cookies: a deployment study and the testing implications, ACM Trans. Web 3 (3) (2009) 1–49.
[16] E. Kirda, C. Kruegel, G. Vigna, N. Jovanovic, Noxes: a client-side solution for mitigating cross-site scripting attacks, in: Proceedings of the 2006 ACM symposium on Applied computing (SAC' 06), ACM Press, 2006, pp. 330–337.
[17] CERT Advisory CA-2000-02 Malicious HTML tags embedded in client web requests, 2000, <http://www.cert.org/advisories/CA-2000-02.html>.
[18] S. Chen, J. Meseguer, R. Sasse, H.J. Wang, Y.-M. Wang, A systematic approach to uncover security flaws in gui logic, in: Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P '07), IEEE Computer Society, 2007, pp. 71–85.
[19] C. Reis, J. Dunagan, H.J. Wang, O. Dubrovsky, S. Esmeir, Browsershield: vulnerability-driven filtering of dynamic html, in: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), USENIX Association, 2006, pp. 61–74.
[20] E. Kirda, C. Kruegel, G. Banks, G. Vigna, R.A. Kemmerer, Behavior-based spyware detection, in: Proceedings of the 15th conference on USENIX Security Symposium, USENIX Association, 2006, pp. 273–288.
[21] M.T. Louw, J.S. Lim, V. Venkatakrishnan, Extensible web browser security, in: Proceedings of the Fourth GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'07), 2007.
[22] S. Saroiu, S.D. Gribble, H.M. Levy, Measurement and analysis of spywave in a university environment, in: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI'04), USENIX Association, 2004, pp. 141–153.
[23] Same origin policy, <http://en.wikipedia.org/wiki/Same_origin_policy>.
[24] Google plugs cookie-theft data leak, 2005, <http://www.eweek.com/article2/0,1895,1751689,00.asp>.
[25] Google slams the door on XSS flaw 'Stop cookie thief!', 2007, <http://software.silicon.com/security/,January17th,2007>.
[26] Flaws in IE7 and Firefox raise alarm, 2007, <http://news.zdnet.co.uk/security,Febuary19th,2007>.
[27] GNU Wget - GNU Project - Free Software Foundation (FSF), <http://www.gnu.org/software/wget/>.
[28] A. Moshchuk, T. Bragin, S.D. Gribble, H.M. Levy, A crawler-based study of spyware in the web., in: Proceedings of the NDSS'06, 2006.
[29] C. Yue, M. Xie, H. Wang, Automatic cookie usage setting with CookiePicker, in: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), IEEE Computer Society, 2007, pp. 460–470.
[30] Data flow inside Gecko, <http://developer.mozilla.org/en/docs>.
[31] B.A. Mah, An empirical model of http network traffic., in: Proceedings of the INFOCOM'97, 1997, pp. 592–600.
[32] W. Yang, Identifying syntactic differences between two programs, Softw. Pract. Exper. 21 (7) (1991) 739–755.
[33] K.-C. Tai, The tree-to-tree correction problem, J. ACM 26 (3) (1979) 422–433.
[34] K. Zhang, D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, SIAM J. Comput. 18 (6) (1989) 1245–1262.
[35] T. Jiang, L. Wang, K. Zhang, Alignment of trees – an alternative to tree edit, Theor. Comput. Sci. 143 (1) (1995) 137–148.
[36] E. Tanaka, K. Tanaka, The tree-to-tree editing problem, Int. J. Pattern Recognit. Artif. Intell. 2 (2) (1988) 221–240.
[37] S.M. Selkow, The tree-to-tree editing problem, Inf. Process. Lett. 6 (6) (1977) 184–186.
[38] G. Valiente, An efficient bottom-up distance between trees, in: Proceedings of the SPIRE'01, 2001, pp. 212–219.
[39] P. Bille, A survey on tree edit distance and related problems, Theor. Comput. Sci. 337 (1-3) (2005) 217–239.
[40] Y. Zhai, B. Liu, Web data extraction based on partial tree alignment, in: Proceedings of the WWW'05, 2005, pp. 76–85.
[41] D.C. Reis, P.B. Golgher, A.S. Silva, A.F. Laender, Automatic web news extraction using tree edit distance, in: Proceedings of the WWW'04, 2004, pp. 502–511.
[42] A. Torsello, D. Hidovic-Rowe, Polynomial-time metrics for attributed trees, IEEE Trans. Pattern Anal. Mach. Intell. 27 (7) (2005) 1087–1099.
[43] R. Al-Ekram, A. Adma, O. Baysal, diffx: an algorithm to detect changes in multi-version xml documents, in: Proceedings of the CASCON'05, 2005, pp. 1–11.
[44] Y. Zhai, B. Liu, Structured data extraction from the web based on partial tree alignment, IEEE Trans. Knowledge Data Eng. 18 (12) (2006) 1614–1628.
[45] HTML, <http://en.wikipedia.org/wiki/HTML>.
[46] S. Gupta, G. Kaiser, D. Neistadt, P. Grimm, Dom-based content extraction of html documents, in: Proceedings of the WWW'03, 2003, pp. 207–214.
[47] S. Joshi, N. Agrawal, R. Krishnapuram, S. Negi, A bag of paths model for measuring structural similarity in web documents, in: Proceedings of the KDD'03, 2003, pp. 577–582.
[48] Cookie Culler, <http://cookieculler.mozdev.org>.
[49] Privoxy - Home Page, <http://www.privoxy.org/>.
[50] Accurate web site visitor measurement crippled by cookie blocking and deletion, jupiterresearch finds, 2005, <http://www.jupitermedia.com/corporate/releases/05.03.14-newjupresearch.html>.
[51] D. Kristol, L. Montulli, Http state management mechanism, RFC 2965 (2000).
[52] C. Jackson, A. Bortz, D. Boneh, J.C. Mitchell, Protecting browser state from web privacy attacks, in: Proceedings of the WWW'06, 2006, pp. 737–744.
[53] Mitigating cross-site scripting with HTTP-only cookies, <http://msdn2.microsoft.com/en-us/library/ms533046.aspx>.
[54] Cookie crusher, <http://www.pcworld.com/downloads>.
[55] Cookie pal, <http://www.kburra.com/cpal.html>.
[56] Permit cookies, <https://addons.mozilla.org/firefox/44>.
[57] J. Goecks, E.D. Mynatt., Social approaches to end-user privacy management, in: Security and Usability: Designing Secure Systems That People Can Use, 2005.

Chuan Yue is a Ph.D. candidate in computer science at The College of William and Mary. His broad research interests include Web-based Systems, Computer and Information Security, Distributed and Parallel Computing, Human-Computer Interaction, and Collaborative Computing. His current research focuses on web browsing security and collaborative browsing. Previously, he received his B.S. and M.S. degrees in computer science from Xidian University and then worked as a Member of Technical Staff at Bell Labs China, Lucent Technologies for four years, mainly on the development of web-based Service Management System for Intelligent Network.

Mengjun Xie received his Ph.D. in Computer Science at the College of William and Mary in 2009. His research focuses on securing network systems, especially Internet-based message systems, and improving the performance of network systems. His research interests include network security, information security, network systems, and operating systems.

Haining Wang is an Associate Professor of Computer Science at the College of William and Mary, Williamsburg, VA. He received his Ph.D. in Computer Science and Engineering from the University of Michigan at Ann Arbor in 2003. His research interests lie in the area of networking, security and distributed computing. He is particularly interested in network security and network QoS (Quality of Service).