Redefining Web Browser Principals with a Configurable Origin Policy

Yinzhi Cao, Vaibhav Rastogi, Zhichun Li†, Yan Chen and Alexander Moshchuk††
Northwestern University †NEC Labs America ††Microsoft Research

***Abstract*—With the advent of Web 2.0, web developers have designed multiple additions to break SOP boundary, such as splitting and combining traditional web browser protection boundaries (security principals). However, these newly generated principals lack a new label to represent its security property. To address the inconsistent label problem, this paper proposes a new way to define a security principal and its labels in the browser. In particular, we propose a Configurable Origin Policy (COP), in which a browser's security principal is defined by a configurable ID rather than a fixed triple <*scheme, host, port*>. The server-side and client-side code of a web application can create, join, and destroy its own principals. We perform a formal security analysis on COP to ensure session integrity. Then we also show that COP is compatible with legacy web sites, and those sites utilizing COP are also compatible with legacy browsers.**

I. Introduction

Web browsers have traditionally used the same-origin policy (SOP) to define protection boundaries between different web applications. According to SOP, a web site origin in the form of <*scheme, host, port*> serves as a label for the browser's security principals (isolated security domains). Each origin is protected from other origins in terms of resource access and usage.

With the advent of Web 2.0, modern web sites place new demands on browser's security that SOP was never designed to handle. Indeed, while intuitively simple, using web site origins to label browser principals has its limitations. Sometimes, SOP is too fine-grained. For example, contents from different web site origins (such as Gmail and Google Docs) may require unrestricted access to each other's resources, but SOP prevents browsers from rendering them as one principal. Other times, SOP is too coarse-grained. For example, it does not let browsers isolate logically different instances of web applications hosted on the same server, i.e., when one site hosts many distinct mashups, blogs, or user profiles, and it does not enable sites such as an email provider to run multiple, isolated sessions of the application in the same browser.

Faced with inflexibility of same-origin policy, web developers have worked around its limitations with a multitude of ad-hoc approaches to merge or separate a browser's security principal. For example, subdomains may merge with each other by setting *document.domain* to a common suffix, a practice prone to security problems [1]. MashupOS [2] proposes a sandbox tag together with a verifiable-origin policy to separate content from the same web site's security principal, which is particularly useful for mashups. Various cross-origin communication protocols are proposed [3]–[5] to break SOP for AJAX's XMLHttpRequest. However, while recent works [2]–[6] have studied ways of breaking SOP, they do not define different labels for those newly generated principals, which leads to a mismatch between principals and their origins (security labels). By utilizing the mismatch, an attack can camouflage the identity of a merged or separated principal and fool another server or client with the old SOP origin, the whole process of which is defined as *an origin spoofing attack*.

In this paper, we study a new way to label browser security principals. We propose a Configurable Origin Policy (COP), in which a browser's security principal is defined by a configurable ID specified by client browsers rather than a fixed triple <*scheme, host, port*>. Drawing inspiration from resource containers [7], we let the applications themselves manage their definition of an origin. That is, COP allows server-side and client-side code of a web application to create, join, destroy, and communicate with its own principals. In our scheme, one browser security principal can involve multiple traditional (SOP) web site origins, and various content from one traditional web site origin may render as multiple different principals. Fundamentally, COP origins break the long-standing dependence of web client security on domain names of servers hosting web content, and offer several compelling advantages:

- *Flexibility.* COP-enabled web applications can specify exactly which content from different domains can interact with one another on the client web browser. For example, Google may wish to let *gmail.com* and *docs.google.com* access each other's resources on the client. Moreover, with COP, we can disable many ad-hoc, error-prone and potentially incoherent workarounds for SOP limitations [1], such as subdomain communication via *document.domain*, while still allowing sites to function correctly. COP also supports many scenarios that required a separate security mechanism, such as sandboxing mashups [2], and those that are not well supported by existing browsers, such as allowing a site to run different isolated web sessions in the same browser, all under one uniform framework.
- *Consistency.* COP-based browser principal defines a new security label, a configurable ID, to represent its property. By examining the new label, other web clients and servers can easily differentiate the principal from old SOP-based browser principals and other new COP-based principals.
- *Compatibility.* Because we change the web's central security policy, we undoubtedly face the challenges of deployment and backward compatibility. To address compatibility,

TABLE I. COMPARING COP WITH EXISTING APPROACHES.

	SOP	SOP+Additions	Other Non-SOPs	COP
Flexibility	No	Yes	Partial	Yes
Consistency	Yes	No	No	Yes
Compatibility	Yes	Yes	Yes	Yes
Lightweight-ness	Yes	No	Yes	Yes
Security	Low	Medium	Low	High

we design SOP to be a special case of our new origin protocol, which makes COP compatible and secure with legacy web sites. COP-enabled web sites also remain compatible with existing browsers, since we convey our new design in existing protocols.

- *Lightweight-ness.* Our modification of WebKit to support COP is lightweight (327 lines), and our evaluation shows it has little overhead (less than 3%). Our modifications on web servers are also small. In examples we studied, less than ten lines were required to be inserted into existing server-side programs. To ease deployment, we also built a proxy that simulates modifications to web application code, and evaluated COP.
- *Security.* COP mitigates SOP specific attacks such as cross-site request forgery [8], and recently-discovered cross-origin CSS attacks [9]. For COP specific attacks, such as leaking the configurable ID (COP security label) through a careless programmer's mistake or an open HTTP connection, we adopt two corresponding defense mechanisms, *i.e.*, defaulting safe behaviors and transmitting IDs in an HTTPs channel. We also show that even if an attacker sniffs the configurable ID through an open HTTP connection, he cannot make additional damages through a COP design. Furthermore, we perform a *formal security analysis* by adopting and modifying an Alloy [10] web security model proposed and written by Akhawe et al [11]. The session integrity is ensured given the scope and the attack model. All details are in Section V.

Contributions. We are making the following contributions:

- Configurable Origin Framework (COF, referring to modifications necessary to support COP on web clients and servers) is the first **unified** framework to merge and separate web browser principals at both client and server side. No other approaches can achieve both functionalities at the same time.
- We are the first to propose and solve **origin spoofing attacks**. We define a new label in COF so that there is no confusing label for a merged or separated principal.
- COF is with **low** overhead compared with existing approaches in combining principals [12]. For example, when addressing the sub-domain communication problem [1], in which two domains completely trust each other, COF with native object access is 2X faster than libraries built upon postMessage channel [12].

II. MOTIVATION AND RELATED WORK

The work is motivated in this section. We present SOP, SOP plus all the additions, and other non-SOPs, together with their limitations, in Section II-A, Section II-B, and Section II-C. A high-level comparison is also shown in Table I.

A. Same-Origin Policy

The same-origin policy (SOP) is an access control policy defined in a client web browser, which allows only resources from the same <*scheme, host, port*> origin to access each other. Although SOP has been a good model with well-understood security properties, it has been inflexible for many modern Web 2.0 applications in the following two main scenarios.

- *Lack of Principal Cooperation.* SOP makes it very difficult for multiple domains to be "combined" into one single principal as shown in the following two cases. (i) Several domains, like mail.google.com and docs.google.com, may be controlled by a single owner who may want to allow sharing among the domains the owner controls. (ii) Since AJAX requests obey SOP, web applications cannot retrieve a URL from a different origin via XMLHttpRequest.
- *Lack of Principal Isolation.* SOP makes it very difficult for one domain to be "split" into different principals as shown in the following two cases. (i) A web site may require multiple isolated web sessions in one client browser. For example, a user may want to log in to multiple email accounts on the same provider site. (ii) To enrich users' experiences, many web sites embed third-party gadgets in iframes, such as Google Gadgets, Microsoft Widgets, and Facebook applications.

B. Existing Additions to Same-Origin Policy

To overcome such well-recognized SOP inflexibility, web sites resort to a multitude of different approach to "patch" SOP. They can be classified into three categories.

- *Splitting One Single Principal.* Various approaches [6, 13]–[17] create a separated environment to isolate different web sessions or third-party content at one client. New *iframe* tag in HTML5 [18] also supports a *sandbox* property to prevent access from the same origin.
- *Combining Multiple Principals at Server Side.* Many proposals [3]–[5] have been made to support cross-origin resource sharing (CORS). To support more fine-grained server-side merging, Content Security Policies (CSP) [19] and CSP-like mechanisms, such as SOMA [20], specifies access control policies at server side through HTTP headers or manifest files. Client browsers will be modified to enforce those specified policies.
- *Combining Multiple Principals at Client Side.* Since *document.domain*, which disobeys least-privilege principle, is insecure [1], Gazelle [21] disables *document.domain* and proposes that sites use these existing cross-principal communication channels, such as *postMessage*. Other work, such as Object Views [12], layers complexity on top of cross-principal communication channels to facilitate easier object sharing.

Limitations of Existing Additions to SOP. Fundamentally, all these approaches *together* fix flexibility problems in SOP, and thus we are showing the limitations of all existing additions to SOP as follows: inconsistency, heavyweight-ness, and insecurity.

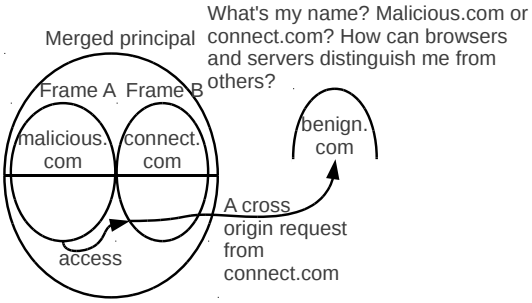

Fig. 1. Origin Spoofing Attack for CORS (A has full access to B through a JavaScript library over postMessage channel. But this new combined principal can still send requests as B. What if C - benign.com, trusts B but not A?).

1) Inconsistency: After existing approaches split or merge different SOP principals, unlike traditional SOP principals with the fixed triple (<*scheme, host, port*>) as security labels, those newly created principals do not own such labels. When browsers or servers check the security label of a principal, they can only inspect the old SOP label, which we call a mismatch between the principal and its security label (origin). In particular, an attacker can utilize the mismatch to camouflage a principal's identity and then fool a server or a browser with SOP origin, which is defined as an **origin spoofing attack**. We present the mismatch and its corresponding origin spoofing attack from two scenarios, namely, for a separated principal and a merged one.

First, there is a mismatch between a separated principal and its origin. For example, as supported by the new *iframe* feature in HTML5 [18], a.com can be separated into a gadget $G1$ (a.com/benign) and another gadget $G2$ (a.com/malicious) by specifying a *sandbox* property. In this case, $G1$ and $G2$ shares the same SOP origin but have different security properties from a.com.

A concrete *origin spoofing attack* utilizing this HTML5 feature is shown below. Say, for example, three frames, Integrator I (top), Attacker Frame A (middle) and Gadget $G1$ (bottom), are nested. Barth et al. [22] shows that if $G1$ sends a message to I, the attack frame A can navigate its child frame $G1$ to a malicious one to receive the reply from A. Therefore, all the up-to-date browsers check SOP origin of target frame in *postMessage*. However, after the HTML5 feature splits one single principal *a.com* into $G1$ and $G2$, the attacker can navigate the benign gadget $G1$ to a malicious one $G2$ in the same SOP origin *a.com*, which is posted by the attacker. The browser only checks the SOP origin (<*scheme, host, port*>) of $G2$ that is the same as $G1$ and thus the attack succeeds.

Secondly, there is a mismatch between a merged principal and its origin. For example, when Principal A from malicious.com is merged with another Principal B from connect.com by Object Views [12] or a similar approach built on top of *postMessage* to achieve full transparent object access, the merged principal AB actually represents both malicious.com and connect.com. However, AB does not have a new origin to represent its new property, which leads to *origin spoofing attacks* and *privilege escalation*.

A concrete example of *origin spoofing attacks* is shown in Figure 1. In a merged principal of A (connect.com) and B (malicious.com), A can ask B to send a cross-origin request with the origin connect.com to a third party server, say benign.com. Benign.com cannot recognize the request is actually from a merged principal AB consisted of both malicious.com and connect.com, since servers only check *origin* header or *referer* header for a cross-origin AJAX request.

In addition, other than *origin spoofing attacks*, the mismatch between a merged principal and its origin also leads to *privilege escalation*. Suppose frame A from facebook.com is merged with frame B from yelp.com. Then, B can access localStorage of the entire facebook.com domain, although B just wants to have full access to the specific frame, A.

2) Heavyweight-ness: In order to have the same flexibility as COP, all the additions have to be deployed upon current browser. The overhead is accumulated together. In particular, when two principals completely trust each other, *e.g., ads.cnn.com* and *www.cnn.com*, merging such two principals by a JavaScript library built upon *postMessage* channel [12] is heavyweight. We illustrate the performance degradation from two aspects, namely, in object access, and in merging more than two principals.

First, object accesses in two principals merged by techniques like Object Views are two times slower than native DOM accesses even with native JSON support [12]. That is due to that objects are serialized into JSON representation, transmitted through *postMessage* channel, and finally de-serialized back to objects.

Secondly, the performance decreases when the number of merged principals increases. For example, a principal from Twitter needs to join another principal merged from Yelp and Facebook. The Twitter principal needs to check and merge with both original principals separately with two *postMessage* channels. This becomes a serious scalability problem as the number of sites that want to communicate grows.

3) Insecurity: The usage of *document.domain* disobeys the least-privilege principle and can be insecure, as shown by Singh et al. [1]. Instead of *document.domain*, as shown in Object View [12] and Gazelle [21], web sites can use libraries over *postMessage* to facilitate communication. However, the authentication process through *postMessage* is often not used correctly even on popular web sites from reputable vendors, including Facebook Connect [23] and Google Friend Connect [24], as shown by Hanna et al [25].

C. Non-SOP Origin

As discussed in Section II-B1, additions to SOP are fundamentally inconsistent with SOP. In this section, we introduce previous attempts of defining non-SOP origins.

1) Finer-grained Label - (SOP + Something): A simple way of defining a non-SOP origin is to use SOP plus something. Current HTML5 specification [18] defines an origin as <scheme, host, port, optional extra data>. In history, there are several ways to define optional extra data, such as path [26]–[28], public key infrastructure [29, 30], ring [31], and capability [32].

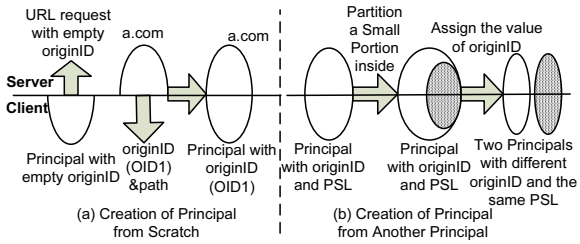

Fig. 3. Creation of Principal.

Limitations of Finer-grained Label. Labeling principals by finer-grained labels (< scheme, host, port, optional extra data >) has the following inflexibility.

- *Lack of Support for Merged Origins.* <scheme, host, port, optional extra data> cannot represent merged origins. For example, a finer-grained origin cannot represent a principal merged by frame A from a.com and frame B from b.com through *postMessage* channel.
- *Lack of Client-side Creation.* Host and port in SOP triple are defined by web servers. When client browsers need to create a new origin for an iframe, they have to send a request to the server and wait for the round-trip delay. Furthermore, in offline mode that is often supported by many modern web applications, servers are not reachable to assign new origins.

2) Verifiable Origin Policy - Not a New Label: MashupOS [2] proposes a new origin policy called the verifiable origin policy (VOP). "A site may request information from any other site, and the responder can check the origin of the request to decide how to respond." This is a great proposal that is also adopted in COP.

However, regardless of name similarity, *VOP is orthogonal to either SOP or COP.* In both SOP and COP, a web site needs to check the origin (SOP origin or COP origin) of the request from the client browser. VOP does not define a new label (origin) but instead stress the fact that origin needs to be checked at server side. In particular, MashupOS cannot merge two principals at client side, like in Facebook Connect [23] case and sub-domain communication.

III. Design

A. Concepts in COP

Configurable Origin Policy (COP) abandons SOP, disallows *document.domain*, and adopts a new origin policy. COP requires small modifications on both the client browsers and the server, as shown in Sections IV and VI-A. Yet, COP is compatible with both legacy browsers and legacy web servers as demonstrated in Section IV-E. In this section, we proceed to introduce several concepts fundamental to COP.

Resources. Resources represent contents inside client side browsers and web servers. Examples of resources from the server are HTML files, images, script files, etc. Examples of resources from the client are display, geolocation, and so on. Resources may be processed to generate further resources. For example, DOM is produced by rendering HTML files and modified by JavaScript code.

Principals. The concept of a principal, as borrowed from operating systems, in the context of web browsers is well discussed in previous work [2, 21]. It is an isolated security container of resources inside the client browser. Resources inside one principal are trusted by the principal itself. Resources outside principal X are not trusted by principal X but are trusted by the principal that the resources belong to. A principal is the atomic trustable unit in the browser.

We extend this concept in COP, where a principal is an abstract container that includes certain resources from both clients and servers with certain properties. A COP principal contains two parts, one on the server and the other on the client. The server-side's part of the COP principal is a worker, a thread or a process or a part of it, which serves the client. The client-side's part of the COP principal is what comprises a typical definition of a principal in a browser, an isolated container that is used to deal with contents from the server. For the rest of the paper, "principal" will refer to the COP principal in general.

Origins. An origin is defined as a label of a principal. Two principals that share the same origin will share everything between each other, which means they are essentially one principal. Two principals with different origins are isolated from each other. They can only communicate with each other through a well-protected channel.

OriginID. An originID is a **private randomly-generated identifier** used to annotate the origin of a principal. The originID is only known by the principal who owns it. Other principals cannot acquire the originID of principal X unless being told by principal X itself. In this sense, an originID is a *capability* to manipulate the principal it represents. OriginIDs are made arbitrarily hard to guess.

There are three reserved values of originIDs: *empty*, *default*, and *secret*. (i) The *empty* as a value of originID, specified by the client browser only, denotes a principal not associated with any content (hence the adjective *empty*). And the server will assign a value for the originID of such a principal. (ii) The *default* as a value of originID, denotes it is the same as the originID in current principal (both clients and server side included). (iii) The *secret* as a value of originID, denotes that the value of current principal's originID is not revealed by the owner.

Each resource in a principal will be labeled by an originID. With the originID, the client-side browser will decide in which principal to render the resource.

PublicID. A publicID provides a public identifier for a principal using which other principals can address this principal. It does not act as a capability like the originID to manipulate the principal it identifies. The publicID is designed to be publicly known. The browser maintains a table of correspondence of originIDs and publicIDs.

Principal's Server List (PSL). For each principal, the principal's server list (PSL), visible[1] to the users, is a list maintained by the browser to record all the servers or part of them

[1]One can adopt similar approaches in making the status of HTTPs certificate more noticeable by users for PSL.

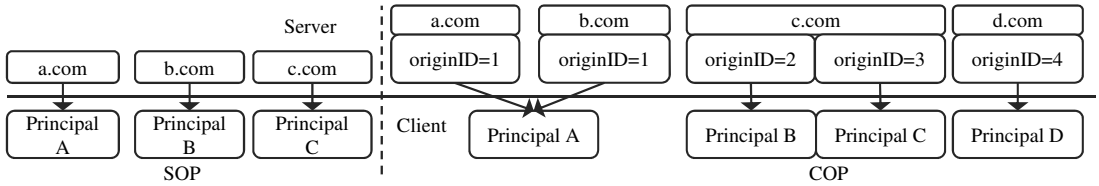

Fig. 2. Content-to-Principal Mapping in SOP vs. in COP (COP's core idea, and originID is simplified for easy understanding).

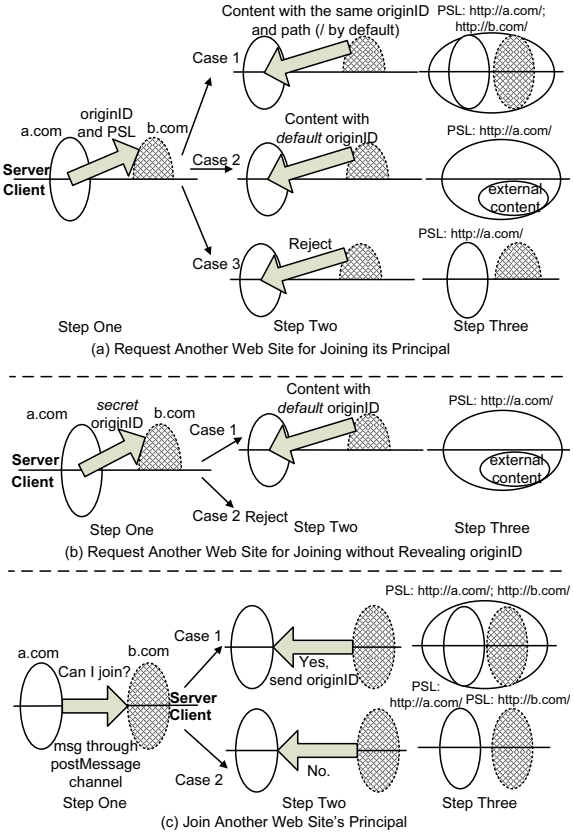

Fig. 4. Joining another Principal (notice that one server acquires originIDs of other principals through clients not directly from other servers).

that are involved in current principal by operations described later in Section III-C. Each server in the list is represented in the format of <scheme, host, port, path>. For example, $http://www.a.com/part1$ denotes that all the resources and sub-directories under part1 of $http://www.a.com$ are participating in current principal. By default, in order to align with SOP, if not specified by the server, the default path will be /, denoting that the PSL includes the whole SOP origin.

Note that PSL, which varies according to participated servers or part of them, is a list showing a principal's property, but *not* an access control list.

B. Configurable Origin Policy (COP)

With all these definitions, we can define our new origin policy, the configurable origin policy, as follows.

A principal (both server and client parts included) can configure any of its resources to an arbitrary but unique origin.

This means that a principal can change its contents' origin to an arbitrary value. The program at the server side of the principal can configure its originID. For example, the server may send its content to clients together with an originID. The program at the client side can also configure its origin. For example, a client-side JavaScript program may change the originID of a document tree.

On server side, unlike the SOP model, in which the content-to-origin mapping is fixed for all contents from the same <scheme, host, port> tuple, in the COP model, we allow the principal to configure its own origin. An SOP origin can be split into several COP origins. As illustrated in Section II-B, mashups and different web sessions are all examples. Similarly, multiple SOP origins can be combined together in configurable origins. For example, Google Docs and Gmail may want to share code or data dynamically, and thus they are better put in a single principal. Also, as illustrated in Section II-B, www.cnn.com and ads.cnn.com can be combined into cnn.com origin. Figure 2 clearly shows the differences between SOP and COP content-to-principal mapping.

On client side, the principal is also given more freedom. In the classical SOP model, the switching of origins is not allowed at the client side. *document.domain* reduces this restriction only a little, which may not be enough for some applications, and that too at the cost of possible malicious access to the principal. For example, a.com and b.com cannot share the same principal, even when using *document.domain*. Because in the COP model, the origin ID of a principal is not tied to its location, the origin of a principal may be arbitrarily decided at the client side.

C. Operations on a COP Principal

We define the following operations on a COP principal.

Creating a Principal. A principal can be created by a server or a client by giving a new originID, as shown in Figure 3.

Figure 3(a) illustrates the client requesting a URL together with an *empty* originID to *a.com* and the server sending the corresponding content with a new originID. In order to have multiple separate sessions from the same server, say for signing into multiple Gmail accounts, the server sends different originIDs to the client for different sessions. Given the different originIDs, the client browser renders the corresponding contents using different principals.

Clients can also create a principal as shown in Figure 3(b). A principal can assign a resource belonging to itself a new originID value to place this resource in a new principal. The child principal will inherit the PSL of its parent. Mashup isolation problem can also be solved using such client-side operations. Web integrators at client side can create different principals for content from different (distrusting) third parties by giving different originIDs based on the information provided by the server.

Joining an Existing Principal. Resources from one principal may wish to collaborate with, or *join*, resources from another principal. The joining process is discussed below.

As shown in Figure 4(a), a web site *a.com* may request to use a resource hosted on a different web site *b.com* under *a.com*'s principal — *a.com* can ask *b.com* to allow the resource to join *a.com*'s principal. Client browser supplies web site *a.com*'s originID and PSL when requesting that resource from *b.com*. If *b.com* agrees that this resource is allowed to join *a.com*, *b.com* will send the resource to *a.com* and attach *a.com*'s originID to it (case one in Figure 4(a)), and then client browser adds *b.com* plus the path specified by the server to the PSL of the current principal. If *b.com* does not want to participate in the principal actively, it will send the resource back with *default* originID (case two in Figure 4(a)), and then client browser will not change the PSL of the current principal. If *b.com* refuses to let this resource join *a.com*, it will fail to respond with the resource (case three in Figure 4(a)).

This join operation can be used for *document.domain* problem. For example, when the client has a www.cnn.com principal and sends a request to ads.cnn.com with the principal's originID and PSL, ads.cnn.com will agree to join the existing principal with the same originID. On the other hand, a bank web site will generally not join an existing principal of another web site.

Second, as shown in Figure 4(b), a web site *a.com* may request to use a resource hosted on a different web site *b.com* under *a.com*'s principal without telling *b.com* its originID. Client browser supplies a *secret* originID when requesting that resource from *b.com*. If *b.com* agrees to provide that resource, it will send the resource with a *default* originID. Otherwise, *b.com* can reject that request the same as case three in Figure 4(a). In this case, no matter *b.com* agrees or not, it will not be participating in that principal but just provide external resource. In other words, *b.com* cannot control the principal.

This pseudo-join operation can be used for supplying cacheable contents or those from content distribution networks. For example, *a.com* may request a cascading style sheet (CSS) by this operation since *a.com* does not want to reveal its originID to *b.com* and meantime *b.com* does not care which web site is using this style sheet.

Third, as shown in Figure 4(c), a resource or a principal from *a.com* may join another existing principal from *b.com*. The resource or principal from *a.com* acquires the originID of the other principal from *b.com* it wishes to join via an auxiliary communication channel (postMessage channel). By changing to this originID, the resource or the principal from *a.com* joins the other principal represented by this originID. And the PSL of the merged principal will also be the merging of those two principals' PSLs.

This case may be useful for collaboration among web sites. For example, Facebook Connect can be implemented with the join operation. A Facebook principal at the client browser may want to share information with another web site, say Yelp. The Facebook principal will create a new principal that is used for sharing and will then give the new originID to the other web site so that the other web site can join that newly created principal.

Communication inside a Principal. For client and server communication, accompanied by current originID, the communication with a server in PSL will be considered as a communication inside the current principal. The communication with a server not in PSL will *always* considered as a join operation (therefore attached with originID and PSL). Details can be found in Section IV-D1.

Pure client-side communication inside a principal is unrestricted. Any resource can freely access any other resource. For instance, one JavaScript object can call the methods of another object.

Communication between Principals. Communication across principals can be achieved with explicitly defined and regulated channels. Namely, we can use the postMessage channel and its communication protocols [22].

Destroying a Principal. Principals may destroy themselves or be destroyed by a user. For example, a user may close all the tabs and windows belonging to a principal and in this way destroy it.

IV. Implementation

Configurable Origin Framework (COF) requires both client-side and server-side modification. The server-side modification requires the server to send each resource together with the corresponding originID to the client. The client side needs to recognize the originID and put this resource into the corresponding principal.

Server-Side Modification. In our implementation, we modify the web application at server side so that resources in one web session will be allocated into one principal at client. This means the content-to-principal mapping is switched from SOP origin per principal to web session per principal. The concept of a session already exists in the present web, and denotes an information exchange process between the server and the client [33]. For example, when a user logs into his account on a web service, the web site sends a cookie to the user as an identity for further communication. Later on, any communication with that cookie is inside this session. The web server will check the identity cookie and reject requests without that cookie. In our paper, we adopt the existing concept of session. We will put resources of the same session[2] from the server into the same principal. Our server-side modification is discussed in Section VI-A. The rest of this section mainly deals with **client browser modification**.

Our client-side prototype implementation is based on WebKit [34], a popular open-source web browser framework. We demonstrate COF with the Qt browser that comes with WebKit. This browser uses WebKit's WebCore and JavaScriptCore. We insert 327 lines into WebKit to implement COF. The source code can be downloaded from configurable origin policy google code project [35].

[2]Notice that originIDs do not substitute session cookies, which still perform the same functionality as before.

```
bool SecurityOrigin::canAccess(const
                      SecurityOrigin* other)const {
 ...
 if (m_protocol == other->m_protocol) {
  if (!m_domainWasSetInDOM
                    && !other->m_domainWasSetInDOM){
   if (m_host==other->m_host&&m_port == other->m_port)
    return true;
  } else if (m_domainWasSetInDOM
                    && other->m_domainWasSetInDOM){
    if (m_domain == other->m_domain)
     return true;
    }
 }
 return false;
}
```

(a) Access Control in SOP

```
bool SecurityOrigin::canAccess(const
                      SecurityOrigin* other) const {
  if (m_originID!=""  || other->originID()!="") {
       return m_originID == other->originID();
      }
    else {
    SOP Access Control
    }
}
```

(b) Access Control in COP

Fig. 5. Access Control Implementation in SOP and COP.

In the rest of this section, we present originID and publicID generation, and WebKit COP enforcement respectively in Section IV-A and IV-B. Then, we discuss the association of originID with different resources in Section IV-C and IV-D. Finally, Section IV-E presents compatibility issues.

A. OriginID and PublicID Generation

The representation of originID is similar to that of a session cookie: a long and random string. We generate a 128-bit random number by CSPRNGs (cryptographically secure pseudorandom number generator) [36] and encode it as the value of an originID.

For a principal X, a publicID is an identifier which can be used by other principals to refer to principal X. Once a principal is created, a unique publicID is assigned to the principal automatically by the browser. The browser maintains a table of publicIDs and its corresponding information, such as the domain name and the description from the principal itself. Other principals can use a new API `getpublicID(domain name)`, which returns a list of publicIDs belonging to the domain, to query this table for the information,.

B. Enforcing COP

Access control methods or other isolation mechanisms are required to protect the boundary of a principal. In COF, to put contents from the same COP origin into one principal we need to replace SOP-based access control mechanisms with those based on COP.

`SecurityOrigin` is defined as a class in the WebKit implementation for controlling access over all domains. It adopts SOP and Figure 5(a) shows its core function. In COF, we modify it to employ originIDs for access control. The key part of new design is shown in Figure 5(b). Resources labeled with the same originID belong to the same principal and can freely access each other. Since only small modifications are required for COP on WebKit, we believe it will be relatively easy to adopt COP for other browsers as well.

HTTP Request	HTTP Response	HTML
HTTP/1.1 200 OK	GET /c.htm HTTP/1.1	<iframe originID=∗>
originID: *****	originID: ******	</iframe>
PSLPath:/part1	PSL: http://a.com/	
(a) OriginID and PSL with HTTP		(b) OriginID with HTML

Fig. 6. Association of originID and PSL with Different Resources.

C. Association of OriginIDs with Resources

A principal is associated with a container for resources. We classify resources into two categories, resources from servers and dynamically-generated resources. Each resource belongs to one principal, implying that each resource may be associated with an originID.

1) Origins for Resources from Servers: Resources obtained from servers, such as HTML, images, and some of plugin data, are mostly transmitted via HTTP protocol[3]. As shown in Figure 6(a), we add a header, named originID, in the HTTP protocol to indicate the originID of the resource. When the browser sees this field, it will add this resource to the principal with this originID.

In addition, HTML can be used for originID association. For example, the content inside an *iframe* tag may belong to another principal but this cannot be represented via HTTP headers alone. As shown in Figure 6(b), for some HTML tags that will cause an HTTP request or can create a *document* object[4], we can have an originID different from the one of the main document. Since HTTP headers, HTML tags, and HTML attributes are designed to be extensible, our new modifications are completely compatible with existing browsers; they simply get discarded in existing browsers.

Some content, such as some plugin data, is not transmitted by HTTP protocol. Such content belongs to the principal which requested it. For example, a Flash program in a Flash plugin creates a TCP connection. Later, contents transmitted in this TCP connection will have the same origin as this Flash program. The Flash program belongs to dynamically-generated resources, which will be discussed in Section IV-C2. In case the plugin program cannot be trusted, the whole plugin may be isolated in a different principal by assigning a different originID. We leave it as our future work to apply COP to plugin data.

2) Origins for Dynamically-Generated Resources: Dynamically generated resources refer to DOM, dynamic JavaScript objects, computed CSS, etc. These resources are derived from resources from servers. For example, DOM is generated from HTML parsing and JavaScript execution. There are two types

[3]HTTPS can be dealt with similarly because overall the confidentiality/integrity of the transfer channel problem is orthogonal to the problem of defining security principals, and can still use host names.

[4]Assigning a new originID is useful for only a few HTML tags, the ones that send another HTTP request or contains another DOM, such as *img* and *iframe*. For other HTML tags, because browser's access control is not fine-grained upon each DOM node, we cannot isolate them.

```
<script type="text/JavaScript">
//Inheritance--create an iframe with the same originID
(1) ifr1=document.createElement("iframe");
(2) document.getElementById("div1").appendChild(ifr1);
(3) ifr1.contentDocument.write("....");
//Dynamic Generation
// --create an iframe with a different originID
(4) ifr2=document.createElement("iframe");
(5) document.getElementById("div1").appendChild(ifr2);
(6) ifr2.contentDocument.write("....");
(7) ifr2.contentDocument.originID=generateOriginID();
</script> <div id="div1"> </div>
```

Fig. 7. Origins For Generated Resources.

TABLE II. DEFAULT BEHAVIORS FOR HTTP REQUESTS (COMPATIBLE WITH SOP).

HTTP Requests	Default Attached OriginID
Type a URL	*empty* originID from a new empty principal
HyperLink	URL in PSL: originID from the current principal
such as <a href="">	URL not in PSL: *empty* value
Scripts or Stylesheet	*secret* originID
Embeded Object, like	URL in PSL: originID from the current principal
iframe and img tag	URL not in PSL: *empty* value
XMLHTTPRequest	URL in PSL: originID from the current principal
	URL not in PSL: *secret* originID

of policies for association of originID with these resources: inheritance and dynamic generation.

Inheritance is the default enforced policy. As shown in Figure 7, we create an iframe *ifr1*, which inherits the same originID from the HTML document (line 1). However, an originID can also be specified dynamically. As shown line 4 of Figure 7, the iframe *ifr2* is created and is given a unique but different originID value through *generateoriginID()* (line 7).

D. Transfer of Resources

Resources are transferred from the server to the client and across browser principals. In this section, we describe how COF secures client-server communications and how the browser mediates cross principal communications.

1) Client-server Communication - HTTP: As shown in Figure 6(a), the HTTP exchanges between the server and client are associated with an originID. As in the spirit of verifiable origin policy [2], the originID of the request from a principal does not decide whether the corresponding response is accessible to the principal, and this principal is allowed to access the response only if the response carries the same originID as the principal's originID or *default* originID. (For *default* originID in the response, if the principal is empty, client browser will generate a new originID.) Now we discuss how the originID is used in the communication.

HTTP Request. HTTP requests in different operations have different behaviors. (i) Communication inside the current principal (a request to a server in PSL): launched from the current principal with its originID. (ii) Join operation (a request to a server NOT in PSL): launched from the current principal with its originID and PSL (iii) Create Operation (no matter whether the requested server is in PSL or not): launched from a different principal with that principal's originID.

TABLE III. ORIGINID IN HTTP RESPONSE ACCORDING TO DIFFERENT HTTP REQUESTS AND SERVER'S DECISIONS.

OriginID in HTTP Request	OriginID in HTTP Response			
	Join Operation	Comm inside Principal	Create Operation	Cacheable Content
empty	N/A	N/A	New Value	*default*
OID1	OID1	OID1	N/A	*default*
secret	N/A	N/A	N/A	*default*

To achieve those three requests, in COF, a principal can configure whether an HTTP request is from the current principal or a different one by specifying originID such as <img originID = "OID1">. However, to be convenient for programmers and compatible with SOP, the client browser can also attach an originID for those HTTP requests without originIDs specified explicitly, as shown in Table II. The default policy aligns with SOP.

HTTP Response. An HTTP response is generated by the web server according to the HTTP request received. Based on different originIDs in the request and operations that the web server wants to perform, the web server will attach different originIDs in the response as listed in Table III. For example, when the web server receives a request with a *empty* originID, it will send its response with a new originID and the client browser will adopt this originID as the originID for that empty principal.

An Example. Suppose a web page has an iframe <iframe originID= "OID1" src="example.com">. The browser will first create a principal with OID1 for the iframe. Then it will send a request with OID1 to example.com. If example.com agrees to join the principal, it will send an HTTP response with header "originID: OID1". Therefore, the browser will render the response inside the OID1 principal. If example.com does not agree, it will send a 404 HTTP response or other error messages.

2) Communications between Principals: The *postMessage* channel facilitates cross-principal communication at client side. The usage of *postMessage2* is like *popup.postMessage2 ("hello!", popup.publicID);* due to the attack in Section II-B1. While *postMessage* takes an SOP origin as its second argument and performs an SOP check, *postMessage2* replaces this with a publicID check. The attack in Section II-B1 is mitigated because a malicious gadget always has a different publicID from a benign one.

E. Discussion on Compatibility

We present whether COP feature can be compatible with existing web servers, existing browsers and new HTML5 features.

Compatibility with Existing Servers. Existing servers don't specify an originID in their transmission. However, we can still be backward compatible with existing servers. We use a SOP tuple as an originID because SOP can be viewed as a special case of COP. We can still assign each SOP origin a principal if originID is not specified. Other COP-enabled principals are not allowed to switch their originID to any SOP tuple. At the same time, we need to allow *document.domain*. The security of older web sites neither improves nor worsens.

Compatibility with Existing Browsers. There are two possible options to make COP-enabled servers compatible with existing client browsers. First, existing servers can detect the client browser and deliver content accordingly, but this can be inconvenient. We have taken the second approach to convey originIDs in a new protocol field that older browsers will ignore. We have shown earlier in the section how this is accomplished for HTML and HTTP.

Compatibility with new HTML5 features. Some new features in HTML5, such as localStorage and FileSystem, are designed to grant access to a long-term identifier. Those features can be still supported in COP. Take localStorage for example. It can be modified to allow access from those principals with the same PSL. Therefore, a merged principal from both Yelp and Facebook cannot access the localStorage of pure Facebook.

V. Security Analysis

In this section, we first analyze possible attacks on COP and how such attacks can be mitigated. Then, we discuss whether COP can help defend against existing web attacks, such as CSRF. In the end, we perform a formal security analysis based on an existing web security model.

A. COP-Specific Attacks and Mitigation

1) Leaking OriginIDs: OriginID is an essential and secret identity for a principal. We neet to prevent leaking originIDs.

Protecting OriginIDs. Given the similarity between originIDs and session cookies, methods of protecting session cookies can also be used for protecting originIDs.

- *Server-side protection:* Reusing protection mechanisms for session cookies. OriginIDs are generated *dynamically* and stored safely the same as session cookies on server side.
- *Protection during transmission:* HTTPS.
- *Client-side protection:* (i) Preventing originID access from a different principal through a sandbox approach [37] or JavaScript rewriting approaches [38]. (ii) Channel bound originID. Similar to channel-bound cookies [39], originID can be made channel-bound too. Even if an attacker acquires a channel-bound originID, he cannot authenticate it with the server via other connections.

How do we prevent originID leaks by a careless programmer's mistake? Two methods are adopted to prevent leaking originID by a mistake: (i) Defaulting safe behaviors. As shown in Table II, default behaviors of sending originIDs are restricted within known servers (if no merging occurs, there is only SOP server). (ii) Using secret originID. In case that the programmer does not know how to use originID correctly, he can use *secret* originID to prevent originID leaks.

Given two aforementioned protection mechanisms, we believe a web site will seldom send its originID to a malicious source by, for example, including third-party content through iframes in a wrong way. It is the same as the fact that a web site rarely includes a malicious script directly or carelessly sends its session cookie to a malicious server.

What if originID is leaked through an HTTP connection? Let us discuss a scenario where a web site *benign.com* is using HTTP and originID can be sniffed. We have the following two arguments: (i) Overall small chance. (ii) Even if it happens, there is no additional damages brought by COP.

First, the chance that the contents of *benign.com* can be manipulated is small. Clients need to visit *benign.com* in an open network such as coffee shop. Then, the attacker needs to lure the client to visit *malicious.com* in the same browser in order to join and manipulate *benign.com*'s principal.

Secondly, even if those two conditions are satisfied, the damage an attacker could make is the same as what he could do from sniffing the network and luring people to visit *malicious.com*.

- Contents of *benign.com* can be directly sniffed or acquired by a sniffed session cookie. Meanwhile, the attacker can also make changes to user's contents on benign.com by the sniffed session cookie.
- If *benign.com* does not have session cookie, phishing by altering contents in a merged principal is not different from the one in pure *malicious.com*, because after merging, *malicious.com* is in PSL, which is easily visible to users in COF.

What is the security implication for Principal A to give its originID to Principal B in order to configure Principal A? It means that Principal A totally trusts Principal B. For example, ads.cnn.com completely trusts www.cnn.com, however, they currently use *document.domain = 'cnn.com'* to communicate with each other, which is error-prone, as shown by Singh et al [1]. In COP, A (ads.cnn.com) can give its originID to B (www.cnn.com).

2) Potential Attacks when Combining SOP and COP: Interaction of web content following COP and web content following SOP may lead to attacks. In a web integrator where all its isolated gadgets are from the same domain, an attacker might modify a principal with originID back to a principal with SOP origin by removing the originID, so that the attacker can access another principal with the same SOP origin but not the same COP origin. We resolve this problem by always using COP when either of two principals is using COP. The originID of a SOP site will be derived from the SOP triple and hence will be different from every COP originID. In this case, two principals need to use the *postMessage* channel to communicate with each other.

Another attack is to integrate COP web sites with SOP web sites. For example, an SOP web site is embedded inside a COP web site using an iframe. COF can deal with this case because SOP is a special case in COP. If we don't find originID specified, we will consider the <scheme, host, port> to be a special originID which is different from any other originID specified by COP web sites. COF will always put contents in the iframe from SOP sites into a separate principal. We can always differentiate COP and SOP web sites because COP web sites will always have an originID HTTP header.

3) Principal Hijacking Attack: Given the similarity between a session cookie and an originID, the attacks to session cookies, such as session substitution/fixation [40], also need to be considered here. Translated to originID attack, a session substitution/fixation will be as follows.

An attacker M visits benign.com on his own and acquire the originID $OID1$ for his principal A. The attacker triggers the client to visit his own web page malicious.com and set the originID of the malicious principal B to $OID1$. The malicious principal B sends a request to benign.com. Then, benign.com will consider the $OID1$ to be within the attacker's principal A and return contents. Client user will see a web page from benign.com but controlled by M.

Defense. When the malicious principal B sends a request to benign.com, since benign.com is not in the PSL of B, the client browser will ask benign.com to join B with B's originID and PSL (malicious.com). Benign.com can recognize that B is controlled by malicious.com and thus decline the request.

B. Mitigating Existing Attacks

Cross-Origin CSS Attacks. Cross-Origin CSS attacks were described recently by Huang et al. [9]. The attacker may inject some crafted content into *example.com* using blog posts or wall posts and then use their site *attack.com* to import *example.com* as a CSS stylesheet. When a user visits *attack.com*, the confidential information of that user from *example.com* will be stolen. If COF were adopted here, because the server will check the originID of the principal that sends the request, *example.com* will reject the request, thus preventing *attack.com* from importing its contents as a stylesheet.

***Document.domain* Threat.** *Document.domain* threat is described by Singh et al [1]. For example, when a web page from *x.a.com* sets its domain to *a.com*, a web page from *y.a.com*, which is compromised by the attacker, can access the resource of that web page of *x.a.com* by setting its domain to *a.com*. This disobeys least privilege, as access control is relaxed too broadly.

In COF, only web pages that know the originID belong to the same principal. For the *document.domain* example above, even if an attacker compromises *y.a.com*, he cannot access any resource from *x.a.com* in a COP principal because he doesn't know the originID at client side.

Cross-Site Request Forgery (CSRF). CSRF is an attack which forces execution of unwanted actions from an end user on a web application in which s/he is currently authenticated [8]. A typical example of CSRF is an img tag such as

<img src = "http://bank.com/transfer.do?acct=A&amount=1000" width = "1" height = "1" border = "0"> .

When embedded inside a web page, it triggers browsers to fetch this link, causing the server to execute the "transfer" action. As we can see, there are several steps in CSRF. First, the link needs to be embedded on the web site. Second, the browser needs to send the request. Third, the server (the bank in this case) needs to allow this action. Defenses involve mitigation at any of the above steps.

Barth et al. [41] have analyzed CSRF and defenses against it comprehensively. They propose the origin header, which is similar to the referrer header but without the path and query parameters so as to ensure user privacy.

In COF, the originID header can effectively play the role of the origin header. In step three, the web server will use the originID (sent in step two) to determine if the request originated from its own principal and thus avoid CSRF attacks by declining the action from a different principal.

Origin Spoofing Attacks. As discussed in Section II-B, an origin spoofing attack is launched by a merged or separated principal using an old SOP origin to camouflage itself. In COF, we define originID as the new principal's label, which can be checked by a client browser or a web server, thus mitigating origin spoofing attacks.

C. Formal Security Analysis

Background. Akhawe et al. [11] abstract a formal web security model and build the model based on Alloy, "a model finder: given a logical formula, it finds a model of the formula" [10]. They apply the model upon five web security mechanism (the Origin header, Cross-Origin Resource Sharing, Referer Validation, HTML5 forms, and WebAuth), and discover two known vulnerabilities and three unknown vulnerabilities. A recent paper [16] also adopts their model to find design flaws.

Modeling. We modify their model to switch same-origin policy to configurable origin policy. The $owner$ property of $ScriptContext$ points to a COP origin instead of an SOP origin. All the operations are introduced for COP origin. For example, a create operation is as follows.

```
pred createCOPOrigin[aResp: HTTPResponse]{
 one originID:COPOrigin |
 originID !in
           (univ.theReqCOPOrigin&univ.theRespCOPOrigin)
 implies {
  (aResp.headers&RespOriginIDHeader).theRespCOPOrigin=
                                                 originID
 }
}
```

We check the COP origin in each HTTP response to let the response fit into different principals.

```
fact COPOriginMatch{
 all sc : ScriptContext, t:sc.transactions |
 sc.owner =
  (t.resp.headers&RespOriginIDHeader).theRespCOPOrigin
 or
  (t.resp.headers&RespOriginIDHeader).theRespCOPOrigin
                                      = defaultOriginID
}
```

Experiment Setup. The attack model that we are using is the *web attacker* model introduced by Akhawe et al. [11]. The attacker controls malicious web sites and clients but does not master the network. Therefore, he cannot sniff or alter the contents on the network. The Alloy codes of the attacker model are inherited from Akhawe et al [11].

```
check checkSessionIntegrity{
    no t:HTTPTransaction | {
      some t.resp
      some (WEBATTACKER.servers &  involvedServers[t])
    }
} for 5 but 0 ACTIVEATTACKER, 1 WEBATTACKER,
 1 COPAWARE, 0 GOOD, 0 SECURE, 0 Secret, 1 HTTPClient
```

Results. Alloy is not able to find any counterexample by operations that are not considered by COF, implying that the session integrity is ensured given the attack model and limited scope.

VI. Evaluation

First, we will discuss the practicality of deploying web applications using COP in Section VI-A. Next, we will evaluate COF's performance in Section VI-B. Then we discuss the compatibility in Section VI-C. We use a client with a 2.5GHz CPU with 16GB memory, and a server with a dual-core 1.6GHz CPU and 1GB memory, running Apache-PHP-MySQL. Both of these machines are on the same local network.

A. Deploying Web Applications

1) Migrating Existing Code: To fully deploy COF, we need browser support and server-side support. For server-side support, the server-side application code needs to be modified to use COF. By modifying several popular web applications, we demonstrate that such modifications are lightweight and easy perform.

Proxy Assistance. Because we don't have control over many web servers, we designed a COF server-side proxy that mediates communication between servers and clients. The COF proxy, which can be found at [35], adds COF support to unmodified web sites to demonstrate our idea.

CNN is using *document.domain* to merge two of its domains: www.cnn.com and ads.cnn.com. When we disallow *document.domain*, an advertisement iframe is missing because the JavaScript access between the main page and the iframe is denied. When deploying our proxy, and disallowing *document.domain* in COF, the CNN web site can still display its content correctly This demonstrates that COF can achieve site collaboration without using *document.domain*.

Server-side Modification. We show how to adopt COF upon server-side applications and demonstrate the relative ease of modifying server-side code. We take web applications with login sessions as an example. The login cookie or session ID assigned by the server is mapped to a unique originID. We can reuse the validation of session ID or login cookie as the validation of originID. We changed one popular web application − Magento, to demonstrate our approach.

Our example Magento [42] is a top eCommerce software platform used by more than 60,000 online stores. It is written in PHP and runs in the Apache, PHP and MySQL platform. Magento adopts PHP built-in session management. As shown in Figure 8, we just need to generate a unique originID for each session ID.

```
protected function _validate() {
  ...
  if (validation failed)  return false;
  if (checkPSL()) return false;
  if (isEmptyOriginID()) createOriginID();
  header('originID:'.getOriginID(session_id()));
  //get originID from sessionID-to-originID mapping
  return true;
}
```

Fig. 8. Modification on Varien.php of Magento. Each originID is mapped to a session ID. Session ID still takes its role in authenticating users, while originID is used to differentiate and isolate principals.

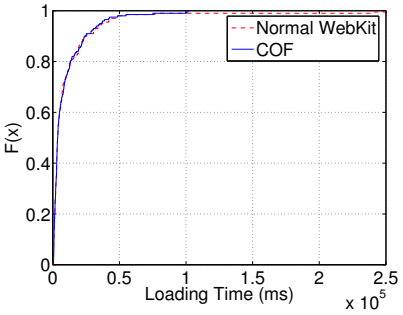

Fig. 9. CDF of Loading Time with COF and with Normal WebKit.

2) Utilizing New Features in COP: As an example, we create a mini web integrator using COP features below.

There are isolated mashups from the same domain in our web integrator. We create different originIDs for different gadgets.

```
<?php
    function generateOriginID() { ...
    }
    header('originID:'.generateOriginID());
    if COPSupportedBrowser() { ?>
     <iframe src="..." originID=
         <?php echo generateOriginID();  ?> >
     </iframe>
     <iframe src="..." originID=
         <?php echo generateOriginID();  ?> >
     </iframe>
     ...
<?php } else {...} ?>
```

B. Performance Evaluation

The loading time of web pages under COP is measured with WebKit modified to support COF and with a COF proxy. The loading time of web pages under SOP is measured with unmodified WebKit. We use the time when the request is made as the starting time of loading a web page and the time of firing of the JavaScript *onload* event as the end time of loading a web page. Alexa top 200 web sites [43] are evaluated.

Figure 9 shows the results. We compare the cumulative distribution function (CDF) of loading time under COP to the one under SOP. The curve is almost the same which means COF brings little delay. The results are not surprising because little time is spent in *SecurityOrigin* checks when compared to other tasks like rendering, parsing, and JavaScript execution.

C. Compatibility Evaluation

We use Alexa top 100 web sites and visually compare the sites rendered with a COP-enabled browser and with an unmodified browser. For some web pages that require login (like Facebook), we log in first. We also follow some of the links on the web page to explore the functionality of that web page. For example, we search with some keywords on Google. We interact with many web sites like Facebook, e.g., by clicking menus, posting messages on friends' wall, and looking at profiles of other people. As expected, all the 100 web sites show no difference when rendered with a COP-enabled browser and when rendered with an unmodified browser.

VII. Conclusions

In this paper, we propose COF, which uses configurable origins that can be dynamically changed by the web server and its client-side program. We change the traditional way of content-to-principal mapping and give the client and the server more freedom of configuring origins. At the same time, we also face that fact that COF requires both client and server side modification, which is actually very common among recent and popular web proposals, such as *postMessage* channel and new HTML5 iframe tag, due to the fast evolvement of web applications. Therefore, we believe COF will be adopted by the community in the future too.

Acknowledgments

We give our special thanks to Shuo Chen at Microsoft Research for his philosophical advices on the paper, Yi Yang at Northwestern University for his efforts on the initial version of the paper, Collin Jackson together with Zack Weinberg at CMU Sillion Valley for commenting on the draft of the paper, and all the anonymous reviewers for their thoughtful comments.

References

[1] K. Singh, A. Moshchuk, H. Wang, and W. Lee, "On the Incoherencies in Web Browser Access Control Policies," in *SP: IEEE Symposium on Security and Privacy*, 2010.

[2] H. J. Wang, X. Fan, C. Jackson, and J. Howell, "Protection and communication abstractions for web browsers in MashupOS," in *SOSP: ACM Symposium on Operating Systems Principles*, 2007.

[3] W3C Working Draft - Cross-Origin Resource Sharing. [Online]. Available: http://www.w3.org/TR/access-control/#origin

[4] XDomainRequest. http://msdn.microsoft.com/en-us/library/ie/cc288060(v=vs.85).aspx.

[5] A. Barth, C. Jackson, and I. Hickson. The HTTP Origin Header - IETF Draft. [Online]. Available: http://tools.ietf.org/html/draft-abarth-origin-00#section-6

[6] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in html5 applications," in *USENIX Security Symposium*, 2012.

[7] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *OSDI: Symposium on Operating Systems Design and Implementation*, 1999.

[8] Cross Site Request Forgery (CSRF) - OWASP. [Online]. Available: http://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29

[9] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson, "Protecting browsers from cross-origin CSS attacks," in *CCS: Conference on Computer and Communications Security*, 2010.

[10] D. Jackson, *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[11] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, "Towards a formal foundation of web security," in *CSF: the Computer Security Foundations Symposium*, 2010.

[12] L. Meyerovich, A. P. Felt, and M. Miller, "Object views: Fine-grained sharing in browsers," in *WWW: Conference on World Wide Web*, 2010.

[13] Private Browsing - Firefox. http://support.mozilla.com/en-us/kb/private+browsing.

[14] Google. Using multiple accounts simultaneously. http://www.google.com/support/accounts/bin/topic.py?hl=en&topic=28776.

[15] S. Crites, F. Hsu, and H. Chen, "OMash: Enabling secure web mashups via object abstractions," in *CCS: Conference on Computer and Communication Security*, 2008.

[16] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: get the security of multiple browsers with just one," in *CCS: conference on Computer and communications security*, 2011.

[17] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama, "SMash: Secure component model for cross-domain mashups on unmodified browsers," in *WWW: Conference on World Wide Web*, 2008.

[18] *HTML5: A vocabulary and associated APIs for HTML and XHTML*, W3C Std. [Online]. Available: http://www.w3.org/TR/html5/

[19] Content Security Policy - Mozilla. http://people.mozilla.com/~bsterne/content-security-policy/index.html.

[20] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "SOMA: Mutual approval for included content in web pages," in *CCS: Conference on Computer and Communications Security*, 2008.

[21] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal OS construction of the gazelle web browser," in *18th Usenix Security Symposium*, 2009.

[22] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," in *USENIX Security Symposium*, 2008.

[23] Facebook. Facebook connect. http://developers.facebook.com/blog/post/108/.

[24] Google Friend Connect - Google. http://code.google.com/apis/friendconnect/.

[25] S. Hanna, R. Shin, D. Akhawe, P. Saxena, A. Boehm, and D. Song, "The emperor's new APIs: On the (in)secure usage of new client-side primitives," in *W2SP: Web 2.0 Security and Privacy*, 2010.

[26] S. Ioannidis and S. M. Bellovin, "Building a secure web browser," in *USENIX Annual Technical Conference*, 2001.

[27] S. Ioannidis, S. M. Bellovin, and J. M. Smith, "Sub-operating systems: a new approach to application security," in *ACM SIGOPS European workshop*, 2002.

[28] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A safety-oriented platform for web applications," in *SP: IEEE Symposium on Security and Privacy*, 2006.

[29] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *CCS: Conference on Computer and Communication Security*, 2007.

[30] C. Reis, S. D. Gribble, and H. M. Levy, "Abstract architectural principles for safe web programs," in *HotNets: The Workshop on Hot Topics in Networks*, 2007.

[31] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin, "Escudo: A fine-grained protection model for web browsers," in *International Conference on Distributed Computing Systems - ICDCS*, 2010.

[32] T. Luo and W. Du, "Contego: Capability-based access control for web browsers - (short paper)," in *Trust and Trustworthy Computing - 4th International Conference - TRUST*, 2011.

[33] Session Definition - Wikipedia. http://en.wikipedia.org/wiki/Session_(computer_science).

[34] Webkit source codes. http://webkit.org/building/checkout.html.

[35] Google code home page of configurable origin policy. http://code.google.com/p/configurableoriginpolicy/.

[36] Cryptographically secure pseudo-random number generator. http://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_numb%er_generator.

[37] S. Tang, H. Mai, and S. T. King, "Trust and protection in the illinois browser operating system," in *OSDI: Proceedings of the 9th USENIX Symposium on Opearting Systems Design & Implementation*, 2010.

[38] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: vulnerability-driven filtering of dynamic html," in *OSDI: USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[39] Channel bound cookies. http://www.browserauth.net/channel-bound-cookies.

[40] A. Bortz, A. Barth, and A. Czeskis, "Origin cookies: Session integrity for web applications," in *W2SP: Web 2.0 Security and Privacy*, 2011.

[41] A. Barth, C. Jackson, and J. Mitchell, "Robust defenses for cross-site request forgery," in *CCS: Conference on Computer and Communication Security*, 2008.

[42] Magento Inc. Magento. http://www.magentocommerce.com/.

[43] Alexa Top Websites. http://www.alexa.com/topsites.