A Privacy-Preserving Defense Mechanism Against Request Forgery Attacks

Ben S. Y. Fung and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
{syfung,pclee}@cse.cuhk.edu.hk

***Abstract*—One top vulnerability in today's web applications is request forgery, in which an attacker triggers an unintentional request from a client browser to a target website and exploits the client's privileges on the website. To defend against a general class of cross-site and same-site request forgery attacks, we propose *DeRef*, a practical defense mechanism that allows a website to apply fine-grained access control on the scopes within which the client's authentication credentials can be embedded in requests. One key feature of DeRef is to enable *privacy-preserving checking*, such that the website does not know where the browser initiates requests, while the browser cannot infer the scopes being configured by the website. DeRef achieves this by using two-phase checking, which leverages hashing and blind signature to make a trade-off between performance and privacy protection. We implement a proof-of-concept prototype of DeRef on FireFox and WordPress 2.0. We also evaluate our DeRef prototype and justify its performance overhead in various deployment scenarios.**

I. Introduction

Session state management [17] is a critical component in modern web applications. It augments stateless HTTP and embeds *authentication credentials* of web clients into HTTP messages (e.g., in the form of cookies or the HTTP authentication header), so that a website can determine the privileges of different clients. However, HTTP session state management is subject to various security vulnerabilities [23]. One such vulnerability is Cross-Site Request Forgery (CSRF), in which an attacker's website triggers a client's browser to send an HTTP request to a target website. If the HTTP request carries the client's credentials, then the attacker can perform actions on the website using the client's privileges, without the client being notified. There are different variants of CSRF, such as Clickjacking [12] and Login CSRF [4].

There have been extensive studies on how to defend against CSRF (e.g., see [4], [15], [16], [18]). One approach is `Referer` checking, in which the target website can determine the complete URL from which the request is initiated. However, the URL information can reveal the access history of the client [4]. A more robust approach is *token validation* (e.g., see [26]), in which the target website embeds secret tokens in HTTP responses, so that the browser can include those tokens in HTTP requests to authorize the request initiations. These tokens are inaccessible by third-party websites due to the same origin policy (SOP) [24]. However, such protection fails if both target and malicious websites have the same origin but are owned by different parties (e.g., http://www.foo.com/~alice/ and http://www.foo.com/~trudy/), as the malicious party can steal the tokens from another same-origin website and trigger forged requests. We call this attack the same-site request forgery (SSRF) attack.

To effectively defend against both CSRF and SSRF attacks, we consider an approach based on *fine-grained access control of scopes*. A *scope* defines a combination of the protocol, domain, and path (see Section III-B). A website can configure, in a policy file, the scopes that are legitimate to initiate or receive sensitive requests that contain authentication credentials. The browser can download the policy file from the website to check the validity of each of its initiated requests, and exclude sensitive credentials from any requests that are considered to be forged. This fine-grained access control is also considered in previous studies (e.g., [9], [22]).

However, one shortcoming of existing fine-grained access control approaches is that the policy file carries sensitive scope information in plain format that is accessible by every browser to check against its initiated requests. Users can find out sensitive information from the policy file, such as how a website designs its access control policies and its trust relationships with other websites. The need of protecting the sensitive access-control information has been justified in prior studies (e.g., [25], [10], [7]). Thus, our goal is to allow the browser and the website to exchange sensitive scope information while they may not need to fully trust each other, in the context of defending against request forgery attacks.

In this paper, we propose DeRef, a practical defense mechanism against cross-site and same-site request forgery attacks using privacy-preserving fine-grained access control. By privacy-preserving, we mean to not only protect a browser from revealing the URLs from which it initiates requests, but also protect a website from revealing how it configures the legitimate scopes, except for those that have been visited by the browser. The main idea of DeRef is to employ *two-phase checking*. First, the website configures (i) the scopes that are permitted to initiate sensitive requests and (ii) the scopes on the website that are protected by DeRef. Then the website sends the hash values of the scopes to the browser, where the hash values are incomplete and reveal only *partial* scope information. In the first phase, the browser checks to see if its initiated requests potentially fall within the configured scopes, and eliminate those that are known to be not configured by the website. In the second phase, the browser sends the *blinded*

The source code of DeRef is available for academic use and can be downloaded at: **http://ansrlab.cse.cuhk.edu.hk/software/deref**.

scopes of its initiated requests to confirm if these scopes actually match the configured scopes. In a nutshell, DeRef uses two-phase checking to make a trade-off between performance and privacy protection in real deployment.

To show that DeRef is deployable in practice, we implement a proof-of-concept prototype of DeRef on FireFox [19] (as a browser plugin) and WordPress 2.0 [30]. We also address how the prototype is backward compatible with the original client/server operations without DeRef. We evaluate our DeRef prototype, and show that its response time overhead can be reduced to within 19% by caching the already checked scopes.

The rest of the paper proceeds as follows. Section II reviews the background on request forgery attacks and their defense mechanisms. Section III presents the design and implementation details of DeRef. Section IV evaluates the performance and scalability of DeRef. Finally, Section V concludes.

II. Background and Related Work

A. Request Forgery Attacks

A *request forgery attack* is to trigger a forged HTTP request from a victim client browser to a target website without the knowledge of the client. A forged request may carry the client's authentication credentials that an attacker can exploit to perform malicious actions on the website using the client's privileges. In the following, we describe different variants of request forgery attacks.

Cross Site Request Forgery (CSRF) [4], [15], [16], [18]. In CSRF, an attacker uses an external website to trigger an HTTP request from a client to a target website. Suppose that a client currently has an active session with a target website A and then visits a malicious website B. The attacker can put a malicious URL on website B that triggers the client's browser to send an HTTP request to website A using the currently active session. Then the credentials associated with website A will be attached to the triggered HTTP request, and website A will process the request using the client's privileges. There are variants of the CSRF attack, including *Clickjacking* [12] and *Login CSRF* [4].

Same Site Request Forgery (SSRF) [21], [22]. Different websites may have the same *origin* [24] (i.e., same protocol, hostname, and port number), while these websites correspond to different owners. For example, Alice (target) and Trudy (attacker) may individually own websites on the URLs http://www.foo.com/~alice/ and http://www.foo.com/~trudy/. Suppose that a client currently has an active session with Alice's website and then visits Trudy's website. In this case, Trudy's malicious page can read the content in Alice's website, which is permitted under the *same-origin policy* [24]. This is referred to as an SSRF attack. Note that the attack still works even though Alice uses token validation [26], which can effectively defend against CSRF attacks.

Note that there are many real cases in which both attacker and victim host their websites under the same domain, such as the personal websites hosted under many legacy university domains. The personal websites are partitioned by path (e.g., http://www.foo.edu/~alice) rather than by subdomain (e.g., http://alice.foo.edu/). In this case, SSRF attacks are possible.

B. Current Defense Approaches

There have been various defense approaches against request forgery attacks.

Header checking. A simple approach is to let the website check the `Referer` header and determine where the request is initiated. However, this approach has privacy concerns, as the `Referer` header reveals the last visited URL of a client from which the request is initiated. To protect a client's privacy, the *origin header* approach [4] introduces the `Origin` header, which is similar to the `Referer` header except that it only contains the origin information with the path details removed.

Token validation (e.g., [26]). Token validation is widely deployed to defend against CSRF. The website generates a secret token in a client session, and validates the token when the client initiates requests to perform privileged actions. The token is protected from other websites by the same-origin policy. However, token validation is difficult to implement due to the possibility of leaking the token value [4].

Client-side defense. Unlike the above approaches, some studies consider client-side approaches that do not require server-side participation, thereby making deployment easier. *RequestRodeo* [15] is a client-side proxy that strips credentials from a request whose URL has a different origin from the originating webpage. Since it is proxy-based, it cannot examine HTTPS traffic. *BEAP* [18] is implemented as a browser plugin so that it can examine HTTP and HTTPS traffic. It focuses on inferring the intentions of clients in generating cross-site requests.

Fine-grained access control. The aforementioned approaches mainly focus on CSRF attacks, and do not address how to defend against SSRF attacks. Fine-grained defense approaches allow website owners configure the access scopes from which requests can be initiated. *SOMA* [22] requires a website to set up the policy files that specify the external websites with which the website can communicate. The browser can use the policy files to enforce protection. *Csfire* [9] is a browser plugin that parses a fine-grained policy file that specifies which third-party sites can initiate cross-site requests. Other studies, such as MashupOS [3], Subspace [13], and OMash [8], consider more fine-grained access control for cross-site communications in mashup applications. W3C [28] also drafts a specification that states how websites can configure the objects that can be shared across origins. Note that while the above approaches focus on protecting against cross-site attacks, we can extend them to defend against SSRF attacks by configuring the access scopes within the same site.

C. Lessons Learned

In this paper, we consider how to use fine-grained access control to defend against both CSRF and SSRF attacks. Similar to SOMA [22], we allow a website to configure a policy file that describes how requests can be initiated and received between a browser and the website. Then the browser uses the policy file to enforce access control. In our prior work

[11], we provide a preliminary implementation for such a fine-grained access control mechanism by storing the policy-based information in the Bloom filter.

Although fine-grained access control is sound, one major concern is that clients can access the policy file and easily determine how a website designs its access control policy and its trust relationships with other websites. This information is sensitive and should be protected as well. Previous studies [25], [10], [7] also address the need of protecting access control information, but they are designed for different types of applications, such as automated trust establishment [25], pervasive systems [10], and firewalls [7]. Besides access control, there are extensive studies on privacy-preserving mechanisms in other aspects, such as in data mining (e.g., see [1]) and two-party communication (e.g., see [5], [20]). Our goal here is to extend our prior design in [11] and deploy a privacy-preserving approach that can protect the policy information from outsiders, while still effectively defending against both CSRF and SSRF attacks.

III. DeRef Design

DeRef is designed as a privacy-preserving, fine-grained defense mechanism against request forgery attacks. In summary, DeRef aims for the following design goals.

- *Detecting forged requests*. DeRef seeks to defend against general request forgery attacks, including both cross-site and same-site.
- *Fine-grained access control*. DeRef enables a website owner to configure the scopes that are under protection, so as to eliminate stringent checking on all incoming requests.
- *Privacy-preserving checking*. DeRef can identify forged requests without requiring both the browser and the website to disclose private information to the other side.
- *Feasible deployment*. DeRef can be feasibly deployed in today's browsers and websites.

A. Threat Model

DeRef seeks to defend against CSRF and SSRF attacks described in Section II. Specifically, DeRef enables a browser to identify "forged" requests and strip any *authentication credentials* from these requests or their corresponding responses before relaying them.

In this paper, we focus on two types of authentication credentials: (i) cookies and (ii) HTTP authentication (i.e., the `Authorization` header). Although authentication credentials can also appear in the query strings of GET requests or in the data in POST requests, their definitions and formats are application-specific and it is difficult to distinguish the credentials from application data. The identification of application-specific credentials will be posed as future work.

To determine if a request is forged, we need to first determine how the request is triggered and where the request is destined for. We define the *initiating URLs* as the set of URLs that can directly or indirectly initiate the request. They include (i) the `Referer` URL and (ii) the URLs of the current active iframe's ancestors in the iframe hierarchy [4]. Also, we define the *target URL* as the destination URL of the request. Figure 1 depicts an example of how the initiating URLs and target URLs are defined. We allow a website owner to configure a set of target URLs on the website that are to be protected, as well as a set of initiating URLs that are "approved" to initiate requests that carry authentication credentials to the protected target URLs (see Section III-B for details). If a request is sent to a protected target URL from any non-approved initiating URL, then we say that the request is *forged*. For example, in Figure 1, if the URL of the target link is protected, then all three initiating URLs (i.e., the URLs of iframe1, iframe2, and the browser tab) must be approved by the website in order for a request to be able to carry authentication credentials; otherwise, the credentials will be removed from the request. Here, we assume that the approved initiating URLs are benign and no request forgery attacks are launched from there.

Fig. 1. Suppose that the target link is clicked. The `Referer` header will have the URL of iframe1. The *target URL* will be the URL of the target link, and there are three *initiating URLs*, including the URLs of iframe1, iframe2, and the browser tab.

B. Fine-Grained Access Control

DeRef is built on two access control lists (ACLs), namely *T-ACL* and *I-ACL*, to enable fine-grained defense against request forgery attacks. T-ACL stores the target URLs on the website those are to be protected. The stored URLs generally correspond to the sensitive web objects that need to respond to the authentication credentials inside the requests, and hence they need protection against forged requests. Other non-sensitive web objects that are not stored in T-ACL will remain unaffected. Thus, a main purpose of T-ACL is to eliminate stringent checking on the non-sensitive web objects.

I-ACL stores the initiating URLs that are trusted to initiate requests to the target URLs configured in T-ACL. A main purpose of I-ACL is to configure the URLs that have different origins while being trusted (i.e., the same origin policy cannot be directly applicable). One real-life example would be the websites www.asiamiles.com and www.cathaypacific.com. While they have different origins, they are mutually trusted as they deploy the Single Sign-On (SSO) mechanism [27]. Thus, I-ACL is used to customize the trusted initiating URLs that may have the same or different origins. If *any* initiating URL of a request is not configured in I-ACL, while the request is destined for the target URL that is configured in T-ACL, then the request is considered to be forged.

Scope. Before deploying DeRef, the website on the server side first configures the ACLs with a set of *scopes*. A scope is defined based on the *same origin policy for cookies* [32], and it specifies the range of URLs using scheme://domain/path, where (i) the `scheme` corresponds to the protocol of the

request (e.g., `http` or `https`), (ii) the `domain` includes the domain itself, its sub-domains, and its underlying hosts, and (iii) the `path` includes the path itself and its path suffixes. To show how a scope is used, let us configure a scope http://.foo.com/dir/. Then examples of URLs that match our configured scope are http://www.foo.com/dir/ and http://www1.foo.com/dir/sub/. On the other hand, examples of URLs that do not match our configured scope are http://www.abc.com/dir/ and http://www.foo.com/, since they have a different domain and different path, respectively. Note that a scope can be simply an individual URL.

Creating privacy-preserving lists. The website should keep the ACLs private to browsers to avoid revealing its defense strategy. Instead, it releases the privacy-preserving lists of scopes derived from the configurations in the ACLs, so that the lists will be used in our two-phase checking approach (see Section III-C). The lists will be stored in a *policy file* that is accessible by client browsers.

Publicizing the policy file. The website owner specifies the *base URL*, which states the exact hostname and path of the website under which the policy file will be stored. We assume that only the website owner has the write permission to store the policy file under the specified base URL. The base URL will be included in a response message to let the browser know where to download the policy file. Note that a browser may have downloaded multiple policy files from different websites. To choose the policy file for a given request, we use the *longest prefix match* based on the target URL of the request. For example, if the target URL is http://www.foo.com/~alice/login.php and there are two policy files with base URLs http://www.foo.com/ and http://www.foo.com/~alice/, then according to the longest prefix match, the browser chooses the policy file with the base URL http://www.foo.com/~alice/.

Checking. For each request to be sent to the website, the browser checks the initiating URLs and the target URL associated with the request against the scopes configured in the policy file. Since a scope may not state the complete URL, we *incrementally* check each URL. The main idea is to check all possible scopes associated with each URL, including all levels of domains starting from the top-level domain, as well as all levels of paths starting from the root path. That is, each scope is formed by the concatenation of each possible level of domain and each possible level of path. To illustrate, suppose that we are given a URL http://foo.com/a/b.html. Then there are six possible scopes to check: including (1) http://.com/, (2) http://.com/a/, (3) http://.com/a/b.html, (4) http://foo.com/, (5) http://foo.com/a/, and (6) http://foo.com/a/b.html. We then apply two-phase checking on all possible scopes (see Section III-C).

Checking all possible scopes may incur high overhead. In Section IV-B, we evaluate the impact of the number of scopes to check in two-phase checking. We also apply caching (see below) to mitigate the overhead of checking too many scopes.

Caching. If a URL has been checked, then the DeRef on the client side will cache the URLs in memory to eliminate checking on the subsequent requests for those URLs. We note that even with simple caching, we can significantly improve the performance of DeRef (see Section IV-A).

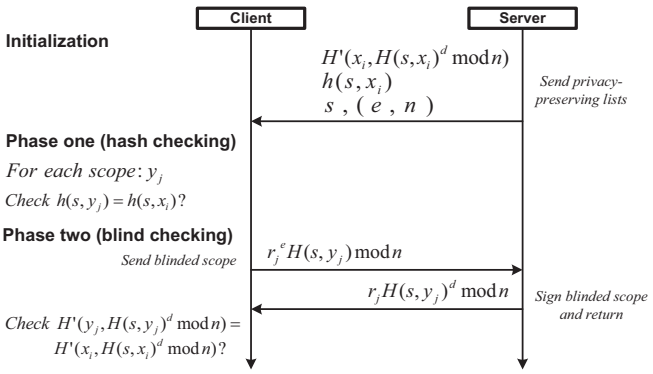


Fig. 2. Main idea of two-phase checking.

C. Two-Phase Privacy-Preserving Checking

We now present our *two-phase checking* approach that acts as a building block in DeRef. It allows the browser and the website to exchange information in a privacy-preserving manner. Let us assume that the website configures L legitimate scopes in an ACL (either T-ACL or I-ACL), denoted by x_i, where i = 1, 2, $\cdots$, L. Now, if the browser initiates a request to the website from URL y, then it checks if y belongs to any of the x_i's, so as to decide whether the request is within the configured scopes. To do this, the browser derives all possible scopes for a given URL y (see Section III-B) into y_1, y_2, $\cdots$, y_m, where m is the number of scopes that are derived from y. Then the browser checks if any y_j (j =1, 2, $\cdots$, m) equals any x_i (i = 1, 2, $\cdots$, L). Our privacy-preserving goals are: (1) the browser does not reveal y to the website and (2) the browser does not know the x_i's configured by the website, unless a scope of y matches any of these.

Figure 2 summarizes the idea of two-phase checking, which consists of two phases: *hash checking* and *blind checking*.

Hash checking. In hash checking, the website sends the browser a list of k-bit hashes of the configured scopes, i.e., $h(s, x_1)$, $h(s, x_2)$, $\cdots$, $h(s, x_L)$, where $h(.)$ is a function derived by the first k bits of some one-way hash function, and s is a random salt [29] that is sent alongside the hash list. When the browser initiates a request from URL y, it computes $h(s, y_j)$ (j = 1, 2, $\cdots$, m) and checks if it matches any $h(s, x_i)$ (i = 1, 2, $\cdots$, m). Note that the checking process does not reveal y to the website (i.e., goal (1) is achieved).

The value of k determines the degree of privacy that the website reveals its configured scopes. If k is large (e.g., k = 128 bits as in MD5) and $h(.)$ is collision resistant, then we claim that it is unlikely for two URLs to have the same hash value[1]. However, having a large k is susceptible to the *dictionary attack*. For example, after downloading the hash

[1]As of December 2010, the number of indexed webpages in the web space is about 22 billion (less than 2^{35}) [31], which is significantly less than the MD5 space size.

list, an attacker can use the popular URLs (e.g., the frequently visited URLs) and the salt s as inputs, and see if the resulting hash values equal any $h(s, x_i)$.

On the other hand, if k is small, then the browser cannot surely tell if a x_i is being configured since there are many false positives that create "noise" to prevent x_i from being fully revealed. For example, if k = 4, then there are $2^4 = 16$ possible values of $h(.)$. If $h(.)$ is uniformly distributed, then on average 1/16 of URLs in the entire web can potentially match a $h(s, x_i)$. However, we need to eliminate the false positives through blind checking (see below) to see if URL y is actually within a configured scope.

Blind checking. Blind checking is built on the *privacy-preserving matching protocol* [20], which uses Chaum's RSA-based blind signature [6]. We adapt the matching protocol to allow the browser to query the website in a privacy-preserving manner. Specifically, we use the potentially matched scopes returned by hash checking as inputs, and conduct blind checking as follow:

- **Initialization.** The website prepares a RSA public-private key pair (e, d) with modulus n. The public key (n, e) will be sent to the browser. Also, the website sends the list to the browser: $H'(x_i, H(s, x_i)^d \bmod n)$ for i=1, 2, $\cdots$, L, where $H(.)$ and $H'(.)$ are some one-way hash functions and s is the salt value (which is also sent to the browser). We assume that $H(.)$ and $H'(.)$ return a long-enough hash (e.g., 128 bits in MD5) so that it is unlikely for two inputs to return the same hash.
- **Step 1.** For each scope y_j (for j = 1, 2, $\cdots$, m) that matches any $h(s, x_i)$ in the first phase, it generates a random value r_j and sends the *blinded* hash $r_j^e H(s, y_j) \bmod n$ to the website.
- **Step 2.** The website signs and returns $r_j H(s, y_j)^d \bmod n$ to the browser, which removes r_j and retrieves $H(s, y_j)^d \bmod n$. It then computes and checks if $H'(y_j, H(s, y_j)^d \bmod n)$ equals any signed hashes $H'(x_i, H(s, x_i)^d \bmod n)$.

Since the browser sends only blinded hashes to the website, it does not reveal y to the website (i.e., goal (1) is achieved). Also, an attacker cannot feasibly launch the dictionary attack offline as in hash checking, since it is computationally infeasible to generate the signature of the website for a given input y without knowing the website's private key. Although the attacker can launch the dictionary attack *online* by querying the website with different values of y_j, the attack becomes more difficult than the offline one as it can easily alert the website if the querying rate is too high. By limiting the query rate of a browser, the privacy of the configured x_i's of the website is also protected (i.e., goal (2) is achieved).

We emphasize that using blind checking alone can still achieve our privacy-preserving goals. A key drawback is that there will be significant process overhead. In blind checking, the browser needs to take a round trip to send every potentially matched scope to the website and have the website sign the scope. Also, each signing consists of an expensive asymmetric cryptographic computation. Thus, we introduce hash checking to ignore any scopes that are guaranteed to be not configured, so as to reduce the overhead of blind checking.

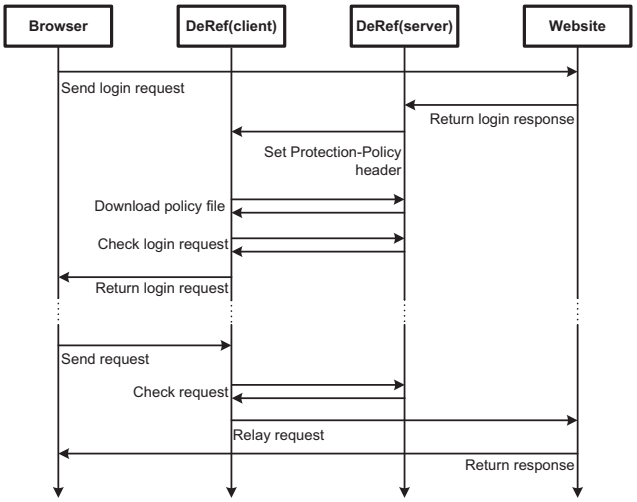


Fig. 3. Flow of DeRef.

D. Putting It All Together

DeRef is implemented on both client and server sides to examine the communication between the browser and the website. We now explain the flow of DeRef and how it enforces protection. Figure 3 shows the flow of DeRef.

Start-up. When a user signs in a website, it initiates a login request with valid authentication credentials. Then the website replies a login response, in which the server-side DeRef includes a new header `Protection-Policy`, whose syntax is `Protection-Policy: Last Update Time=[ Time stamp ]; Expiry Time=[ Time stamp ]; Base URL=[ Base URL ]`. This header serves two purposes: to indicate DeRef is implemented in this website and to state the base URL in which the policy file is stored. Also, the header includes the last update time and the expiry time of the policy file. If the policy file with the same base URL has been downloaded before, while the last update time remains the same and the expiry time is not yet reached, then the client-side DeRef will not download it again.

Downloading the policy file. If no up-to-date policy file is available, then the client-side DeRef downloads the policy file as specified in the base URL and stores it locally. However, an attacker may intercept and modify the policy file when it is being downloaded, for example, by deleting some of the entries in the policy file. To prevent the policy file from being modified, we propose to have it transmitted through HTTPS, which authenticates all message transmissions. Since the policy file is downloaded during the login process, we expect that HTTPS has been enabled by default.

Checking Process. The client-side DeRef performs two-phase checking on the login request that is previously relayed before returning the login response to the browser, so as to defend against any possible login CSRF attack. For subsequent requests originated from the browser, the client-side DeRef

checks the target URLs and the initiating URLs against the policy file. It strips any authentication credentials (i.e., cookies and HTTP authentication headers) from the requests and the corresponding responses if the requests are considered forged.

E. Implementation

We implement a prototype of DeRef to justify its practicality in deployment. DeRef is built on the components residing on both server and client sides. We now explain in detail the implementation on both sides, and address the deployment issues if only one side enables DeRef.

Server side implementation. The server-side DeRef is implemented in PHP, and hence is applicable in any PHP-enabled websites. There is a PHP program `genPolicy.php`, which generates the policy file with respect to the URLs defined by the website owner. Here, we use MD5 for hash operations and 1024-bit RSA for blind checking. In addition, we use the `header` function of PHP to specify a new custom HTTP header `Protection-Policy` to indicate the base URL that specifies the locations of the policy file. The browser can retrieve the policy file by visiting `genPolicy.php`.

Client side implementation. We implement a Firefox browser plugin compatible with Firefox versions 3 and 4. It retrieves the policy file from the base URL stated by the server-side DeRef, and inspects any outgoing requests for any forged requests. Our plugin intercepts requests and responses by listening to the events `http-on-modify-request` and `http-on-examine-response`, respectively, both of which are available in the Firefox implementation. Our implementation of the plugin consists of about 1000 lines of code.

Incremental deployment. DeRef requires the supports of both the client and server sides. If only one side has DeRef enabled, then our implementation is *backward compatible* with the normal operations without DeRef. To elaborate, if the client side implementation is absent, then the browser simply ignores the custom header `Protection-Policy` defined by the server side and will not download any policy file. On the other hand, if the server side implementation is absent, then the browser plugin will find that the custom header `Protection-Policy` is absent and will directly forward all outgoing requests.

IV. Evaluation

We now evaluate our implemented DeRef prototype in real network settings. The client-side DeRef is deployed as a plugin in Firefox 4.0, where the browser is deployed in a desktop PC with CPU 2.4GHz. We deploy the server-side implementation of DeRef in WordPress 2.0 [30]. We choose WordPress 2.0 as it has a known CSRF vulnerability [14], which allows us to test the security effectiveness of DeRef in defending against request forgery attacks. Note that we also verify that the modification we make in this version is applicable to the latest WordPress versions as well.

Suppose that Alice wants to host WordPress 2.0 on her personal website http://www.foo.com/~alice/ (note that we anonymize the real hostname here), on which she deploys DeRef. First, Alice needs to first configure T-ACL to specify the target URLs to be protected. Here, we include three scopes in T-ACL for WordPress, including:

- http://www.foo.com/~alice/wp-admin/,
- http://www.foo.com/~alice/wp-login.php, and
- http://www.foo.com/~alice/wp-comments-post.php.

The folder `wp-admin/` contains the webpages that manage all WordPress operations, and hence needs to be protected. We include `wp-login.php` so as to defend against the Login CSRF attack by restricting all login actions to be initiated from authorized URLs only. We also include `wp-comments-post.php`, which handles the comments posted by visitors.

Alice also needs to configure the valid initiating URLs in I-ACL to specify where the requests can be triggered to the protected scopes. Here, we assume that Alice includes http://www.foo.com/~alice/, meaning that all requests must be initiated from within Alice's website.

Both T-ACL and I-ACL are transformed into a privacy-preserving policy file (see Section III-B). Alice can store the policy file on http://www.foo.com/~alice/, from which different browsers can retrieve.

We set up a testbed that consists of three entities: a client browser (Firefox), a target website (WordPress), and a malicious website. We deploy all entities in the same local area network of a university department, so as to minimize the overhead of network transmission and enable us to focus on evaluating the performance overhead due to DeRef.

Summary of results. Overall, DeRef incurs minimal response time overhead (within 4%) when browsing insensitive webpages, which we believe are the majority of webpages that we visit in practice. The overhead of DeRef is mainly observed when we visit the sensitive webpages, but this tradeoff can be justified if security protection is necessary. By caching the already checked scopes, we can reduce the response time overhead to within 19%. In addition, we evaluate the overhead of two-phase checking in DeRef, and demonstrates how we tune the parameter settings to trade-off between performance and privacy preserving protection.

A. Performance Overhead of DeRef in Real Deployment

We first evaluate the performance overhead of our DeRef prototype in real deployment using Firefox and WordPress. Our goal is to understand the overhead of DeRef in surfing different types of webpages. We also evaluate how the use of caching (see Section III-B) on the client-side DeRef improves the performance.

Recall that DeRef uses two-phase checking. Here, we focus on the case where there is no false positive returned by hash checking by setting a large enough value of k (e.g., using k = 128 bits as in MD5). In Section IV-B, we evaluate how different values of k affect the performance.

We measure the *response time*, i.e., from the time when the browser sends the first request until it receives all response messages from the WordPress website. Note that the response time also includes the processing time of performing two-phase checking between the browser and the website. The

TABLE I
PERFORMANCE OVERHEAD OF DEREF IN DIFFERENT SETTINGS.

	Exp. A.1	Exp. A.2		Exp. A.3	
	Index	**Admin**	**Login**	**CSRF**	**Login CSRF**
No DeRef	132.44ms	174.56ms	230.06ms	64.41ms	55.83ms
DeRef (no cache)	137.23ms (4%)	654.72ms (275%)	585.15ms (154%)	117.57ms (83%)	111.09ms (99%)
DeRef (w/ cache)	138.01ms (4%)	200.27ms (15%)	254.27ms (11%)	76.07ms (18%)	66.4ms (19%)

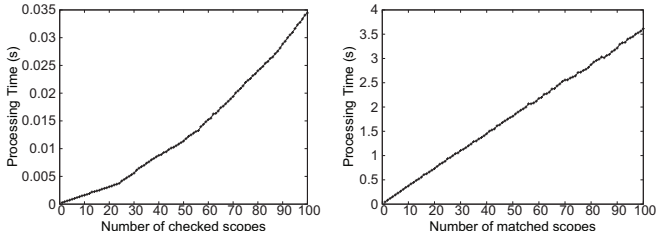
measurements are averaged over 100 runs. Table I summarizes the results of our experiments.

Experiment A.1 (Browsing insensitive webpages). We first consider the case where the browser visits an insensitive webpage that is not under the protection of DeRef, i.e., the URL of the webpage is not configured in T-ACL. Here, we measure the response time when we visit the index page index.php on WordPress. Since the index page is insensitive, DeRef does not need to perform blind checking (provided that no false positive is returned in hash checking). Thus, we expect that DeRef incurs minimal overhead. Table I shows that the additional overhead of DeRef is around 4%, which conforms to our intuition. Note that the performance is similar with or without cache.

Experiment A.2 (Browsing sensitive webpages). We next consider the case when the browser visits a sensitive webpage. In this case, the DeRef browser plugin will perform both hash checking and blind checking, to confirm that the URL of the sensitive webpage is in T-ACL and the initiating URL is in I-ACL. Here, we measure the time when the browser visits /wp-login.php and /wp-admin/ on WordPress from a legitimate initiating URL.

Table I shows that both cases incur significant performance overhead, mainly due to the RSA blind signature computation in blind checking. If no caching is used, then the overheads are 154% and 275% for /wp-login.php and /wp-admin/, respectively. We argue that such tradeoffs only apply to browsing sensitive webpages, by trading performance for security. Also, we can mitigate the overhead via caching, which stores the URLs that are known to be configured in T-ACL and I-ACL. When we visit the webpages /wp-login.php and /wp-admin/ again, the overheads decrease to 11% and 15%, respectively.

Experiment A.3 (Browsing malicious webpages). We now consider the case when we visit malicious webpages that trigger request forgery attacks to sensitive webpages. Here, we consider the CSRF and login CSRF attacks, in which forged requests are sent from our malicious website that we set up to the URLs /wp-admin/ and /wp-login.php, respectively. Note that in both cases, the initiating URLs are not configured in I-ACL, so DeRef only performs two-phase checking to confirm that the target URLs are configured in T-ACL. Thus, the number of URLs to be signed in blind checking is less than Experiment A.2. Overall, the additional overheads are 83% and 99% for CSRF and Login CSRF, respectively, when caching is disabled, and they reduce to 18% and 19%, respectively, when caching is used.



(a) Number of checked scopes in hash checking (b) Number of matched scopes in blind checking

Fig. 4. Experiment B.1: Scalability study of two-phase checking.

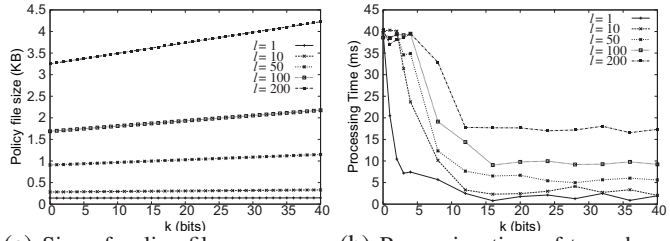
B. Performance Overhead of Two-Phase Checking

We now conduct a microbenchmark study on different configurations of two-phase checking.

Experiment B.1 (Scalability study of two-phase checking). We evaluate the scalability of DeRef in performing a large number of checking steps during two-phase checking (see Section III-B). We note that there are two potential performance bottlenecks in two-phase checking. First, we apply hash checking for all possible scopes derived from a URL, and its performance depends on the number of checked scopes. Second, we conduct blind checking for all matched scopes found in hash checking, and its performance depends on the number of the matched scopes.

We modify our DeRef browser plugin to generate a random number of scopes and measure the processing times of the two potential bottlenecks. Figure 4(a) shows the processing time of incremental checking versus the number of checked scopes. We observe that the processing time increases with the number of checked scopes, and it is within 35ms when the number reaches 100. We expect that this processing time has limited impact when compared to the overall performance in DeRef in real deployment (see Section IV-A), where the response time is on the order of 100ms. Figure 4(b) shows the processing time of blind checking (i.e., the time from the browser sending the blinded hashes for all matched scopes until the website returning the signed hashes) versus the number of matched scopes. We observe that the processing time increases linearly with the number of matched scopes, and it reaches 3.6 seconds when the number of matched scopes is 100. As shown in Section IV-A, the performance overhead can be significantly reduced by caching the already checked URLs.

Experiment B.2 (Trade-off between performance and privacy). Recall that the performance-privacy trade-off of two-phase checking is determined by the value of k (see Section III-C), which decides how much information is revealed in hash checking. In this experiment, we evaluate the impact of k. We first collect the top 500 website URLs on Alexa [2]. We then configure the first l of the 500 URLs in I-ACL, where l = 1, 10, 50, 100, or 200. We generate 500 requests from our DeRef browser plugin to the WordPress website that we set up, such that each request has its initiating URL hardcoded to each of the 500 collected URLs. For different values of



(a) Size of policy file (b) Processing time of two-phase checking

Fig. 5. Experiment B.2: Performance versus k for different numbers of URLs configured in I-ACL.

k, we measure the processing time for performing two-phase checking (i.e., hash checking, followed by blind checking if needed) on each initiating URL between the browser plugin and the WordPress website. We do not include the time of returning the response from WordPress, so the processing time of two-phase checking is less than the total response time that we measure in Section IV-A. Note that when $k = 0$, we assume that the browser directly conducts blind checking.

Figure 5(a) shows the size of the policy file versus k for different numbers of URLs configured in I-ACL. The size of the policy file increases with k and the number of URLs being configured in I-ACL, but the size is within 4.5 KB in all cases. Note that the policy file is downloaded once at the start-up phase and is cached until it expires (see Section III-D). Thus, we expect that the policy file itself introduces minimal overhead.

Figure 5(b) shows the processing time of two-phase checking. We observe that when k increases, the time used in two phase checking decreases, mainly because hash checking discovers most non-configured URLs and skips the second-phase blind checking. For example, if I-ACL contains only 10 URLs, then the processing time is reduced by 40% from k = 0 to k = 4. The trade-off is that more information of the configured scopes is revealed with a larger value of k. Another observation is that when the number of configured URLs (i.e., l) increases, the processing time is higher. The reason is that hash checking can only filter non-configured scopes. If more scopes are configured in an ACL, then more scopes need to be verified by blind checking as well.

V. Conclusions

We present DeRef, a practical privacy-preserving approach to defending against cross-site and same-site request forgery attacks. DeRef uses fine-grained access control to allow a website owner to decide how requests should be sent and received within protection scopes, so as to prevent forged requests from being initiated outside the scopes. We use two-phase checking as a building block that allows the browser and the website to exchange configuration information in a privacy-preserving manner. We implement a proof-of-concept prototype of DeRef, and demonstrate that it can successfully defend against request forgery attacks in real-life applications, while incurring justifiable performance overhead.

References

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. of ACM SIGMOD*, 2000.

[2] Alexa the Web Information Company. http://www.alexa.com.

[3] J. H. andCollin Jackson, H. J. Wang, and X. Fan. MashupOS: operating system abstractions for client mashups. In *Proc. of USENIX HOTOS*, 2007.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proc. of ACM CCS*, 2008.

[5] S. M. Bellovin and W. R. Cheswick. Privacy-enhanced searches using encrypted bloom filters. Technical Report CUCS-034-07, Columbia University, 2007.

[6] D. Chaum. Blind Signature for Untraceable Payments. In *Advances in Crypto.: Proc. of Crypto*, 1983.

[7] J. Cheng, H. Yang, S. Wong, P. Zerfos, and S. Lu. Design and implementation of cross-domain cooperative firewall. In *Proc. of IEEE ICNP*, 2007.

[8] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proc. of ACM CCS*, 2008.

[9] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Int. Symp. on Engineering Secure Software and Systems (ESSOS)*, 2010.

[10] K. Dolinar, J. Porekar, A. Jerman-Blažič, and T. Klobučar. Pervasive systems: Enhancing trust negotiation with privacy support. In *Workshop on Security and Privacy in Wireless and Mobile Networks*, 2006.

[11] B. S. Y. Fung. A Fine-Grained Defense Mechanism Against General Request Forgery Attacks. In *Proc. of IEEE/IFIP DSN Student Forum*, 2011.

[12] R. Hansen and J. Grossman. Clickjacking. http://www.sectheory.com/clickjacking.htm, 2008.

[13] C. Jackson and H. J. Wang. Subspace: Secure CrossDomain Communication for Web Mashups. In *Proc. of WWW*, 2007.

[14] M. Johns. Outdated advisory: Code injection via CSRF in Wordpress $<$ 2.03. http://shampoo.antville.org/stories/1540873, 2007.

[15] M. Johns and J. Winter. RequestRodeo: Client Side Protection against Session Riding. In *Proc. of the OWASP Europe 2006*, 2006.

[16] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *SecureComm*, 2006.

[17] D. Kristol and L. Montulli. HTTP State Management Mechanism, Oct 2000. RFC 2965.

[18] Z. Mao, N. Li, and I. Molloy. Defeating Cross-Site Request Forgery Attacks withBrowser-Enforced Authenticity Protection. In *Financial Cryptography and Data Security*, 2009.

[19] Mozilla. About mozilla add-ons. https://addons.mozilla.org/en-US/firefox/about.

[20] R. Nojima and Y. Kadobayashi. Cryptographically Secure Bloom-Filters. *Transactions on Data Privacy*, 2:131–139, 2009.

[21] Nytro. Using XSS to bypass CSRF Protection. http://packetstormsecurity.org/files/view/82676/Using_XSS_to_bypass_CSRF_protection.pdf, Nov 2009.

[22] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. In *Proc. of ACM CCS*, 2008.

[23] OWASP. Owasp top 10 for 2010. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[24] J. Ruderman. Same origin policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript, 2010.

[25] K. Seamons, M. Winslett, and T. Yu. Limiting the disclosure of access control policies during automated trust negotiation. In *Proc. of NDSS*, 2001.

[26] E. Sheridan. OWASP CSRFGuard Project. http://www.owasp.org/index.php/CSRF_Guard, 2010.

[27] The Open Group. Introduction to Single Sign-On. http://www.opengroup.org/security/sso/sso_intro.htm.

[28] W3C. Cross-origin resource sharing. http://www.w3.org/TR/cors/, Jul 2010.

[29] C. Wille. Storing Passwords - done right! http://www.aspheute.com/english/20040105.asp, Jan 2004.

[30] WordPress. http://en.wordpress.com/about/.

[31] WorldWideWebSize.com. http://www.worldwidewebsize.com/.

[32] M. Zalewski. Browser security handbook, part 2. http://code.google.com/p/browsersec/wiki/Part2, 2010.