Automatic Cookie Usage Setting with CookiePicker

Chuan Yue Mengjun Xie Haining Wang
The College of William and Mary
{cyue,mjxie,hnw}@cs.wm.edu

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 Web sites to track and build user profiles. In addition, stolen cookies may also incur 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 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 [29]. 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 [11], the ability to monitor browsing habits, and possibly to associate what you've 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 [7, 21, 22].

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 [5].

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 some cookies are harmful is almost impossible, because very few Web sites inform users how they use cookies. Platform for Privacy Preferences Project (P3P) [30] enables Web sites to express their privacy practices but its usage is too low to be a feasible 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 [13] and very inconvenient to users.

In this paper, we present a system called *CookiePicker* to automatically make cookie usage decisions on behalf of a Web user. The distinct features of CookiePicker include (1) fully automatic decision making, (2) high accuracy on decision making, and (3) very low running overhead. Based

on the two complementary 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. Then, CookiePicker enables useful cookies but disables useless cookies. All the tasks are performed without user involvement or even notice. We implement CookiePicker as a Firefox Web browser extension, and validate its efficacy through experiments over various Web sites.

The remainder of this paper is structured as follows. Section 2 gives the background of cookies and the focus of CookiePicker. Section 3 describes the design of CookiePicker. Section 4 details the two HTML page difference detection algorithms, which are the core of CookiePicker. Section 5 presents the implementation of CookiePicker and its performance evaluation. Section 6 surveys related work, and finally, Section 7 concludes the paper.

2. Background

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 also be classified into session cookies, which are stored in memory and deleted after the close of the Web browser; and persistent cookies, which are stored on a hard disk until they expire or are deleted by a user.

Third-party cookies bring almost no benefit to Web users and have long been recognized as a major threat to user privacy since 1996 [10]. 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 user profiling done by third parties [11], 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. We have conducted a large scale measurement study on the usage of cookies, with over five thousands Web sites involved. Our measurement results show that first-party persistent cookies are widely used and above 60% of them are set to expire after one year or even longer [24]. Some cookies perform useful roles such as personalization and authentication. Some cookies, however, provide no benefit but pose serious privacy and security risks to users. The major risks lie in two aspects. First, these first-party persistent cookies can be used to track the user activity over time by the original Web site; second, they can be stolen or manipulated by two kinds of long-standing attacks—cross-site scripting (XSS) attacks that exploit Web applications vulnerabilities [27, 21] and various attacks that exploit Web browser vulnerabilities [22], since those attacks can bypass the *same origin policy* [31] enforced by all modern Web browsers. For example, recently a cookie-related XSS vulnerability was even found in one of Google's hosting services [28].

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 limited functions such as manual deletion or blocking, which 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. Instead of addressing XSS or 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.

3. 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.*

3.1 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 will analyze the container page and issue 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 Figure 1, respectively. Web page contents coming with the HTTP responses will be passed into the Web browser layout engine, which will parse the container page and built it into a DOM (W3C Document Object Model) tree.

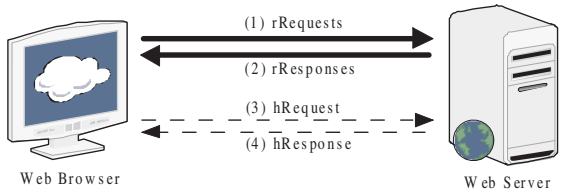

Figure 1: HTTP requests/responses in a single Web page view.

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 [16], 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 Figure 1, 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 Figure 1, as regular requests and responses; and refer the extra request and response, represented by the dashed lines (3) and (4) of Figure 1, as the hidden request and response.

3.2 Forward Cookie Usefulness Marking

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 [12] 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 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 are 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 from the hidden request are useful. The two algorithms are the

core of the CookiePicker and will be detailed in Section 4.

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 she wants to continue the training process.

3.3. 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 mis-classified 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. One 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 one 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.

4. 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 [23] 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.

4.1. Restricted Simple Tree Matching

As mentioned in Section 3, 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 [17] 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.

4.1.1. 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' [17]. 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 as defined in [17].

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* [8], *isolated subtree distance* [18], *top-down distance* [15, 23], and *bottom-up distance* [20]. The description and comparison of these algorithms are beyond the scope of this paper, see [2] and [20] for details.

4.1.2. 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 3. *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':*

$$\textit{if pair } (i,j) \in M \textit{ then } (parent(i), parent(j)) \in M.$$

The top-down distance problem was introduced by Selkow [15]. In [23], 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 between 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 [25] used it for extracting structured data from Web pages; and Reis *et al.* [14] applied it for automatic Web news extraction. In contrast, bottom-up distance [20], although can be more efficient in time complexity ($O(|T| + |T'|)$), falls short of being an accurate metric [19] and may produce a far from optimal result [1] for HTML DOM tree comparison, in which most of the differences come from the leaf nodes.

4.1.3. Restricted Simple Tree Matching

Based on the original STM algorithm [23], Figure 2 illustrates, RSTM, our restricted version of STM algorithm.

Algorithm: RSTM($A, B, level$)
1. **if** the roots of the two trees A and 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** $(M[m, n] + 1)$;

Figure 2: The Restricted Simple Tree Matching 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. Figure 3 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 computed return result by using 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 Figure 3, 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. This kind of noises, although very annoying, has a distinct characteristic that they often appear at the lower level of the DOM trees. In contrast, the Web page changes caused by enabling/disabling cook-

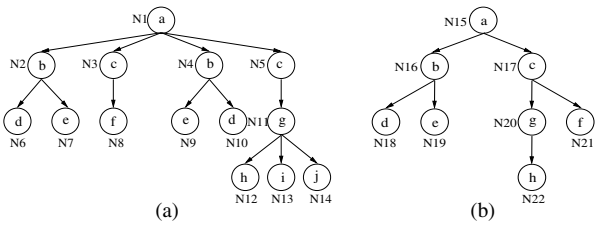

Figure 3: (a) Tree A, (b) Tree B.

ies are often so obvious that the structural dissimilarities are clearly reflected at the upper level of the DOM trees. 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.

The second reason 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 5, 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.

4.1.4. 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.

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

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

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

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

Figure 4: The Text Content Extraction Algorithm.

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

4.2. 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, while image contents often serve as supplements to text contents. In practice, users can block the loading of various images and browse most of the Web page 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. Figure 4 depicts the recursive algorithm to extract the text content.

The contentExtract algorithm preorder traverses the

whole DOM tree 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 [4], scripts, styles, obvious advertisement text, date and time string, and option text in dropdown list (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.

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 [9] 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.

4.3. 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 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 both *NTreeSim* and *NTextSim* are less than the two tunable thresholds, *Thresh1* and *Thresh2*, respectively, CookiePicker will make a decision that the difference is due to cookie usage. Figure 5 depicts the final decision algorithm.

5. System Evaluation

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

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**

Figure 5: CookiePicker Decision Algorithm.

5.1. 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.

5.2. Evaluation

We installed CookiePicker on a Firefox version 1.5.0.8 Web browser 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 all 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.

5.2.1. First Set of Experiments

From each of the 15 categories we measured [24] in directory.google.com, 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 1, 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 1, 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 1. 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 mis-classified as "useful". Thus, the number of backward error recovery is minimized.

In Table 1, 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. 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 more than 10 seconds [12]. Note that Web sites S4, S17, and S28 have abnormally high identification duration at about 10 seconds, which is mainly caused by the very slow responses from those three Web sites.

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

Table 1: Online testing results for thirty Web sites (S1 to S30).

5.2.2. 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 we measured [24] in directory.google.com. 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, 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 2 and represented as P1 to P6 for privacy concerns.

In Table 2, 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

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	-

Table 2: Online testing results for 6 Web sites (P1 to P6) that have useful persistent cookies.

"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 NTreeSim$(A, B, 5)$ and 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 Figure 5. 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 cookies from the majority of Web sites (25 out of the 30 Web sites that we intensively tested). Meanwhile, all the useful persistent cookies can be 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. About 10% Web sites (3 out of 30), in which persistent cookies are useless, are wrongly identified by CookiePicker as "useful". However, the number may be further reduced if we fine-tune the two thresholds and the implementation of the two algorithms, which we leave as our future work.

5.3. Evasion against CookiePicker

In CookiePicker, the identification of useful cookies are based on perceivable changes on a Web page. Once the identification metric becomes known, CookiePicker may be circumvented. 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 discuss cookie security and privacy issues within 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. However, we argue that most 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, or because of inaccuracy in cookie-based user behavior tracking, which has long been recognized [26].

For third-party attackers, unless they compromise a legitimate Web site, it is very difficult for them to manipulate the Web pages sending back to a user's browser and circumvent CookiePicker.

6. Related Work

RFC 2109 [10] is the first document that raises the general public's awareness of cookie privacy problems. Later on, the *same origin policy* was introduced in Netscape Navigator 2.0 to prevent cookies and Javascripts of different sites from interfering with each other. The successful fulfillment of the *same origin policy* on cookies and Javascripts further encourages the enforcement of this policy on browser cache and visited links [6].

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 softwares such as Cookie Crusher [32] and CookiePal [33] mainly rely on pop-ups to notify incoming cookies. However, the studies in [5] 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 true understanding of the advantages and disadvantages of using cookies.

Recently, the most noticeable research work in cookie management is Doppelganger [16]. 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 such as high overhead and the need for human involvement. Although CookiePicker follows Doppelganger's basic princi-

ple of comparing Web page differences to identify useful cookies, it works in a fully automatic way and has much lower overhead.

7. Conclusions

In this paper, we have presented a system, called CookiePicker, to automatically manage 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. The core of CookiePicker are two complementary detection algorithms, which 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 manage the usage of cookies, CookiePicker helps a user to strike an appropriate balance between easy usage and privacy risks. We believe CookiePicker has the potential to be widely used, for its fully automatic nature, its high accuracy, and its low overhead.

Acknowledgments: This work was partially supported by NSF grants CNS-0627339 and CNS-0627340.

References

[1] R. Al-Ekram, A. Adma, and O. Baysal. diffx: an algorithm to detect changes in multi-version xml documents. In *Proceedings of the CASCON'05*, pages 1–11, 2005.

[2] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[3] S. Chapman and G. Dhillon. Privacy and the internet: the case of doubleclick, inc, 2002.

[4] S. Gupta, G. Kaiser, D. Neistadt, and P. Grimm. Dom-based content extraction of html documents. In *Proceedings of the WWW'03*, pages 207–214, 2003.

[5] V. Ha, K. Inkpen, F. A. Shaar, and L. Hdeib. An examination of user perception and misconception of internet cookies. In *CHI'06 extended abstracts on Human factors in computing systems*, pages 833–838, 2006.

[6] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the WWW'06*, pages 737–744, 2006.

[7] M. Jakobsson and S. Stamm. Invasive browser sniffing and countermeasures. In *Proceedings of the WWW'06*, pages 523–532, 2006.

[8] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995.

[9] S. Joshi, N. Agrawal, R. Krishnapuram, and S. Negi. A bag of paths model for measuring structural similarity in web documents. In *Proceedings of the KDD'03*, pages 577–582, 2003.

[10] D. Kristol and L. Montulli. Http state management mechanism, RFC 2109, 1997.

[11] D. M. Kristol. Http cookies: Standards, privacy, and politics. *ACM Trans. Inter. Tech.*, 1(2):151–198, 2001.

[12] B. A. Mah. An empirical model of http network traffic. In *Proceedings of the INFOCOM'97*, pages 592–600, 1997.

[13] L. I. Millett, B. Friedman, and E. Felten. Cookies and web browser design: toward realizing informed consent online. In *Proceedings of the CHI'01*, pages 46–52, 2001.

[14] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the WWW'04*, pages 502–511, 2004.

[15] S. M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[16] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *Proceedings of the ACM CCS'06*, 2006.

[17] K.-C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[18] E. Tanaka and K. Tanaka. The tree-to-tree editing problem. *International journal Pattern Recognition And Atificial Intelligency*, 2(2):221–240, 1988.

[19] A. Torsello and D. Hidovic-Rowe. Polynomial-time metrics for attributed trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(7):1087–1099, 2005.

[20] G. Valiente. An efficient bottom-up distance between trees. In *Proceedings of the SPIRE'01*, pages 212–219, 2001.

[21] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the NDSS'07*, 2007.

[22] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the NDSS'06*, 2006.

[23] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.

[24] C. Yue, M. Xie, and H. Wang. Cookie measurement and design of CookiePicker. Technical Report WM-CS-2007-03, The College of William & Mary, 2007.

[25] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proceedings of the WWW'05*, pages 76–85, 2005.

[26] Accurate web site visitor measurement crippled by cookie blocking and deletion, jupiterresearch finds, 2007. http://www.jupitermedia.com/corporate/releases/05.03.14-newjupresearch.html.

[27] CERT Advisory CA-2000-02 Malicious HTML tags embedded in client web requests. http://www.cert.org/advisories/CA-2000-02.html.

[28] Google slams the door on XSS flaw 'Stop cookie thief!'. http://software.silicon.com/security/, January 17th, 2007.

[29] Http cookie, 2006. http://en.wikipedia.org/wiki/HTTP_cookie.

[30] Platform for privacy preferences (P3P) project, 2006. http://www.w3.org/P3P/.

[31] Same origin policy, 2007. http://en.wikipedia.org/wiki/Same_origin_policy.

[32] Cookie crusher, 2006. http://www.pcworld.com/downloads.

[33] Cookie pal, 2006. http://www.kburra.com/cpal.html.