Cache Cookies for Browser Authentication

Ari Juels
RSA Laboratories and
RavenWhite Inc.
ajuels@rsasecurity.com

Markus Jakobsson
Indiana University and
RavenWhite Inc.
markus@indiana.edu

Tom N. Jagatic
Indiana University
tjagatic@iu.edu

***Abstract*—Like conventional cookies, *cache cookies* are data objects that servers store in Web browsers. Cache cookies, however, are essentially unintentional byproducts of protocol design for browser caches. They do not enjoy any explicit interface support or security policies.**

In this paper, we show that despite limitations, cache cookies can play a useful role in the identification and authentication of users. Many users today block conventional cookies in their browsers as a privacy measure. The cache-cookie tools we propose can help restore lost usability and convenience to such users while maintaining good standards for privacy. As we show, our techniques can also help combat online security threats as phishing and pharming that ordinary cookies cannot. In fact, the ideas we introduce for cache-cookie management can strengthen ordinary cookies as well.

Because cache cookies have been viewed traditionally as a threat to user privacy, and lack important read-access restrictions, we propose cache-cookie protocols that aim to protect privacy by design.

Keywords: cache cookies, personalization, malware, pharming, phishing, privacy, Web browser

I. Introduction

In this paper, we investigate new ways of using *cache cookies* for user authentication. A conventional cookie is a piece of information stored in a specially designated cache in a Web browser. Cookies can include user-specific identifiers, or personal information about users (e.g., this user is over 18 years of age). Servers typically employ cookies to personalize Web pages. For example, when Alice visits the Web site X, the domain server for X might place a cookie in Alice's browser that contains the identifier "Alice." When Alice visits X again, her browser releases this cookie, enabling the server to identify Alice automatically.

A cache cookie, by contrast, is not an explicit browser feature. It is a form of persistent state in a browser that a server can access in unintended ways. There are many different forms of cache cookies; they are byproducts of the way that browsers maintain various caches and access their contents.

For example, browsers maintain caches known as *histories*; these contain lists of the Web pages that users have recently visited. If Alice visits iacr.org, her browser history will store the associated URL for some period of time as a convenience to Alice. Thanks to this feature, when Alice types the letter 'i' into her browser navigation field, her browser can pop up "www.iacr.org" as an option for her to visit. A server can *write* to the browser-history cache of a client by means of redirection. For example, if "www.iacr.org" redirects Alice's browser to "www.iacr.org/Alice," the effect will be to write the string "www.iacr.org/Alice" into her browser cache.

As it turns out, it is also possible for a server to *read* the contents of a user's browser history. For example, if Alice visits the IACR Web site, the server there can test for the presence of the (verbatim) string "www.iacr.org/Alice" in her browser history. This capability is, as we have said, a side-effect of browser design [3] whose nature we discuss in detail below. (It also represents a privacy threat, particularly as a server can test for the presence of *any* string in a browser history, not just those associated with its own domain name [6].) Nonetheless, the result is that a server can both read and write to, and therefore manipulate a browser-history cache as a form of general-purpose storage.

In this regard, a cache-cookie can function very much like an ordinary cookie. It is common for servers to plant cookies containing secret values in the browsers of users. The presence of these cookies helps a server authenticate a user – or, more precisely, her browser. Cache-cookies, as we show, can serve much the same goal.

A. Our work: Cache cookies as authenticators

Cookies were not initially designed for authentication, but merely as a convenient way to pass state. They have been effectively co-opted in many systems to achieve security goals. We take the same approach to cache cookies: We co-opt them for the unintended but beneficial application of browser authentication.

The basis for our work is a new conceptual framework in which cache cookies underpin a general, virtual memory structure within a browser. We refer to this type of structure as *cache-cookie memory*, abbreviated *CC-memory*. We show how a server can read and write arbitrary data strings to CC-memory – and also how it can also erase them.

An important feature of CC-memory is that it spans a very large space. Because it is a virtually addressed memory structure, and not a physically addressed one, its size is exponential in the bit-length of browser resource names such as URLs. Indeed, so large is the space of CC-memory for an ordinary browser that a server can only access a negligible portion. Consequently, servers can exploit the size of CC-memory to hide information in browsers. An attacker cannot feasibly scan more than a negligible portion of CC-memory. Based on this characteristic, we propose new techniques for privacy-enhanced identifiers and user-authentication protocols that resist brute-force attacks against browser caches. Quite importantly, our techniques require *no special-purpose client-side software*.

At the same time, we investigate techniques to audit and detect abuses of cache cookies. We propose a technique for creating cache-cookie identifiers whose privacy is linked to that of a Web server's private SSL key.

Another contribution of this paper is a type of cache cookie based on Temporary Internet Files (TIFs) that we believe to be new to the literature. These cache cookies work with most of our proposed schemes. As we show, moreover, in terms of read privileges they have the privacy features of conventional (first-party) cookies, and are easier to manipulate than cache-cookies based on browser histories. In general, therefore, we favor use of TIF-based cache cookies for our proposed schemes.

Given increasing concerns about user privacy, it is possible that some of the techniques we propose here depend on browser features that will be (perhaps rightly) eliminated in future. That said, cache cookies exist because of the conveniences they afford users. For example, browser histories, as we have explained, facilitate navigation by reducing users' need to type frequently visited URLs; they also permit the highlighting of previously visited links in Web pages. Caches for images and other files result in better performance for clients and servers alike. For these reasons, while browser vendors may in the future consider more restrictive default privacy policies and therefore trim cache functionality, caches and cache cookies are here to stay. We believe that the techniques we propose, therefore, will have lasting value.

B. Related work

While we emphasize the use of cache cookies for authentication in this paper, most of the literature thus far has treated cache cookies purely in the light of their threat to privacy.

Felten and Schneider (FS) first brought the problems of browser sniffing and cache cookies to light [5]. They demonstrated that a Web server could detect the presence of a cached image in a client browser, even if the image originated at a different site. A browser displays a cached image more rapidly than an uncached one that it needs to retrieve remotely. Thus, as Felten and Schneider showed, a server can embed a pointer to a target image in a Web page (invisibly, if desired), and based on delicate statistical analysis of the resulting loading time, determine whether the image has been previously cached. Many target images are site-specific; a cached "Democratic Party" banner image probably indicates that a user visited www.democrats.org. Thus, a cached image offers insight into a user's browser history. FS also noted that a server can plant images in a user's browser as a way of storing information there. It is they, in fact, who first coined the term cache cookies. The timing analysis proposed by FS is somewhat difficult to mount, so such cache cookies are unreliable.

Clover [3], however, brought to light more easily manipulated cache cookies based on browser histories. A feature of Cascading Style Sheets (CSS), rules for formatting and presentation of Web content, permits previously visited links to be highlighted in Web content. As a side-effect of this feature, a server can embed code in a Web page that determines whether or not a browser contains a particular URL in its history – irrespective of the domain with which the URL is associated. For example, *any* server can determine if Alice has visited the specific Web page www.arbitrarysite.com/randompath/index.html. Such privacy infringements are often referred to as *browser sniffing*.

This CSS browser-sniffing exploit is straightforward to mount. Moreover, it enables a server to plant cache cookies. To plant a desired URL in a browser history cache, a server need merely redirect the browser. This redirection can be invisible to the user (when executed in, e.g., an iframe). And a server can plant URLs from any domain, not just its own. Additionally, as Clover notes, tools for anonymous browsing do not protect against this exploit. As a countermeasure, Clover proposed that links only be highlighted when associated with the domain of the currently visited site.

We note that cache cookies as we propose them do not have to involve browser sniffing. Cache cookies are bits of information that servers can read and write to browser caches without regard to the designed use of these caches. In other words cache cookies involve the treatment of caches as general data repositories. In contrast, browser sniffing exploits the naturally accumulated data and intended semantics of caches.

A common form of tracking used by marketers today and related to sniffing is what is known as a "Web bug," an HTML link to an (invisible) image in e-mail. By downloading the image, a client alerts a server to opening of the e-mail. Web bugs can also be planted in Web pages.

Jakobsson, Jagatic, and Stamm [7] highlight a new, more dangerous mode of phishing in which a phisher harvests user information to target deceptive e-mail more effectively. They call this attack *context-aware* phishing. As an example, they show how, by means of browser sniffing, a phisher can learn information about which banks a user has relationships with, and then potentially direct e-mail to that user that is attractive because it appears to originate with her bank. Jakobsson and Stamm [9] propose a server-side countermeasure to context-aware phishing. Their solution is for a server to add randomized extensions to URLs. Since URLs can only be sniffed in verbatim form, this countermeasure renders URLs at a protected site opaque to would-be phishers and other malefactors. These two papers do not explicitly treat issues around cache cookies.

More recently, Jackson, Bortz, Boneh, and Mitchell [6] present a unifed view of ways in which browser sniffing and various types of cache cookies permit cross-domain tracking of users. They identify vehicles for cache cookies, such as *entity tags* (Etags), a type of meta-information associated with URL requests. Jackson et al. propose browser extensions that permit users to create privacy policies that apply across a range of known cross-domain tracking methods, and not just to conventional cookies. They note that eliminating cache cookies is impossible without severely impacting browser usability, but their techniques address broad classes of browser sniffing and tracking.

Again, our emphasis in this paper is on the *positive* face of cache cookies. We do not propose solutions to the thorny privacy problems they pose. Instead, we propose ways to use cache cookies beneficially, and in ways that do not exacerbate existing privacy problems.

Organization

In section II, we present our framework for CC-memory, along with some new implementation options. We introduce our schemes for user identification and authentication in section III. In particular, we propose two schemes for private cache-cookie identifiers. One scheme is based on a data structure that we call an *identifier tree*, while another involves a simple system of rolling pseudonyms. In a similar technical vein, we also introduce a pharming-resistant authentication scheme. In section IV, we sketch some methods of detecting and preventing abuse of cache cookies. We present some supporting experiments for our ideas in section V, and conclude in section VI. In Appendix A, we explain how cache cookies can enhance useability for browsers with multiple users.

II. Cache-Cookie Memory Management

We now explain how cache cookies can support the creation of CC-memory. As explained above, CC-memory is a general read/write memory structure in a user's browser. As we demonstrate experimentally in section V, it is possible for a server not merely to detect the presence of a particular cache cookie, but to test quickly for the presence of any in a list of perhaps hundreds of cache cookies. More interesting still, a server can unobtrusively mine cache cookies in an *interactive* manner, meaning that it can refine its search on the fly, using preliminary information to guide its detection of additional cache cookies. In consequence, as we show, CC-memory can be very flexible. As we have explained, CC-memory can also be very large – *so large as to resist brute-force search by an adversary.* We exploit this characteristic in section III to support various privacy and security applications.

Thanks to our general view of cache cookies as a storage mechanism, we are also able to propose fruitful uses for a new type of cache cookies based on what are called Temporary Internet files, namely general-purpose data files downloaded by browsers.

A. Cache-cookie memory

We now explain how to construct CC-memory structures. We use browser-history caches as an illustrative example, but the same principles apply straightforwardly to other types of cache cookies.

Thus, let us consider a particular browser history cache containing recently visited URLs. A server can, of course, plant any of a wide variety of cookies in this cache by redirecting the user to URLs within its domain space (or externally, if desired). For example, a server operating the domain www.arbitrarysite.com can redirect a browser to a URL of the form "www.arbitrarysite.com?Z" for any desired value of Z, thereby inserting "www.arbitrarysite.com?Z" into the history cache. Thus, a server can create a CC-memory structure over the space of URLs of the form, e.g., "www.arbitrarysite.com?Z," where $Z \in \{0,1\}^{l+1}$. In other words, Z is an index into the space. In practice, the space can be enormous – larger than a cryptographic key space. (Current versions of IE, for instance, support 2048-bit URL paths.)

Let the predicate $P_{i,t}[r]$ denote whether the URL corresponding to a given index $r \in R$ is present in the cache of user i. If so, $P_i[r] = 1$; otherwise $P_i[r] = 0$.

For clarity of exposition, we do not include time in our notation here.

Of course, a server interacting with user i can change any $P_i[r]$ from 0 to 1; the server merely plants the URL indexed by r in the cache. The reverse, however, is not possible in general; there is no simple mechanism for servers to erase cache cookies.

We can still, however, achieve a fully flexible read/write structure. The idea is to assign two predicates to a given bit b that we wish to represent. We can think of the predicates as on-off switches. If neither switch is on, the bit b has no assigned value. When the first switch is on, but not the second, $b = 0$; in the reverse situation, $b = 1$. Finally, if both switches are on, then b is again unassigned; it has been "erased."

More formally, let $S = \{0,1\}^l$. Let us define a predicate $Q_i[s]$ over S, for any $s \in S$. This predicate can assume a bit value, i.e., $Q_{i,t}[s] \in \{0,1\}$; otherwise, it is "blank," and we write $Q_i[s] = \phi$, or it is "erased," and we write $Q_{i,t}[s] = \nu$. Let $\|$ denote string concatenation. Let us define Q_i as follows. Let $r_0 = P_{i,t}[s \parallel \text{`0'}]$ and let $r_1 = P_i[s \parallel \text{`1'}]$. If $r_0 = r_1 = 0$, then $Q_i[s] = \phi$; if $r_0 = r_1 = 1$, then $Q_i[s] = \nu$. Otherwise, $Q_i[s] = b$, where $r_b = 1$.

This definition yields a simple write-once structure M with erasure for cache cookies over the set S. When $Q_i[s] = \phi$, a server interacting with user i can write an arbitrary bit value b to $Q_i[s]$: It merely has to set $P_{i,t}[s \parallel b] = 1$. The server can erase a stored bit b in $Q_i[s]$, by setting $P_i[s \parallel 1-b] = 1$.

Within the structure M, we can define an m-bit memory structure M' capable of being written c times. We let M' consist of a sequence of $n = cm$ bits in M. Once the first block of m bits in M' has been written, the server re-writes M' by erasing this first block and writing to the second block; proceeding in the obvious way, it can write c times to M'. To read M', the server performs a binary search, testing the leading bit of the memory blocks in M' until it locates the current write boundary; thus a read requires at most $\lceil \log c \rceil$ queries.

The memory structures M and M' permit only random access, of course, not search operations. Thus, when l is sufficiently large – in practice, say, when cache cookies are 80 bits long – the CC-memory structure M is large enough to render brute-force search by browser sniffing impractical. Suppose, for example, that a server plants a secret, k-bit string $x = x_0 x_1 \ldots x_k$ into a random location in memory in the browser of user i; that is, it selects $s \in_U 2^l - k - 1$, and sets $Q_i[s+i] = x_i$ for $1 \leq i \leq k$. It is infeasible for a second server interacting with user i to learn x — or even to detect its presence.

We can employ hidden data of this kind as a way to protect the privacy of cache cookies.

A.1 Variant memory structures:

There are other, more query-efficient encoding schemes for the CC-memory structures M and M'. For example, we can realize an m-bit block of data in M as follows. Let $\{P_i[s], P_i[s+1], \ldots, P_i[s+c]\}$ represent the memory block in question. We pre-pend a leading predicate $P_i[s-1]$. The value $P_i[s-1] = \text{`0'}$ indicates that the block is active: It has not yet been erased. To encode the block, a server may change any predicate to a '1.' $P_i[s-1] = \text{`1'}$ indicates that the block has been erased. A drawback to this approach is that erasure does not truly efface information: The information in an "erased" block remains readable. Full effacement is sometimes desirable, as in the rolling pseudonym scheme we shall present. Another possible approach is that proposed by Rivest and Shamir in a patent application [14]. In the basic form of their scheme, the value associated with a block of stored bits is represented by the XOR of the bits; thus it is possible to rewrite a block m times. Reading of such structures, though, requires more memory accesses in general than the one we propose.

B. C-memory

Conventional cookies have optionally associated *paths*. A cookie with associated path P is released only when the browser in which it is resident requests a URL with prefix P. For example, a cookie might be set with the path "www.arbitrarysite.com/X". In that case, the cookie is only released when the browser visits a URL of the form "www.arbitrarysite.com/X/...". Using paths, it is obviously possible to create *C-memory*, an analog to CC-memory based on conventional cookies, and to the best of our knowledge, a new approach to the use of cookies. Effectively, C-memory is a way of restricting access to cookies based on secret keys, rather than just domain names (which can be spoofed). *C-memory* of course has the same property of random-access only that CC-memory has. Moreover, it can similarly be constructed with the same property of very large size. Most of our proposed protocols for CC-memory in this paper are implementable in C-memory.

For certain of our proposed schemes, C-memory has some very notable advantages. Unlike a cache cookie, a conventional cookie carries not a single bit of information, but a full bitstring. (More precisely, a cookie carries a name/value pair of bitstrings, and up to 4k bits of data by default in, e.g., current versions of Firefox.) Thus, it is natural to construct C-memory such that each memory address stores a bitstring. Hence

C-memory is much more efficient than CC-memory in that it allows for block reads and writes. Moreover, browser protocols for cookies support direct erasure. And, finally, cookies can be set with any desired expiration date. Shared Flash objects have properties similar to those of cookies, but no expiration.

Nonetheless, for our purposes, cache cookies have two main benefits lacking in conventional cookies: (1) Cache cookies are not commonly suppressed in browsers today like conventional cookies and (2) Some cache cookies can support cross-domain access delegation, that is, they can be accessed from any domain provided that it possesses the underlying secret keys. This can be advantageous from an engineering perspective, but requires careful attention to user privacy.

C. Browser-history cache cookies

As explained above, browsers contain *histories* of the URLs that users have recently visited. Current versions of Firefox and Internet Explorer retain history information for nine days by default, a parameter that most users are unlikely to modify. As such, browser histories provide only time-limited CC-memory.

Any server can write any desired URL to a history cache in a user's browser by means of a redirect to the URL in question. This redirect can occur in a hidden window or otherwise without the perception – and therefore knowledge – of the user. Thus, a server can write arbitrarily and unobtrusively to a browser history cache. Other browser caches permit similar forms of imperceptible writing. And as explained, servers can also unobtrusively read the data they have written.

Cache cookies based on browser history have much looser read privileges than conventional cookies. Conventional *first-party* cookies may only be read when a browser visits the site that has created them. *Third-party* cookies may be read by a domain other than the one a browser is visiting. Major browsers today allow users to choose whether to accept or reject first-party and/or third-party cookies, and therefore permit users to opt out from the planting of cookies within their browsers. By contrast, unless a user suppresses the operation (and benefits) of caches in her browser, browser-history cache cookies are readable by querying servers irrespective of browser policy settings. Thus, read-access for cache cookies can be looser than the loosest browser policy permits for conventional cookies. On the one hand, this means that cache cookies permit such privacy infringements as clandestine cross-domain user tracking and information transfer. Thus, use of browser-history cache cookies demands special care.

Despite their flexibility, cache cookies based on browser history are a clumsy storage medium. They offer very low bandwidth: A server can read and write only one bit per operation. As suggested by experiments that we describe below, it is possible nonetheless for a server to manipulate as many as several hundred cache cookies without any perception by the user.

D. TIF-based cache cookies

Temporary Internet files (TIFs) are files containing information embedded in Web pages. Browsers cache these files to support faster display when a user revisits a Web page. Temporary Internet Files have no associated expiration; they persist indefinitely. Browsers do, however, cap the amount of disk space devoted to these files, and delete them so as to maintain this cap. Therefore, the persistence of TIFs varies from user to user.

In order to place a TIF X in a browser cache, a server can serve content that causes the downloading of X. A server can verify whether or not a given browser contains a given TIF X in its cache by displaying a page containing X. If X is not present in its cache, then the browser will request it; otherwise, the browser will not pull X, but instead retrieve its local copy. In order not to change the state of a cache cookie for whose presence it is testing, a server must in the former case withhold X. While this typically triggers a "401" error, manipulation of cache files can take place in a hidden window, unperceived by the user.

Cache cookies based on TIFs restrict read privileges, a useful privacy feature. When a browser requests a TIF X, therefore, it sends a request to the domain associated with X, not to the server displaying content containing X. In this regard, TIF-based cache cookies are like first-party cookies: Only the site in control of the domain for X can (easily) detect the presence of X in a browser cache. It is important to note that TIFs *are* subject to cross-domain timing attacks, which can undermine the first-party privacy property, but these attacks appear to be somewhat challenging to mount.

A notable limitation of TIFs is that they cannot be manipulated over SSL. As a security measure, HTTPS sessions do not cache information on disk.

Remarks:

- The method of initiating and selectively refusing client resource requests (provoking hidden "401" errors) also permits non-destructive reads of other cache-cookie types, like those for browser histories. Thus, even if the CSS-based method for reading browser-history caches should be unavail-

able, cache cookies remain realizable.

- We do not treat all possible forms of CC-memory in detail. For example, we do not discuss Etags here, nor the cached state associated with SSL sessions. Our techniques apply equally well to such forms of CC memory, though.
- Of course, browsers have individual quirks. Bookmarks in Safari, for example, are accessible using the same CSS-based techniques that permit reading of browser histories. Thus such bookmarks can serve as a foundation for CC-memory; a drawback is that a user must explicitly confirm bookmark additional, and thus become manually involved in writing of such CC-memory. Browsers in many mobile phones today do not accept conventional cookies (although this is changing). Often, however, they do have caches; thus, for certain mobile environments, C-memory or CC-memory may be particularly attractive as the only practical means to achieve functionality like that of conventional cookies.
- Beacuse of the volatility of C-memory and CC-memory due to object expirations, cache clearing by users, and so forth, it can be desirable to duplicate data across different memory structures.

E. Restricted and unrestricted cache-cookie memory

We now refer to CC-memory with domain-based read restrictions as *restricted* CC-memory. In other words, restricted CC-memory can only be read by one particular associated domain. We can regard (first-party-cookie) C-memory as a type of restricted CC-memory, and, barring timing attacks, also regard TIFs-based CC-memory as restricted. CC-memory that any server can read, like CC-memory based on browser-history caches, we call *unrestricted*. Unrestricted CC-memory is a virtual memory space that any server at all can read from. Although write-privilege restrictions can differ from read-privilege distinctions, it is sufficient for our purposes here to classify CC-memory based on read-privileges only.

III. Schemes for User Identification and Authentication

CC-memory can serve as an alternative to conventional cookies for storage of user identifiers, i.e., indices into user accounts on Web sites. As we have noted, however, unrestricted CC-memory does not possess the desirable read-privilege restrictions of conventional cookies. It is tempting to consider a simple scheme in which a server tries to restrict access to identifiers for its users by writing them to a secret (virtual) address in unrestricted CC-memory. This address, however, would have to be the same for every user, otherwise the server would not know where to find user identifiers *a priori*. The problem, then, is that any user could record which addresses in CC-memory the server accesses, and thereby determine the secret address. Any third party with knowledge of the secret address could then read user identifiers. Thus, we require a more sophisticated approach.

In this section, we propose some constructions for identification that support restrictions on read-accesses for user identifiers in unrestricted CC-memory. Rather than linking identifier access to particular domains, however, our techniques restrict access on the basis of server secrets, cryptographic keys in one case and pseudonyms in another. Server secrets can be delegated or held privately. Thus the approaches we propose are quite flexible, and can achieve access policies roughly like those for first-party or third-party cookies if desired.

We consider two ways to structure user identifiers in unrestricted CC-memory in a privacy enhancing way: A *tree-based* scheme and a *rolling pseudonym* scheme. The tree-based scheme restricts cache cookie access to a site that possesses an underlying secret key. The rolling-pseudonym approach decouples appearances of user identifiers to disrupt third-party tracking attempts.

The problem of privacy protection in cache-cookie identifiers bears an interesting similarity to that of privacy protection in radio-frequency identification (RFID) systems. In both cases, the identifying system is highly resource-constrained: RFID tags have little memory and often little computational power, while cache-cookie systems cannot invoke computational resources on the client side, but must rely on CC-memory accesses alone. Similarly, in both systems, the aim is to permit a server to identify a device (a tag or browser) without the device revealing a static identifier to potential privacy-compromising adversaries. Our tree-based scheme is reminiscent of an RFID-privacy system of Molnar and Wagner [13], [12], while our pseudonym system is somewhat like that of Juels [10].

We note that even though our identifier schemes are designed to limit read privileges for unrestricted CC-memory, they are also useful for restricted CC-memory. Recall that restricted CC-memory is readable only by a particular domain. Such CC-memory is vulnerable to pharming, that is, attacks in which an adversary spoofs a domain, and can therefore harvest domain-tagged data. Thus, our techniques can help defend restricted CC-memory against pharmers.[1]

[1] Our identifier-tree scheme is not resistant to a range of pharming attacks, but not to the strongest forms, such as online man-in-the-middle attacks involving pharming. Such at-

At the end of this section, we also briefly consider how *secret* cache cookies can aid in *authenticating* users that a server has already identified, and how they can help combat pharming attacks.

A. Identifier trees

Our first construction involves a tree, called an *identifier tree*, whose nodes correspond to secrets in CC-memory, which we call *secret* cache cookies. To administer a set of identifiers for its users, a server creates its own identifier tree and associates each user with a distinct leaf in the tree. The server plants in the browser of the user the set of secret cache cookies along the path from the root to this leaf. To identify a visiting user, the server interactively queries the user's browser to determine which path it contains; in other words, the server performs a depth-first search of the identifier tree. In identifying the user's unique leaf, the server identifies the user. This search is feasible only for the original server that generated the identifier tree (or for a delegate), because only the server knows the secret cache cookies associated with nodes in the tree. In other words, the key property of an identifier tree is privacy protection for user identifiers.

As an illustration, consider a binary tree T. Let d denote the depth of the tree. For a given node n within the tree, let $n \parallel$ '0' denote the left child, and $n \parallel$ '1', the right child; for the root, we take n to be a null string. Thus, for every distinct bitstring $B = b_0 b_1 \dots b_j$ of length j, there is a unique corresponding node n_B at depth j. The leaves of T are the set of nodes n_B for $B \in \{0,1\}^d$.

With each node n_B, we associate two secret values, y_B and u_B. The first value, y_B, is a k-bit secret key. The second value, u_B, is a secret (l-bit) address in CC-memory. To store node n_B in the CC-memory of a particular browser, the server stores the secret key y_B in address u_B. The sets of secret values $\{(y_B, u_B)\}_{B \in \{0,1\}^d}$ may be selected uniformly at random or, more better efficiency, generated pseudorandomly from a master secret key.

The server that has generated T for its population of users assigns each user to a unique, random leaf. Suppose that user i is associated with leaf $n_{B^{(i)}}$, where $B^{(i)} = b_1^{(i)} b_2^{(i)} \dots b_d^{(i)}$. When the user visits the server, the server determines the leaf – and thus identity – of the user as follows. The server first queries the user's browser to determine whether it contains n_0 or n_1 in its cache; in particular, the server queries address u_0 looking for secret key y_0, and then queries address u_1 looking for secret key y_1. The server then recurses. When it finds that node n_B is present in the browser, it searches to see whether $n_{B \parallel}$'0' or $n_{B \parallel}$'1' is present. Ultimately, the server finds the full path of nodes $n_{b_1^{(i)}}$, $n_{b_1^{(i)} b_2^{(i)}}$, $\dots$, $n_{b_1^{(i)} b_2^{(i)} \dots b_d^{(i)}}$, and thus the leaf corresponding to the identity of user i.

Of course, we can deploy identifier trees with any desired degree. For example, consider an identifier tree with degree $\delta = 2^z$, where d is a multiple of z, and the number of leaves is $L = 2^d$. The depth of such a tree, and consequently the number of stored secret cache cookies, is d/z; so too is the number of rounds of queries required for a depth-first search, assuming that each communication round contains the δ concurrent queries associated with the currently explored depth. Therefore, higher-degree trees induce lower storage requirements and round-complexity. On the other hand, higher-degree trees induce larger numbers of queries. Assuming δ (concurrent) queries per level of the tree, the total number of queries is $\delta d/z = 2^k d/z$.

In fact, it is possible to simulate trees of higher degree in searching a binary tree. Observe that we can "compress" a subtree of depth z anywhere within a binary tree by treating all of its 2^z deepest nodes as children of the root node of the subtree (effectively disregarding all of its internal nodes). Depth-first search over this subtree can be replaced by a single round of 2^z queries over the deepest child nodes. Likewise, we can compress any set of z consecutive levels within a binary tree by treating them as a single level of degree 2^z. Such compressed search achieves lower round-complexity than a binary depth-first search, at the cost of more queries. For example, a binary tree of depth $d = 12$ could be treated as a tree of degree $\delta = 16$ and $d = 3$. A server would perform a depth-first search with four rounds of communication; in each round, it would query the sixteen possible great-great grandchildren of the current node in the search. Compressed search within a binary tree can be quite flexible. A server can even adjust the degree of compression in its dynamically in accordance with observed network latency; high round-trip times favor high compression of tree levels, while low round-trip times may favor low compression.

As we explain in Appendix A, our identifier-tree scheme not only accommodates multiple users of a single browser, but can actually help in the management of shared browsers.

A.1 Simplified identifier tree

Because CC-memory has an "unwritten" state for bits, we can condense our identifier-tree so as to eliminate secret keys $\{y_B\}$. Instead, a node n_B is deemed

tacks effectively permit full eavesdropping, and thus full extraction of browser secrets.

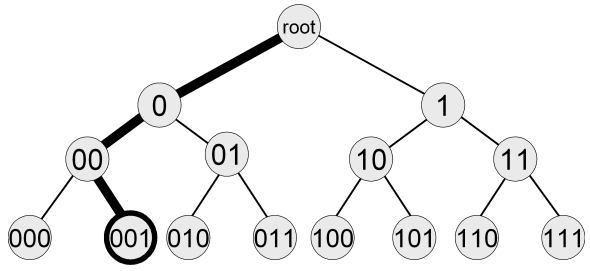

Fig. 1. A simplified identifier tree of depth $d = 3$. This tree supports eight identifiers, one for each leaf. The highlighted path corresponds to identifer '001'. To store this identifier in a user's browser, a server sets the bit value in addresses u_0, u_{00} and u_{001} of CC-memory to '1'. In a depth-first search of this browser's CC-memory, the server perceives set bits only along the highlighted path, leading it to the '001' leaf.

present if a '1' bit is written to u_B. This effectively means $k = 0$. Given sufficiently large d, we believe that this approach can still afford reasonably robust privacy. We show a schematic of a toy, simplified identifier tree in Figure 1.

A.2 Security

We do not give a formal or in-depth security analysis of our identifier-tree scheme. Briefly stated, our aim is to protect against an adversary that: (1) Controls a number of users and thus knows their identifiers and (2) Can lure users to a rogue server via a pharming attack. We assume, however, that beyond this: (A) The adversary does not possess knowledge of the set $\{(y_B, u_B)\}_{B\in\{0,1\}^d}$ of secrets composing the identifier tree; and (B) The adversary cannot mount an active (real-time) man-in-the-middle attack. Our security goals are twofold:

1. Privacy: The adversary should be unable to extract a unique identifier for a user from the identifier tree. Consequently, the adversary should be unable on the basis of the identifier tree to link independent sessions initiated by a given user.[2]
2. Authentication: The adversary should be unable to impersonate any user it does not control.

Several observations are important. The security parameter d determines the size of the identifier tree, and thus the size of the space of paths that an adversary must explore in exploiting identifiers. The security parameter l for CC-memory size determines the difficulty for an adversary of finding unknown nodes – and thus identifiers – when sniffing the browser of a user. It is easily possible to make l large enough to render brute-force sniffing of any node in the tree infeasible.

The security parameter k governs the difficulty for an adversary in simulating portions of the tree for which it does not know the corresponding secrets. It may in some cases be desirable to set k small for reasons of efficiency – even choosing $k = 0$ as we have proposed. When k is small, though, an adversary can extract path information by interacting directly with the server. With high probability, the adversary can simulate the presence of a node in its browser: When the server queries a node n_B, the adversary can successfully guess the corresponding secret y_B with probability 2^{-k}. Thus, when k is small, it is important to ensure that a server does not betray information about which paths correspond to real user identifiers. For this reason, a server should not terminate its depth-first search when it encounters valid secrets along a path that does not correspond to a real user identity. (As an alternative countermeasure, the server can support an army of "ghost" identifiers vastly exceeding its real user population.)

We briefly consider two security limitations of our scheme:

- *Man-in-the-middle attacks:* Irrespective of our security-parameter choices, an adversary capable of a man-in-the-middle attack can readily extract an identifier from a user. Thus, while our scheme does not afford privacy or authentication as defined above in the face of an attacker that interacts simultaneously with a valid server and the user.
- *Control of many users:* An adversary that controls many users can pool the secret keys along their respective paths in order to obtain partial information about the secret keys associated with T. We regard this as a limited problem, however. Given a tree of sufficient depth, even an adversary that controls many users can gain complete (or near-complete) knowledge of only the top layers of the tree. Such an adversary can fingerprint users according to the positions of their respective paths in the top layers of T, but cannot extract their unique identifers. This is already the case today: Servers can already fingerprint users to some extent based on their browser types, IP addresses, and so forth.

B. Rolling-pseudonym scheme

Another approach to protecting user identifiers is not to hide them, but rather to treat them as pseudonyms and change them on a regular basis.

The idea is very simply for the server to designate a series of k-bit CC-memory addresses $v_1, v_2, \ldots$, where

[2] The adversary can learn partial information about user identifiers and therefore can *correlate* appearances of a given user.

v_j is associated with time epoch j. The server additionally maintains for each user i and each time epoch j a k-bit pseudonym $\pi_j^{(i)}$.

Whenever the server has contact with the browser of a given user, it searches sequentially backward over p positions $v_j, v_{j-1}, \ldots, v_{j-p+1}$ until it locates a pseudonym π'_j in position $v_{j'}$ and identifies the user, or determines that no pseudonyms are present within its search window. On identifying the user, the server implants the current pseudonym $\pi_j^{(i)}$ in address v_j.

It is important that the server erase the contents of memory addresses $v_{j'}, v_{j'+1}, \ldots, v_{j-1}$. Otherwise, memory addresses associated with epochs in which the user has not communicated with the server will be blank. These blanks reveal information to a potential adversary about the visitation history of the user: Each blank indicates an epoch in which the user did not contact the server.

It is also important that the parameter p be sufficiently large to recognize infrequent users. Of course, a server can check whether a memory address has been previously accessed simply by reading its leading bit, and can therefore easily check hundred of epochs. If an epoch is three days, for example, then a server can easily scan years worth of memory addresses, and therefore the full lifetime of any ordinary browser – and well beyond the lifetime of certain cache-cookies, like those for browser history.

There are a number of ways, of course, of managing pseudonyms and mapping them to user identities. The simplest is for the server to use a master key x to generate pseudonyms as ciphertexts. If $e_x[m]$ represents a symmetric-key encryption of message m under key x, then the server might compute $\pi_j^{(i)} = e_x[i \parallel j]$. The server can decrypt a pseudonym to obtain the user identifier i.[3]

Of course, with this scheme, since a pseudonym is static over a given epoch, an adversary can link appearances of a user's browser within an epoch. Time periods, however, can be quite short – easily as short as a few days. Infrequent users are vulnerable to tracking, as their pseudonyms will not see rapid update. Therefore it is most appropriate to deploy rolling pseudonyms with CC-memory based on cache-cookies that expire, like those for browser histories.

Remarks:

- Of course, the secret key for these two identifier schemes can be delegated to a site other than the one that sets the cache cookies. This possibility can be useful for maintaining seamless user identification across associated sites. It is also subject to abuse if not carefully monitored.
- Cache cookies do not automatically support server-selectable expiration times, of course, but a server can cause an identifier to expire simply by erasing it. Servers can even write their cache-cookie use policies as data appended to identifiers.
- Our rolling-pseudonym scheme does not make much sense for implementation in C-memory, since a single cookie can act as a pseudonym and can be easily erased and replaced. In our identifier-tree scheme, on the other hand, C-memory permits efficient storage of long bit-strings at nodes, rather than single bits.

C. Denial-of-service attacks

In our rolling-pseudonym scheme, pseudonyms must reside in regions in CC-memory that an attacker can easily determine by observing the behavior of the server. Thus a malicious server can erase pseudonyms. This is a nuisance, of course, but it is no worse than the result of a flushed cache of conventional cookies. Indeed, a server can detect when an attack of this kind has occurred, since it will cause invalid erasures. To mitigate the effects of this type of attack, a server can keep secret the blocks of memory to be used for future pseudonyms.

An attacker can mount a denial-of-service attack against the identifier-tree scheme by adding spurious paths to a user cache. If the attacker inserts a path that does not terminate at a leaf, the server can detect this corruption of the browser data. If the attacker inserts a path that does terminate at a leaf, the result will be a spurious identifier. To prevent this attack, the server can include a cryptographic checksum on each identifier associated with the browser (embedded in a secret, user-specific address in CC-memory). The problem, of course, is that an attacker that has harvested a valid identifier can also embed its associated checksum. Instead, therefore, a checksum must be constructed in a browser-specific manner. For example, the checksum can be computed over *all* valid identifiers in the browser. Provided that the server always refreshes all identifiers simultaneously, this checksum will not go stale as a result of a subset of identifiers dropping from the cache. The checksum can serve to weed out spurious identifiers.[4]

[3] Authentication at some level is a necessary adjunct; while an identifier need not itself be authenticated, a user ultimately must be. To follow standard practice for well-constructed conventional cookies, pseudonyms might also include message authentication through, e.g., use of authenticated encryption.

[4] Note that an attacker that embeds spurious identifiers in caches is providing a forensic trail, since the attack must either have registered or compromised the associated accounts.

Another form of denial-of-service attack is for the attacker to write a large number of spurious paths to a cache in order to prolong the time required for the server to perform its depth-first search. It is unclear how to defend against this type of attack, although a server should be able to detect it, since the cache will either contain paths that do not terminate in leaves, or will contain implausibly many leaves.

To prevent long-lasting effects of such corruption in the identifier-tree scheme, a server can refresh trees on a periodic basis.

D. Secret cache cookies

We have described two schemes for user identification using cache cookies. In cases where users are identified by other means, like conventional cookies, cache cookies can still play a useful role in user login. Rather than supporting user identification, they can support user *authentication*, that is, confirmation of user identity. While cache cookies have privacy vulnerabilities that ordinary cookies do not, they also have privacy-enhancing strengths, like resistance to pharming, as we now explain.

Some vendors of anti-phishing platforms, such as PassMark Security [15], already employ conventional cookies and other sharable objects as authenticators to supplement passwords. Because conventional cookies (and similar sharable objects) are fully accessible by the domain that set them, they are vulnerable to *pharming*. A pharming attack creates an environment in which a browser directed to the Web server legitimately associated with a particular domain instead connects to a spoofed site. A pharmer can then harvest the cookies (first-party or third-party) associated with the attacked domain. Even the use of SSL offers only modest protection against such harvesting of cookies. It is generally difficult for a pharmer to obtain the private key corresponding to a legitimate SSL certificate for a domain under attack.[5] But a pharmer can use an incorrect certificate and simply rely on the tendency of users to disregard browser warnings about certificate mismatches. Pharming attacks can thus undermine the use of cookies as supplementary authenticators.

Secret cache cookies offer resistance to pharming. A secret cache cookie, very simply, is a secret, k-bit key y_i specific to user i that is stored in a secret, user-specific address u_i in CC-memory (or C-memory). We have already proposed the use of secret cache cookies in our identifier-tree scheme.

[5] Pharmers have been known to obtain fraudulent certificates, however [17]. And in principle a pharmer dupe a user to installing an invalid certificate in her browser.

For the purposes of authentication, we can employ secret cache cookies more simply. Once the user identifies herself and perhaps authenticates with other means, e.g., a password or hardware token, a server can check for the presence of the user-specific secret cache cookie y_i as a secondary authenticator.[6]

D.1 Security

For both restricted and unrestricted CC-memory, as well as C-memory, secret cache cookies are resistant to basic pharming, i.e., an attacker that lures a user to a rogue server, steals credentials, and then interacts with a server. This is because secret cache cookies do not rely on domain names for server authentication. In order to access the key y_i, a server must know the secret address u_i associated with a user.

A more aggressive pharming attack can, however, compromise a secret cache cookie. For example, a pharmer can lure a user, steal her password, log into a server to learn u_i, lure the user a second time, and steal y_i. We cannot wholly defend against this type of multi-phase attack, but using the same methods of interactive querying as in our identifier-tree scheme, we can raise the number of required attack phases. We associate with user i not one secret cache cookie, but a series of them, $\{(u_i^{(1)}, y_i^{(1)})\}_{i=1}^{d}$. A server searches first for $y_i^{(1)}$ in address $u_i^{(1)}$. It then searches for the other $d-1$ secret cache cookies sequentially, rejecting the authentication attempt immediately when it is unable to locate one. The scheme can be strengthened by having the server refresh the set of secret cache cookies whenever a user authenticates successfully.

In order to defeat authentication via secret cache cookies, a pharmer must interact with a server and client in turn at least d times. For large enough d, this virtually requires a real-time man-in-the-middle attack. Of course, a pharmer can alternatively compromise a victim's machine (at which point the victim is subject to a broad range of attacks), or compromise the link between a server and client.

Because CC-memory only in general allows for single-bit reads and writes, secret cache cookies are inefficient authenticators for large l or d. They are much more efficient in C-memory, i.e., with ordinary cookies, since the costs associated with reads and writes for C-memory are practically invariant in l. Another considerable benefit of C-memory is that it can be accessed over SSL. (Recall, as mentioned above, that certain forms of CC-memory, like TIF-based CC-memory, cannot.) In fact, "secure" cookies are ac-

[6] Even though the address u_i is itself secret, it is important to use a secret key y_i, rather than to plant a single indicator bit at u_i: If a server merely checks for the existence of a bit at u_i, a saavy attacker can simulate the presence of such a bit.

cessible exclusively over an SSL connection. Again, CC-memory is beneficial largely when C-memory is unavailable, as when the user blocks first-party cookies, or when C-memory is restricted, as when the user blocks third-party cookies.

As with conventional cookie-based authenticators, and mechanisms like IP-tracing, there is a drawback to secret cache cookies: A user who changes browsers loses the benefit of the secondary authenticator.

IV. Audit Mechanisms

As we have explained, cache cookies can serve as a mechanism for abuse of user privacy, particularly in unrestricted CC-memory. In this section, we enumerate a few possible strategies for detecting and preventing misuse of cache cookies.

As noted above, there are many possible abuses of cache cookies stored on browsers in the natural course of their operation. For example, a server that mines data from a browser history cache can profile a user according to which political sites she has visited. Here we focus instead on the use and abuse of the general read/write memory structures for cache cookies that we have described here, and focus particularly on user identifiers. A server can abuse these structures in two ways: (1) By tracking users with cache cookies when users do not wish for such tracking to occur and (2) By making cache-cookie information available to third parties against the wishes or simply in the absence of knowledge of users.

Briefly then, we have proposed ways for servers to deploy cache cookies with privacy support. The question now is how can users be sure that servers are in fact deploying cache cookies appropriately?

It should be noted that in some measure, cache cookie reads and writes are transparent. This is to say that with the appropriate software, e.g., a browser plug-in, a client can detect which cache cookies a server has accessed. By detecting which portions of its caches are "touched," for example, a client can determine the pattern of browser probing. (Of course, these remarks do not apply to timing-based cache cookies. Thankfully such cookies are relatively difficult to exploit.) Jackson, Bortz, Boneh, and Mitchell [6], for instance, propose ways in which modified browsers can provide users with the ability to unify their privacy polices to extend beyond conventional cookies to cache cookies. Of course, similar tools can serve instead to detect and track server-side behavior. Auditing, rather than universal browser changes, can also help enforce user privacy. For example, if identifier-tree leaves are tagged with the domain of the server that sets them, client-side software can detect third-party sniffing of identifiers.

On the server side, a number of standard tools can support privacy auditing, among them:

1. **Software inspection:** An organization with a published privacy policy can rely on audits of its servers to ensure the use of compliant software.
2. **Trusted computing modules (TPMs):** A server containing a TPM can provide assurance to an external entity that it has a particular software configuration. This configuration can include software that protects the privacy of user databases as in, e.g., [16].
3. **Platform for Privacy Preferences (P3P):** P3P [1] is a standard by which a browser and server can negotiate the exchange of information based on their respective privacy policies. Extensions to P3P can naturally underpin server-side determination of how and whether or not to deploy cache cookies. P3P provides no external means of enforcement, however, i.e., it presumes compliance by a participating server.
4. **Mixnets and proxies:** Privacy-protections on the server side can be distributed among multiple servers or entities. For example, an authentication service could identify users by means of third-party cookies simulated via cache cookies. This service could translate user identifiers into pseudonyms on behalf of relying sites.

On the client side, various forms of direct auditing of server behavior are possible, as we have explained. Here, however, we propose a special mechanism specific to identifier trees called *proprietary identifier-trees*.

A. Proprietary identifier-trees

As we have noted, a domain can share the secret key underlying an identifier tree with other domains. If the tree is present in unrestricted CC-memory, such sharing permits cross-domain tracking of users. We now consider a way to restrict this type of sharing.

Limiting the ability of servers to read unrestricted CC-memory is fundamentally a digital-rights management (DRM) problem. Here, as in many consumer-privacy settings, it is the user that wishes to control the flow of data, rather than a corporate interest. In particular, the user wishes to constrain the right to access information in her browser to the relying domain only.

Given this view, we appeal to the DRM-oriented "digital signet" concept of Dwork, Lotspiech, and Naor [4]. Briefly, their idea is to protect against one entity, Alice, sharing proprietary information with a second entity, Bob, by making a key to the information transmissible in one of two ways: (1) As a short secret that includes private data that Alice would be

reluctant to share with Bob, e.g., her credit-card number, or (2) As a secret so long that it is cumbersome to transmit.

Our idea is to link the underlying secret for the tree for our identifier-tree scheme in section A to a secret belonging to the server, namely the private key for its SSL certificate. We call the resulting structure a *proprietary identifer-tree*. To share a proprietary identifier-tree with another party, a server must either transmit a large portion of the tree (at a minimum, the path for every user), or compromise its SSL certificate.

For meaningful use of proprietary identifier trees, we require an additional property absent in the Dwork et al. scheme (as their scheme is purely secret-key based). We would like a third party to be able to *verify* the linkage between the identifier tree and the SSL certificate. In this sense, we draw on the ideas of Camenisch and Lysyanskaya [2] and Jakobsson, Juels, and Nguyen [8]. They show how digital credentials may be created such that the secret key for one credential leaks the private key associated with another, and demonstrate how one party can *prove* this linkage of credentials.

For the particular case of identifier trees, we can employ a *verifiable unpredictable function* (VUF) [11] as proposed by Micali, Rabin, and Vadhan. This is a function f with a corresponding secret s such that $f_s(m)$ is efficiently computable for any message m; additionally, knowledge of s permits construction of an efficiently checkable proof of the correctness of $f_s(m)$. Knowledge of the value of f_s at any number of points does not permit prediction of f_s on a new point.

Micali et al. demonstrate an RSA-based construction. In this construction, the public information consists of an RSA modulus N, a random value $r \in Z_N^*$, along with a polynomial Q (whose form we omit here), and a random string *coins*; the value a is an associated parameter on the bit-length of inputs. For any a-bit input value m, the pair $(Q, coins)$ defines a deterministic mapping to a corresponding prime p_m. The secret key s consists of the prime factors p, q of N. The VUF is simply computed as $f_s(m) = r^{1/p_m} \bmod N$. It is easy to see that anyone can verify the correctness of $f_s(m)$ based on the public information $(N, r, Q, coins)$ alone.

Application to our identifier-tree scheme is straightforward. Let us suppose, then, that N is the RSA modulus associated with the SSL certificate for a server administering an identifier tree T; let s be the associated prime factors. In addition to or as an extension to its SSL certificate, the server also publishes an associated triple $(r, Q, coins)$ for which $a = 2d + 1$. Let $g : \{0,1\}^z \rightarrow \{0,1\}^{2d}$, for $0 \leq z \leq d$ be a padding function that maps every bitstring length at most d to a unique bitstring of length $2d$. For a given node n_B, then, the server computes $u_B = f_s(\text{`0'} \parallel g(n_B))$ and $y_B = f_s(\text{`1'} \parallel g(n_B))$. (The '0' and '1' prefixes here ensure distinct inputs for computation of u_B and y_B.) Given this use of a VUF, it is easy to see that a user can verify that the path corresponding to her identifier is consistent with an identifier-tree linked to the SSL certificate of a given server. If many users – or auditors – perform such verification, then it is possible to achieve good assurance of server compliance with the scheme.

Of course, it is feasible for a server to circumvent disclosure of its private SSL key by transmitting portions of its identifier tree. For example, with a tree of depth 80, 1024-bit digital signatures, and a base of 1,000,000 users, the size of the associated tree data would be slightly more than 10GB. Thus, even without sharing its private key, a server can plausibly share its set of user identifiers. The more important aspect of our scheme is that, without sharing its private key, a server must share *updates* to the identifier tree when new users join. The resulting requirements for data coordination are a substantial technical encumbrance and disincentive in our view. Additionally, the ongoing relationship required for such updates would expose more evidence of collusion to potential auditors.

Remark: In a VUF, the function f is deterministic, and thus the value $f_s(m)$ is uniquely determined. This property is not in fact essential for a proprietary identifier-tree. Rather, we require the weaker property that the values y_B and u_B be *infeasible to compute* without knowledge of the private key associated with the server's SSL key. A VUF permits simple, rigorous proof of this property, but we believe that the same property can be achieved using an ordinary digital signature with existential unforgeability.

A simpler scheme based on digital signatures is as follows. A random value r is assigned to the root node. Now, for any node n_B, the values y_B and u_B are computed respectively as digital signatures on the messages "'0' $\parallel$ $g(n_B)$" and "'1' $\parallel$ $g(n_B)$". These digital signatures take the form of RSA signatures, e.g., PKCS #1 signatures, with the SSL certificate defining the public key. Provided that the signature scheme carries the right security properties, an adversary cannot guess the value of unrevealed secrets in the identifier tree. This is true, for instance, for signature schemes that are existentially unforgeable under chosen-message attacks. Given this property, the ability to construct any unrevealed portion of the identifier tree implies knowledge of the private key for the SSL certificate. A simple signature-based scheme, however, lacks the crisp security properties

of one based on VUFs. For example, the signer, that is, the creator of the identifier tree, can embed side-information in its signatures, perhaps undermining its security guarantees. Thus, careful construction and analysis – beyond the scope of this paper – are warranted for such a scheme.

V. Implementation

We now describe an implementation of CC-memory based on TIFs. Our server is an Apache 1.3.33 using FastCGI, Perl and Gentoo Linux (2.4.28 kernel), on a 1 GHz Pentium III with 256MB memory. Our client uses Mozilla 1.5.0.1 and Windows XP, on a machine with identical hardware as the server. Thus, the server is clearly under-powered for its task; on the other hand, we performed experiments on a 100 Mbps private local area network with minimal network traffic and congestion.

We execute a *write* to the browser cache by causing the client to make a series of HTTP requests to cacheable content. In our implementation we chose to cache GIF image files referenced from a dynamically generated document. These images contain solely the HTTP header and no actual content, resulting in very quick loads. The HTTP/1.1 server response header for the first load contains *Last-Modified*, *ETag*, *Cache-Control*, and *Expires* fields and values. The Cache-Control and Expires fields are set to instruct the Web client to cache the content many years into the future. An ETag (short for "entity tag") is a field that enables a server to distinguish among different instances of a single resource, e.g., different versions or copies of a cached browser image.

We execute a *read* via subsequent client retrievals of the cached objects. These result in the client sending *Last-Modified* and *ETag* values to the server in HTTP requests in the form of *If-Modified-Since* and *If-None-Match* fields respectively. If these values match those in the initial *write*, then a cache hit is observed. In this case, the server returns an HTTP 304 (Not Modified) response so as not to "clobber" the cached value. Otherwise, it returns a 404 (Not Found) HTTP response. (This process of a client sending data to a server to be validated is called a conditional GET request.)

Our uses proposed above for cache cookies are likely to involve considerably more frequent reads, i.e., authentications, than writes, i.e., initializations. Thus in our experiment we measured the full, round-trip time for the server to read a batch of n TIFs, i.e., to read n TIFs in a single communication round. We refer to Figure 2 for our results; we have plotted one hundred data points for each value of n within the range of 1 to 80.

As an example, consider a translation of these timing results into a performance estimate for an identifier tree, such as a binary tree of depth $d = 60$. For $n = 2$, the average read time was 0.04175 seconds. This corresponds to the expected time for the server to test the pair of descendants of a given node. Thus traversal of the full tree would require an average of approximately 2.5 seconds.

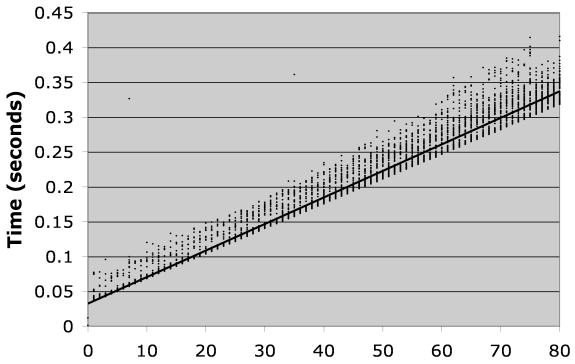

Fig. 2. Round-trip time for a server to read a batch of n TIF cache-cookies

We can greatly extend the amount of information in a TIF in CC-memory by co-opting two fields. There is the *Last-Modified* field, which contains 32 bits. The *ETag*, though, is particularly useful for our purposes; in Mozilla 1.5.0.1, for example, an ETag can contain up to 81864 bits. (The line buffer for the ETag is 10k bytes, some devoted to header information.) Thus for secret cache cookies, a single TIF can furnish essentially as much secret data as needed – well beyond the 128 bits typical for a cryptographic secret key.

VI. Conclusion

Cache cookies will be a long-lived browser feature. We have demonstrated that with careful deployment, cache cookies can support user identification schemes with good privacy protection. They can also support stronger forms of user authentication and thus indeed enhance user privacy by protecting against phishing and pharming attacks. Users are increasingly blocking and erasing cookies because of concerns about privacy infringement. As we have shown, cache cookies are an alternative that can replace some of the lost functionality of conventional cookies. Additionally, some of the techniques we have introduced here for cache cookies can help strengthen conventional cookies.

Of course, cache cookies are indeed subject to abuse. We have briefly proposed some tools to help ensure appropriate use. Among these is audit – a solution that will be most valuable with supporting communal policy guidelines and perhaps even legislation. Proper use and regulation of cache cookies will

present an important ongoing problem for the security and privacy communities. Thankfully, given their scant use today, cache cookies offer a clean slate on which to develop new polices. For this purpose, the history of conventional cookies affords ample experience and hindsight.

Acknowledgments

The authors thank Collin Jackson for his suggested improvements to this work, and to Dan Boneh, Andrew Bortz, and John Mitchell for their helpful comments.

References

[1] Platform for privacy preferences (P3P) project. World Wide Web Consortium (W3C). Referenced 2005 at http://www.w3.org/p3p.

[2] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In B. Pfitzmann, editor, *Eurocrypt 01*, pages 93–118. Springer-Verlag, 2001. LNCS no. 2045.

[3] A. Clover. Timing attacks on Web privacy (paper and specific issue), 20 February 2002. Referenced 2005 at http://www.securiteam.com.

[4] C. Dwork, J.B. Lotspiech, and M. Naor. Digital signets: Self-enforcing protection of digital information (preliminary version). In *ACM Symposium on the Theory of Computing (STOC)*, pages 489–498, 1996.

[5] E. W. Felten and M. A. Schneider. Timing attacks on Web privacy. In *ACM Conference on Computer and Communications Security*, pages 25–32. ACM Press, 2000. Referenced 2005 at http://www.cs.princeton.edu/sip/pub/webtiming.pdf.

[6] C. Jackson, A. Bortz, D. Boneh, and J. Mitchell. Web privacy attacks on a unified same-origin browser, 2005. In submission.

[7] M. Jakobsson, T. Jagatic, and S. Stamm. Phishing for clues: Inferring context using cascading style sheets and browser history, 2005. Referenced 2005 at http://www.browser-recon.info.

[8] M. Jakobsson, A. Juels, and P. Nguyen. Proprietary certificates. In B. Preneel, editor, *RSA Conference Cryptographers Track (CT-RSA)*, pages 164–181. Springer-Verlag, 2002. LNCS no. 2271.

[9] M. Jakobsson and S. Stamm. Invasive browser sniffing and countermeasures. Manuscript.

[10] A. Juels. Minimalist cryptography for low-cost RFID tags. In C. Blundo and S. Cimato, editors, *Security in Communication Networks – SCN 2004*, pages 149–164. Springer-Verlag, 2004. LNCS no. 3352.

[11] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

[12] D. Molnar, A. Soppera, and D. Wagner. A scalable, delegatable pseudonym protocol enabling ownership transfer of RFID tags. In B. Preneel and S. Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.

[13] D. Molnar and D. Wagner. Privacy and security in library RFID : Issues, practices, and architectures. In B. Pfitzmann and P. McDaniel, editors, *ACM Conference on Communications and Computer Security*, pages 210 – 219. ACM Press, 2004.

[14] R. Rivest and A. Shamir. Method and apparatus for reusing non-erasable memory media. United States Patent 4,691,299. Issued 1 Sept. 1987.

[15] PassMark Security. Company Web site, 2005. Referenced 2005 at http://www.passmarksecurity.com/.

[16] S.W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Sys. J.*, 40(3):685–695, 2001.

[17] J. Vijayan. Microsoft warms of fraudulent digital certificates. *Computerworld*, 22 March 2001. Referenced 2006 at http://www.computerworld.com.

Appendix

A. Sandwich cookies

Our cache cookie-based identifier schemes have a useful characteristic. They permit the co-existence of multiple identifiers; we use the term *sandwich cookies* to refer to cache-cookie identifiers for a single Web site that are resident in the same browser. In our tree scheme, provided that the server explores edges for both children at every node, it can detect the presence of multiple paths in a cache, and thus the presence of multiple identifiers. Likewise, in our rolling pseudonym scheme, a server can reserve different portions of memory for different pseudonyms, and search these addresses to check for the existence of multiple pseudonyms.

The ability of a server to detect the presence of multiple identifiers can be very convenient. Multiple users often access their Web accounts on a single machine through the same browser. Such sharing frequently causes one user to "bump" the cookie of another user. Alice may recently have visited site XYZ, causing a cookie to be planted such that when she again visits XYZ on the same browser, she need not re-authenticate; the browser in effect "recognizes" Alice as the current user. In most systems today, however, if Bob logs into XYZ from the same browser, the cookie associated with Alice is entirely effaced, and the browser instead recognizes Bob as the current user. Sandwich cookies, in contrast, permit a server easily to manage multiple accounts in the same browser. For example, a server that detects identifiers for both Alice and Bob, can cause a browser display a script offering easy selection between the two accounts. (How and when users are required to reauthenticate is then a matter of policy.)

Of course, conventional cookies can be transformed into sandwich cookies; a cookie need merely store multiple identifiers instead of a single one, or expand a single identifier on the server end into multiple identifiers. With cache cookies, however, it is possible to manipulate identifiers independently. In other words, sandwich cookies based on cache cookies do not require any linking of separate user accounts.

Sandwich cookies can enable some simple enhancements to the user experience in Web browsing. Many Web sites at present maintain persistent user sessions by means of cookies: A user remains logged into the

site while her browser contains an appropriate cookie. When a session is in force for one user on a particular Web site, another user, by logging into the same site, "bumps" the first. This second user's cookie takes the place of that of the first user, whom the Web site no longer recognizes.

Sandwich cookies permit an enhancement to the experience of users who share a browser to access the same Web site. Rather than allowing one user to bump another, a Web site can maintain a set of active user logins, and allow a user to choose her account identifier from a menu. (Various policies are possible for determining when an account identifier appears in the menu and when users must re-authenticate.)