AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements

Mike Ter Louw Karthik Thotta Ganesh V.N. Venkatakrishnan

Department of Computer Science
University of Illinois at Chicago

Abstract

Web publishers frequently integrate third-party advertisements into web pages that also contain sensitive publisher data and end-user personal data. This practice exposes sensitive page content to confidentiality and integrity attacks launched by advertisements. In this paper, we propose a novel framework for addressing security threats posed by third-party advertisements. The heart of our framework is an innovative isolation mechanism that enables publishers to transparently interpose between advertisements and end users. The mechanism supports fine-grained policy specification and enforcement, and does not affect the user experience of interactive ads. Evaluation of our framework suggests compatibility with several mainstream ad networks, security from many threats from advertisements and acceptable performance overheads.

1 Introduction

On September 13, 2009, readers of the New York Times home web page were greeted by an animated image of a fake virus scan. Amidst widespread confusion, NY Times clarified the situation in an article [48], explaining the source of the rogue anti-virus attack was one of its advertising partners. Just two months prior, members of social web site Facebook were presented with advertisements (henceforth, "ads") deceptively portraying private images of their family and friends [38]. Facebook responded in an article [42] blaming advertisers for violating policy terms governing the use of personal images.

Publishers of online ads (like the NY Times and Facebook) face two serious challenges. They must ensure ads will neither violate the integrity of publisher web pages (as occurred with NY Times), nor breach confidentiality of user data present on publisher web pages (as occurred with Facebook). Ads are often tightly integrated into publisher web pages, and therefore must coexist with high integrity content and sensitive information. Typically, ad content is dynamically fetched from ad networks (e.g., Google AdSense) by the user's browser, leaving little opportunity for publishers to inspect and approve ads before the ads are rendered.

Online advertising is currently a lucrative market, expected to hit the US$50 billion mark in the U.S. during 2011 [52]. For many publishers, online advertising is an economic necessity. However, publishers have few resources enabling them to enforce integrity and confidentiality policies on ads. One common approach is for ad networks to screen each ad for potential attacks. This *passive* approach simply shifts the burden of protection from publisher to ad network. To enforce compliance, publishers must use out-of-band mechanisms (e.g., legal agreements), which leave the publisher vulnerable to any gaps in the ad network's screening strategy. Rogue ads may slip through and cause damage, as in the above, high profile examples.

Due to the dangers of rogue ads, publishers are in great need of an *active*, technological approach to protect themselves and their end users. Therefore, in this paper we confront the problem of rogue ads from a publisher-centric perspective. At a basic level, a publisher is a web application that includes dynamically sourced content from an ad network in its output. Our objective is to empower this web application to serve ads from mainstream ad networks, while protecting its end users from several threats posed by rogue ads.

1.1 Contributions

In this paper, we present ADJAIL, a framework that aids web applications to support rendering of ads from mainstream ad networks without compromising publisher security. Our framework achieves this protection by applying policy-based constraints on ad content. There are five significant contributions of our approach:

1. *Confidentiality and integrity policy specification and enforcement.* We define a simple and intuitive policy specification language for publishers to specify several confidentiality and integrity policies on advertisements at a fine-grained level. We provide a novel and con-

ceptually simple policy enforcement mechanism that offers principled security guarantees.

2. *Compatibility with ad network targeting algorithms.* Ad networks use targeting algorithms to select which ads to display, based on several factors such as page context and user behavior. In many cases, these algorithms are implemented as scripts that analyze publisher content to select and fetch appropriate ads to be displayed. Our approach supports these targeting scripts, with the added benefit of restricting the targeting script's access to sensitive data.

3. *Compatibility with ad network billing operations.* Ad networks employ complex billing strategies based on several metrics, including ad *impressions* (number of times an ad is shown) and mouse clicks. Furthermore, ad networks have mechanisms for dealing with click fraud [2]. To remain transparent to billing and click-fraud detection mechanisms, our approach preserves impression and click metrics.

4. *Consistency in user experience.* Our approach does not affect the user experience in interacting with ads, for any change in the user experience (in terms of content, position and interactivity) may reduce the effectiveness of advertising. Furthermore, ADJAIL highlights the security trade-offs that are required for ensuring consistency in user experience for certain types of ads (such as inline text ads).

5. *Satisfaction of practical deployment requirements.* Publishers should not have to expend significant labor in adopting a new framework, as this may make adoption prohibitively expensive. Furthermore, publishers should be able to deploy a solution that does not require end users to install new client software (e.g., browsers, plug-ins, etc.) or make changes to their existing client software. Therefore, we offer a practical solution that is easy to adopt, and works on mainstream browsers in their default settings, without any modifications.

1.2 Overview

The crux of our approach is a novel policy enforcement strategy that can be employed by the publisher to interpose itself transparently between the ad network and end user. The enforcement strategy starts by fetching and executing ads in a hidden "sandbox" environment in the user's browser, thus shielding the end user and web application from many harmful effects.

In order to preserve the user experience, all ad user interface elements are then extracted from the sandbox and communicated back to the original page environment, as permitted by the publisher's policy. This step enables the user to see and interact with the ad as if no interposition happened. All user actions are communicated back to the sandbox, thus completing a two-way message conduit for synchronization. Our approach ensures transparency with regard to the number of ad clicks and impressions by interposing on the browser's Document Object Model to suppress extraneous HTTP requests.

We have built a prototype implementation of ADJAIL that supports specification and enforcement of fine-grained policies on ads sourced from leading ad networks. The prototype is designed to be compatible with several mainstream browsers including Google Chrome, Firefox, Internet Explorer (IE), Safari and Opera. One minor limitation of our implementation (but not of our architecture) is that it is not compatible with IE 7.x or below. However, the current ADJAIL prototype is compatible with IE 8.0 and later.

We evaluate ADJAIL on the dimensions of ad network compatibility, security, and performance overheads. Our compatibility evaluation tested ads from six mainstream ad networks. We find that ADJAIL provides excellent compatibility for most ads. We also demonstrate the strong protection offered by ADJAIL from many significant threats posed by online ads. In our experiments, the currently unoptimized ADJAIL prototype encountered at most a $1.69\times$ slowdown in rendering ads.

The remainder of this paper is organized as follows: Section 2 provides the threat model, scope and related work. We provide the architecture and the main ideas behind ADJAIL in Section 3. Section 4 discusses the details in the implementation of ADJAIL. Our security, compatibility and performance evaluation appears in Section 5. In Section 6 we conclude.

2 Threat Model and Related Work

2.1 Threat model

Consider a publisher who wishes to carry ads on a *webmail* (Web-based email) application. We will use this as a running example throughout the paper to illustrate the various aspects of our framework. A screenshot from an actual webmail application we used in our evaluation appears in Figure 1. The top pane of the window presents the message list and the bottom pane presents the email message text. Four numbered advertisements also appear in the figure: (1) a banner ad that appears on top of the webmail page, (2) a skyscraper ad that appears as a sidebar, (3) an inline text ad that appears when the user's mouse hovers over an underlined word, and (4) a floating ad that overlays the image of a clock on the page.

These ads highlight two interesting challenges we need to overcome. First, the sidebar ad requires access to the email message text, which it mines to ascertain page context and select relevant ads for display (i.e., *contextual targeting*). The inline text ad also requires access to the message for contextual targeting and to integrate ads among the text. However, supporting these ads by providing ac-

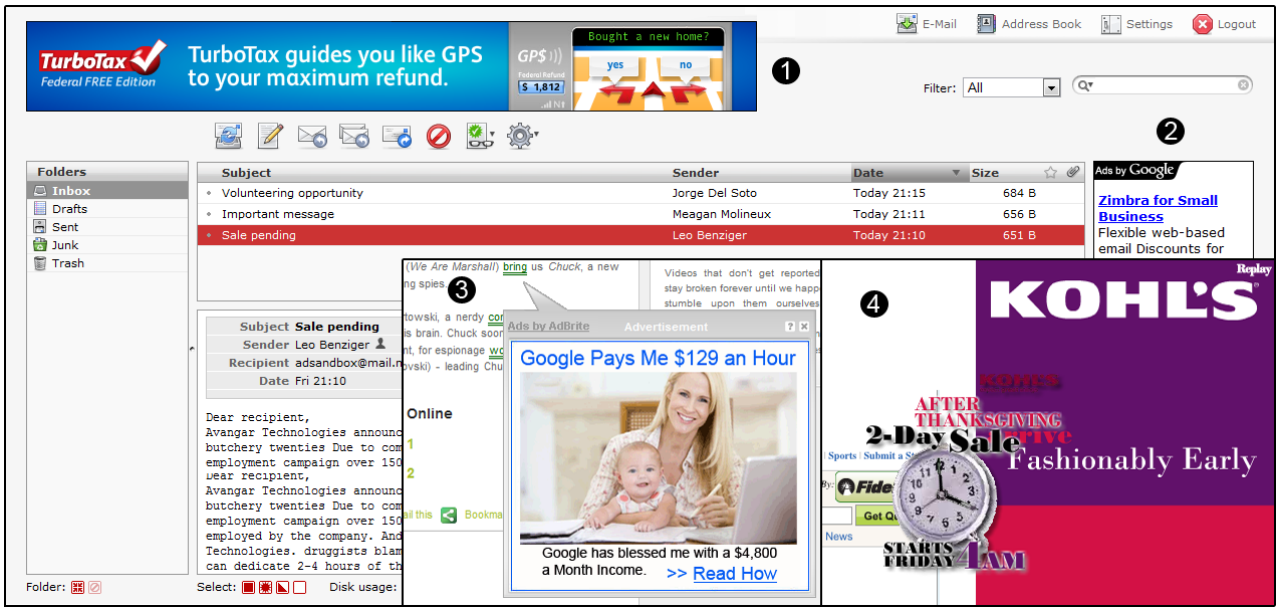

Figure 1: Samples of various ad types. A webmail application with (#1) banner and (#2) skyscraper ads. Also illustrated are (#3) an inline text ad and (#4) an floating ad.

cess to the entire message carries the risk of exposing private content (e.g., email addresses) to the ad script. Second, the floating ad requires access to the real estate of the page to place the image of the clock over the message text. However, providing access to the page real estate enables an ad to overlay content over the entire page, which may interfere with trusted interface components.

These common examples illustrate how ads require non-trivial access to publisher content and the screen, and will not work if such access is denied. Also, in all of the examples above, the ad content is loaded and rendered by a third-party ad script (an ad script example appears in Figure 4a). Ad scripts are given full page access by default, and thus pose threats to the confidentiality and integrity of page content. Our goal is to support the non-trivial access required by these and many other typical forms of ads, while addressing the security concerns of executing third-party ad scripts.

2.2 Threat scope

Web applications that display third-party content on client browsers are exposed to a wide variety of threats. It is therefore important to clarify our threat model, specifically on the nature of protections that we offer and the threats that are outside the scope of this work.

In-scope threats The broad threats that we address in this work are those targeted by recent efforts in the Web standards community for content restrictions (e.g., Content Security Policy [32, 43]). These policies are specified by a website to restrict the capabilities of third-party scripts, specifically with reference to access and modification of first-party (site owned) content, as well as control over the screen. Policies can be negotiated between a publisher and its customers, or directly reflect the site security and privacy practices.

Our framework provides a means for specification and enforcement of such policies. For instance, in our webmail example, an integrity policy can be enforced such that email message content cannot be tampered with, but can still be read (for contextual targeting of ads). Publishers may also choose to restrict where ads can appear on the page.

Publishers can also use our framework to enforce policies about confidentiality of content. For instance, a publisher can enforce a policy that mail headers and email "address books" (containing private email addresses) cannot be read by ads. For the Facebook attack in §1, a policy specifying confidentiality of user images, combined with our enforcement mechanism, would have prevented the attack.

Out-of-scope threats Many security threats posed by ads (and other third party content) have been identified by the security community. Recently, there has been intense research in this area which can complement our approach for protection against specific attacks. In particular, our work does not address the threats listed below. In this section we omit threats for which publishers can readily deploy strong protection (e.g., cross-site request forgeries).

1. *Browser security bugs.* We do not address browser vulnerabilities such as drive-by-downloads [49, 36, 5], attacks launched through plug-ins [24], vulnerabilities in image rendering [23] and so on.

2. *Opaque content.* Our approach leverages web content introspection capabilities of JavaScript, and is therefore most capable of enforcing fine-grained control where such transparency is available. Although our approach provides coarse-grained confidentiality and integrity protection from opaque content (e.g., Flash), the many possible attack vectors from these binary formats require special treatment [13].

3. *Frame busting & navigation attacks.* These are difficult attacks for any dynamic policy enforcement mechanism to prevent, due to the limited API exposed by browsers. A detailed discussion of protection measures against frame busting has been explored [39] and could be used to enhance our approach.

4. *Behavior tracking attacks.* These are attacks that track a user across multiple sites and sessions through use of cookies. These could be addressed by users choosing restrictive cookie policies, though such policies may interfere with the functionality of some web sites.

5. *Attacks through side channels.* Sites can track users through side channels, such as the cache timing channel [11], the "visited links" feature of browsers [19] and so on. It is difficult to defend these vectors without browser customization, which is impractical for publishers to deploy.

2.3 Related Work

Privacy and behavioral targeting A few recent approaches have looked at the problem of addressing security issues in online advertising. Privads [15] and Adnostic [47] address this problem primarily from a user privacy perspective. They both rely on specialized, in-browser systems that support contextual placement of ads while preventing behavioral profiling of users. In contrast, our work mainly focuses on a different, publisher-centric problem of protecting confidentiality and integrity of publisher and user-owned content. Our work is also aimed at providing compatibility with existing ad networks and browsers.

Restricting content languages There have been a number of works [9, 6, 28, 29, 30, 12] in the area of JavaScript analysis that restrict content from ad networks to provide security protections. These works focus on limiting the JavaScript language features that untrusted scripts are allowed to use. The limitation is enforced statically by checking the untrusted script and ensuring it conforms to the language restrictions. Only those language features that are statically deterministic and amenable to analysis are allowed. Since much of the policy enforcement is done statically, these solutions typically have good runtime performance. In the cases of FBJS [9] and ADsafe [6], untrusted scripts are allowed to make calls to an access-controlled DOM (document object model) interface, which incurs some overhead but affords additional control. The cost in employing a restricted JavaScript subset is that ads authored by many advertisers may not conform to this subset, and therefore require re-development of ad script code. In contrast, ADJAIL neither imposes the burden of new languages nor places restrictions on JavaScript language features used in ad scripts. The only effort required from a publisher that incorporates ADJAIL is to specify policies that reflect site security practices.

Code transformation approaches Many recent approaches [37, 53, 22, 14, 34, 10, 35] have been pursued to transform untrusted JavaScript code to interpose runtime policy enforcement checks. These works cover the many diverse aspects by which third-party content may subvert policy enforcement checks. Since these works are aimed at general JavaScript security, they are not specialized to the problem of securing ads for publishers, where the main issue is ensuring transparent interposition. This is to avoid any conflict with ad targeting and billing strategies employed by ad networks. The recommended method of transforming JavaScript dynamically by a publisher involves using a proxy (e.g., for handling scripts sourced from an external URI). However, routing all ad script HTTP requests through a script-transformation proxy may appear suspicious to click-fraud detection mechanisms [2] employed by the ad network.

Publisher-browser collaboration An alternative approach is for a publisher to instruct a browser to enforce the publisher's policies on third-party content, leaving the enforcement entirely to the browser. This publisher-browser collaborative approach is a sound one in the long term to enforce a wide range of security policies as illustrated in BEEP [21], End-to-End Web Application Security [8], Content Security Policies [43] and ConScript [33]. The main positives of this approach are that it can enforce fine-grained policies with minimal overheads. The primary drawback is that today's browsers do not agree on a standard for publisher-browser collaboration, leaving a large void in near-term protection from malicious third-party content.

3 Architecture

Let us revisit our running example of a publisher who wishes to carry ads on a webmail application. Recall that the publisher embeds an ad network's JavaScript code within the HTML of the webmail page to enable ads. In the benign case, this JavaScript code scans the webmail user's email message body to find keywords for contextual ad targeting, then dynamically loads a relevant ad. For simplicity, we refer to the ad network's JavaScript and an advertiser's JavaScript (the latter loaded dynamically by the former) as *the ad script*. This section gives a high level overview of how we prevent the ad script from per-

forming a variety of attacks against the publisher and end user.

Our approach is to initially confine the ad script to a hidden isolated environment. The hidden environment is locally and logically isolated [27, 44] as opposed to requiring additional physical and remote resources [31]. We then detect effects of the ad script that would normally be observable by the end user, had the script not been confined by our approach. These effects are replicated, subject to policy-based constraints, outside the isolated environment for the user to observe and interact with. User actions are then forwarded to the isolated environment to allow for a response by the ad script. Thus we facilitate a *controlled cycle* of interaction between the user and the advertisement, enabling dynamic ads while blocking several malicious behaviors.

3.1 Ad confinement using shadow pages

As a basic policy, the publisher wants to ensure ad script does not access the publisher's private script data. If this policy is not enforced, ad script can read the sensitive `document.cookie` variable and leak its contents, enabling the recipient of the cookie to hijack the authenticated user's webmail session. Furthermore, ad script should not be allowed to read confidential user data from the page (e.g., email message headers and address book entries). Such data is normally accessible via the browser's document object model (DOM) script interfaces.

To enforce the publisher's policy, we leverage browser enforcement of the *same-origin policy* (SOP) [50], an access control mechanism available in all major JavaScript-enabled browsers. Web browsers enforce the SOP to prevent mutually distrusting web sites from accessing each other's JavaScript code and data. As a script instantiates code and data items, the browser places each item under the ownership of the script's origin principal. *Origin* principals are identified by the domain, protocol and port number components of the script's uniform resource identifier (URI). Whenever a script references code or data, both the script and item being accessed must be owned by the same origin, else access is denied.

To enforce the publisher's ad script policy, we begin by removing the ad script from the publisher's webmail page. Next, we embed a hidden `<iframe>` element in the page. This `<iframe>` has a different origin URI, thus invoking the browser's SOP and thereby imposing a code and data isolation barrier between the contents of the `<iframe>` and enclosing page. Finally, we add the ad script to the page contained in the hidden `<iframe>`. We refer to the hidden `<iframe>` page as the *shadow page*, and the enclosing webmail page as the *real page*. This transformation just described is depicted in Figure 2.

In the process of rendering the real page, the browser renders the shadow page, executing the ad script within. Our use of the SOP mechanism effectively relegates this ad script to an isolated execution environment. All access by ad script to code or data in the real page will be blocked due to enforcement of the SOP. Furthermore, the ad script can not retrieve confidential address book data via DOM interfaces, as access to those APIs are denied by SOP. We can say the publisher's basic policy is enforced, because (1) all such ad scripts are relocated to the shadow page, and (2) the browser correctly enforces the SOP.

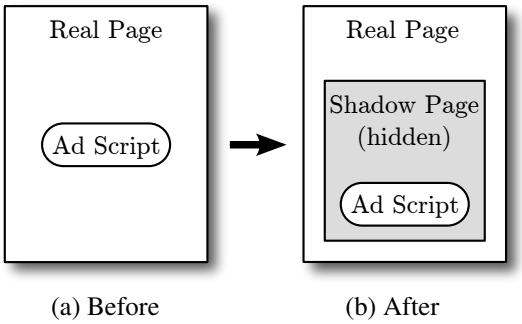

Figure 2: Relocating the ad script to a hidden shadow page invokes the browser's *same-origin* policy for confinement.

3.2 Controlled user interaction with ads

Consider an ad script that loads a product image, or *banner*. Normally the banner appears on the real page, but since the ad script runs in the shadow page, the banner is rendered on the shadow page instead. Without further steps, the webmail user viewing the real page will never see this banner because the shadow page is hidden. We now describe how the user is able to interact with the shadow page ad by content mirroring (§3.2.1) and event forwarding (§3.2.2), subject to policy-based constraints (§3.2.3).

3.2.1 Ad mirroring

A detailed view of the real and shadow pages that depicts mirroring of ad content is shown in Figure 3. We add Tunnel Script A to the shadow page that monitors page changes made by the ad script (①), and conveys those changes (②) to the real page via inter-origin message conduits [1, 20]. We add complementary Tunnel Script B to the real page that receives a list of shadow page changes and replicates their effects on the real page. Thus when ad script creates a banner image on the shadow page, Tunnel Script A sends a description of the banner to Tunnel Script B, which then creates the banner on the real page for the end user to see.

Special care is taken to prevent sending redundant HTTP requests to the ad server during the mirroring process, as such requests can interfere with an ad network's record keeping and billing operations. These details are discussed at depth in §4.3.2.

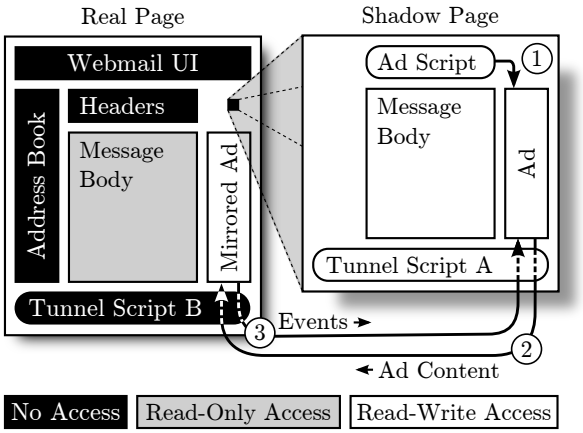

Figure 3: Overview of ADJAIL integrated with a webmail application. Ad script is given read-only access to email message body for contextual targeting purposes. Ad script can write to designated area to right of message body. Confidential data such as address book and mail headers are inaccessible to ad script.

3.2.2 Event forwarding

Ads sometimes respond in complex ways to user generated events such as mouse movement and clicks. To facilitate this interaction, we capture events on mirrored ad content and forward these events (Figure 3, ③) to the shadow page for processing. For example, if the ad script registers an `onmousemove` event handler with the original banner image, we register our own (trusted) event handler on the mirrored banner image. Our handler listens for the mouse-move event and forwards it to the shadow page's banner via an inter-origin message. If the ad script responds to the mouse-move event by altering the banner or producing new ad content, these effects are replicated on the real page by our mirroring strategy outlined above.

3.2.3 Ad policies

All messages sent between the real and shadow pages are mediated by our policy enforcement mechanism. This mechanism enforces policy rules which are specified by the publisher as annotations in the real page HTML. For the webmail example in Figure 3, the following access control policies are specified (shown in bold):

```
1 <div id="MessageBody"
2   policy="read-access: subtree;">
3   Message body text here...       </div>
4 <div id="Advertisement"
5   policy="write-access: subtree;"></div>
```

The policy in line 2 allows the ad script read-only access to the email message body. Read-only access is enforced by initially populating the shadow page with content from the real page (ref. Message Body regions in Figure 3). If ad script makes changes to read-only content, those changes are not mirrored back to the real page. Any attempts to mirror those changes to the real page message body (perhaps by a compromised Tunnel Script A) are denied.

The policy in line 5 permits the ad script write access to the sidebar on the right of the email message body. This is the region where the ad banner is to appear. When ad script creates content in the shadow page sidebar, this policy allows our mirroring logic to reproduce that content on the real page sidebar.

An implicit policy restriction on all mirrored content is that executable script code can not be written to the real page. To enforce this restriction, we only mirror items conforming to a configurable whitelist of static content types. Note this script injection threat is distinct from cross-site scripting (XSS), which the site can defend against using well-researched approaches (e.g., [46]).

The full policy language (detailed in §4.1) supports content restrictions to block Flash, deny the use of images (for text-only ads), restrict the size of ads, and more. These constraints can be tailored to the minimum compatibility requirements of individual ad networks, which we show in §5 can prevent attacks such as clickjacking [17].

Our policy enforcement mechanism is implemented on the real page as part of Tunnel Script B. As stated earlier, the ad script can not access the real page (including Tunnel Script B) due to SOP enforcement. Therefore ad script can not tamper with our policy enforcement mechanism.

4 Implementation

The implementation of ADJAIL is described in the context of a single webmail page with an embedded ad, which is integrated with our defense solution. We present the policy language used to restrict ads in §4.1. Then in §4.2 we describe how the real and shadow pages are constructed. §4.3 explains how we facilitate interaction between the two.

4.1 Policies

By default, ad script is given no access to any part of the real page unless granted by policies (i.e., *default-deny*). An implicit policy we always enforce is that ad script can not inject script code onto the real page, nor execute script code with privileges of the real page. We now describe in detail the individual permissions granted by policies, how policies are specified, and how multiple policies are combined to form a composite policy.

Permissions ADJAIL supports a basic set of permissions that control how ads appear on the real page and how ads can behave, summarized in Table 1. We define a *policy* as an assignment of values to each of the permissions. Our permissions have been designed iteratively by studying requirements of ads from several ad networks, and our results presented in §5 show the supported permissions can be composed to form useful advertisement policies.

The permissions `read-access` and `write-access`

Permission	Values	Description / Effects
read-access	none†*, subtree	Controls read access to element's attributes and children.
write-access	none†*, append, subtree	Controls write access to element's attributes and children. Append is not inherited.
enable-images	deny†*, allow	Enables support in the whitelist for <img> elements, CSS background-image and CSS list-style-image properties.
enable-iframe	deny†*, allow	Enables <iframe> elements in whitelist.
enable-flash	deny†*, allow	Enables <object> elements of type application/x-shockwave-flash in whitelist.
max-height, max-width	0*, n%, n cm, n em, n ex, n in, n mm, n pc, n pt, n px, none†	Sets maximum height / width of element to n units. Smaller dimensions are more restrictive. When composing values specified in incompatible units, most ancestral value wins.
overflow	deny†*, allow	Content can overflow boundary of containing element if allowed.
link-target	blank*, top, any†	Force targets of <a> elements to _blank or _top. Not forced if set to any.

Table 1: Permissions that can be set in policy statements. *Most restrictive value. †Default value.

control what parts of the page ad script may read from or write to. Of particular interest is the append setting for write-access. This level of access allows ad script to add child content to an element, but neither read nor modify existing children of the element. Any appended children are automatically given a policy attribute set to write-access: subtree;. Some ads, such as the clock ad (#4) in Figure 1, require the append permission to add floating (i.e., absolutely positioned) content to the <body> element. In supporting these ads, we don't want to grant subtree write access to the <body> element, as that would enable a malicious ad to overwrite the entire page. Granting append access in this case is safer as it adheres to the principle of least privilege [40].

Part of our policy enforcement is a whitelist of HTML elements, attributes and CSS properties that ad script is allowed to write to the real page. Although this whitelist can be modified by the publisher at a low level, we support the following higher-order controls for tuning the whitelist. Ads are text-only by default; to enable images, the enable-images permission can be set to allow, thus expressing a publishers content restrictions policy on the use of third-party images. Another content restrictions permission is the enable-flash permission, that allows Flash ads to be displayed. Since our framework doesn't address security threats from opaque content such as Flash (§2.1), a publisher must exercise severe caution in enabling this permission. Also <iframe> elements can be allowed via enable-iframe. However, allowing <iframe> elements can facilitate attacks such as clickjacking [17] and drive-by downloads [36].

The max-height, max-width and overflow permissions control how the ad appears on the page. If an element's size surpasses the max-width or max-height dimension and the overflow permission is set to deny, then excess content is hidden. Otherwise the excess content will overlap other parts of the page. The overflow permission is useful because some ads consume a small area when not in use, but may overlap non-ad content when engaged by the user (e.g., expanding menus). Publishers may wish to disallow expanding ads because they can overlap trusted page content.

The link-target permission controls the HTML target attribute of all <a> elements (and <form> elements, if allowed by whitelist) in mirrored content. By setting this permission, the publisher can specify that activated links or submitted forms must be directed to a new browser tab / window (if set to blank), or directed to the tab / window hosting the real page (if set to top). Whether links open in the same or new window is often agreed to between the publisher and ad network. The link-target permission can be used to protect the publisher from ad script that mistakenly creates content that does not adhere to the agreed upon link behavior.

Policy specification The publisher can annotate any HTML element of the real page with a policy attribute. The policy attribute contains a set of statements, each terminated by a semicolon. Each statement specifies the value of a particular permission in the form, *permission*: *value*;. Acceptable values for *permission* and *value* are listed in Table 1.

Permissions granted in an element's policy attribute are inherited by descendants in the HTML document hierarchy. That is, the scope of a permission $\mathcal{P}$ is the HTML subtree rooted at the element whose policy attribute grants $\mathcal{P}$.

Algorithm 1: ComputePolicy($targetElement$)

```
1  policy ← new Object();
2  WABeforeAppend ← undefined;
3  foreach element from root to targetElement do
4      if policy[ "write-access" ] = "append" then
5          policy[ "write-access" ] ← WABeforeAppend
6      statements ←Parse( element.getAttribute( "policy" ));
7      foreach stmt in statements do
8          policy ← ComposePolicies( policy, stmt );
9      if policy[ "write-access" ] ≠ "append" then
10         WABeforeAppend ← policy[ "write-access" ];
11 foreach permission in all permissions do
12     if permission is not defined in policy then
13         policy[ permission ] ← GetDefaultValue( permission );
14 return policy;
```

Policy composition Multiple policy statements may assign different values to a single permission. This can occur within a single `policy` attribute or through inheritance. We resolve the ambiguity of multiple permission values through a composition process. The composition algorithm, given in Algorithm 1, takes a target element as input and derives an assignment of values to each of the permissions listed in Table 1.

We can describe the composition algorithm intuitively as follows. The effective value for a permission is the most restrictive value specified for that permission across all composed policy statements. That is, if a permission appears in multiple statements (either within an element's `policy` attribute or in separate inherited policies), we take the intersection of all specified values for the permission. After all statements have been composed, any permissions left unspecified are set to their most restrictive values.

To enhance usability we introduced three minor exceptions to the above. First, the `max-height` and `max-width` permissions default to their *least* restrictive value (i.e., `none`). We chose this default because a definitive maximum height and width will not be satisfactory for every type of ad. It is better for each publisher to explicitly declare these values if such restrictions are desired. The policy semantics is still *default-deny*, because write permissions must first be granted before restrictions on the size of written content can have any impact. For the same reasons, our second exception defaults `link-target` permission to its least restrictive value. The third exception is we prevent inheritance of `append` write permissions. This is important as `append` specifically does not grant access to existing children of an element; thus any existing children should not inherit the `append` permission.

(a)

```
<script type="text/javascript">
    google_ad_client = "pub-...";
    google_ad_width = 728;
    google_ad_height = 90;
    google_ad_format = "728x90_as";
    google_ad_type = "text";
</script>
<script type="text/javascript"
    src="http://pagead2.googlesyndi ↵
         cation.com/pagead/show_ads.js"
></script>
```

(b)

```
  <script type="text/javascript"
    src="AdJail.js"></script>
```

Figure 4: (a) Google AdSense ad script, removed from real page. (b) Tunnel Script B, added to real page.

4.2 Real and Shadow pages

The architecture of our implementation requires changes to the original web page (real page) and creation of a corresponding shadow page as described in §3.1. The shadow page is hosted on a web server having an origin different from the real page, thus the browser's same-origin policy ensures the shadow page by default has no access to the cookies, content or other data belonging to the real page. Deploying our implementation requires a publisher to configure their DNS and web server to support the shadow page origin domain. Care must be exercised in the selection of the shadow page domain (one for each advertiser) in order to ensure that there is no reuse or overlap of domains.

To facilitate voluntary communication between the two pages, we leverage the `window.postMessage()` browser API. `postMessage()` is an inter-origin frame communication mechanism that enables two collaborating frames to share data in a controlled way, even when SOP is in effect [1].

Construction of the real page The real page is a version of the publisher's original page modified in three ways. The first modification is to remove the ad script (Figure 4a). Second, we add the tunnel script (Figure 4b) to the end of the page. The third modification to the original page is annotation of HTML elements with policies, which we discussed at length in §4.1. Only two annotations, illustrated in §3.2.3, are required for the webmail example.

The real page tunnel script has an initialization routine that first scans the real page to find all elements with policies granting the following permissions: `read-access: subtree;`, `write-access: append;`, and `write-access: subtree;`. All matching elements are converted into models (i.e., JavaScript

```
{ nodeType: "ELEMENT_NODE",
  tagName: "div", syncId: 0,
  top: y, left: x, width: w, height: h,
  attributes: {
    id: "MessageBody",
    policy: "read-access: subtree;"
  },
  children: [
    {
      nodeType: "TEXT_NODE",
      nodeValue: "Message body text here..."
    }
  ],
  computedStyle: { ... }
}
```

Figure 5: Model of `MessageBody` element (as defined in §3.2.3) sent from real page to shadow page

data structures) that will be sent to the shadow page in a later stage. Script nodes are omitted from models because we can not guarantee their semantics are preserved on the shadow page. An example model is shown in Figure 5, which models the readable Message Body `<div>` element in the webmail page (corresponding HTML given in §3.2.3).

Of the elements found in the initial scan, those with read permission are modeled by encoding (non-script) element attributes and readable child nodes into the model. The remaining elements (i.e., those having write access but no read access) are modeled as empty containers. That is, any attributes and child nodes are omitted from the model.

All elements with a policy annotation and their descendant elements are assigned a unique `syncId` attribute during initialization. The sync ID is used to match elements on the real page with their corresponding elements on the shadow page as content is kept synchronized between the two pages. As the final step of initialization, the tunnel script creates and embeds the hidden `<iframe>` element for the shadow page.

Construction of the shadow page The shadow page begins as a template web page containing only the tunnel script. As the template page is rendered, the shadow page tunnel script receives content models (described above) from the real page's tunnel script. The model data is sent as a character string in JSON [7] syntax via `postMessage()`. Once received by the shadow page, models are converted into HTML constructs using the browser's DOM interfaces. This results in a web page environment containing all the non-sensitive content and constructs of the real page, in which we will allow the ad script to execute.

To support ads (such as inline text ads) that appear or behave differently depending on where content is positioned, the shadow page is virtually sized to the dimensions of the real page, and content models are rendered in the same absolute position and size of their real page counterpart. Position and size information is depicted in Figure 5 as `top`, `left`, `width` and `height` properties. Throughout dynamic updates these attributes are kept synchronized by an approach given in §4.3.4.

Next, we install wrappers around several DOM API methods to interpose between the ad script and the DOM. Although ad script can circumvent our wrappers in Mozilla browsers by using the JavaScript `delete` operator [35], we do not rely on wrappers to enforce policies or security properties. Wrappers are used to monitor page updates and provide transparency with regard to the number of impressions generated by ads, which we discuss at length in §4.3.

Default ad zone Lastly, the ad script is embedded in the shadow page inside a container `<div>` element, which we automatically map to a corresponding `<div>` on the real page. We refer to these linked elements as the *default ad zone*. Automatic mapping is required because many ad scripts, such as Google AdSense, will not independently find and inject ads into the content imported from the real page. Rather they simply write ad content into the element containing the ad script. To support these ad scripts, the publisher indicates the default ad zone element on the real page by setting its HTML `class` attribute to include the class `AdJailDefaultZone` and ensuring the element's policy grants `subtree` write access. If the real page has no valid and unique default zone, content written to the shadow page default zone will not appear on the real page.

4.3 Synchronization

After initial rendering of the real and shadow pages in the browser, the two pages are kept synchronized by exchanging the messages listed in Table 2. We conserve the total number of generated ad impressions, using an approach given in §4.3.1. Content written by ad script to the shadow page is mirrored to the real page by a process described in §4.3.2. User interface events are forwarded from the real page to the shadow page as detailed in §4.3.3. Lastly, §4.3.4 describes how content position and style is kept synchronized on both pages as needed by some ad scripts.

4.3.1 DOM interposition

A primary goal of our approach is to conserve the number of ad impressions detected by an ad server, which we achieve using DOM interposition. Ad networks bill advertisers, and in turn pay publishers, based in part on the number of ad impressions. Impression counts are correlated to the number of requests for ad resources submitted to the web server [18]. When ad content is rendered on the real page, any external resources not available in the browser's cache will be requested, causing an impression. This may occur for several possible reasons out of our control, such as: the user disabled the cache, the ad net-

(a) Real page to shadow page:
`DispatchEvent(` *event* `)` Dispatch *event* to shadow page.
`SetScrollPos(` *x*, *y* `)` Scroll hidden shadow page to coordinates (*x*, *y*).
`SetStyle(` *syncId*, *properties* `)` Set style of shadow page element identified by *syncId* as specified in *properties*.
(b) Shadow page to real page:
`Initialize(` *step* `)` Initialize communication channel (two steps)
`InsertNode(` *syncId*, *index*, *model* `)` Insert node described by *model* as child *index* of element identified by *syncId*.
`ModifyAttribute(` *syncId*, *name*, *value* `)` Set attribute *name* to *value* on element identified by *syncid*.
`ModifyStyle(` *syncId*, *name*, *value*, *priority* `)` Set CSS property *name* to *value* and *priority* on element identified by *syncId*.
`ModifyText(` *syncId*, *index*, *data* `)` Set text content to *data* on *index* child of element identified by *syncId*.
`RemoveNode(` *syncId*, *index* `)` Remove node *index* child node of element identified by *syncId*.
`ReplaceChildren(` *syncId*, *models* `)` Replace child nodes of element identified by *syncId* with children described in *models*.
`WatchEvent(` *syncId*, *type*, *phase* `)` Register a listener for event *type* and *phase* (bubble / capture) on element identified by *syncId*.

Table 2: Synchronization messages sent between real and shadow pages via DOM `postMessage()` API.

work instructed the browser (via `Cache-Control` HTTP headers) not to cache a resource, or per-origin cache partitioning [19] is in effect.

Impressions will be generated when the ad is rendered on the real page. Therefore, when ad content is initially rendered on the shadow page, we must prevent the browser from submitting HTTP requests for external resources, as that would cause superfluous impressions. Our implementation supports conserving impression counts for the following elements in our whitelist: `<img>`, `<iframe>` and `<object>` (Flash). Additionally we conserve impression counts for background image CSS properties in our whitelist: `background`, `background- image`, `list-style` and `list-style-image`.

To prevent ad impressions on the shadow page, we interpose on the common interfaces ad scripts use to create content. First, we hook DOM object prototype interfaces [25] to prevent ad scripts from setting URI attributes. For instance, we interpose on the `src` property of `HTMLImageElement` objects, and `getAttribute()` and `setAttribute()` DOM methods. We also hook other interfaces that access URI attributes, such as `document.write()`, `document.writeln()`, and *`element`*`.innerHTML`, to increase completeness and transparency of the interposition.

When ad script writes a URI attribute using one of these APIs, we substitute the real URI value with a placeholder value. For `write()`, `writeln()`, and `innerHTML`, this substitution requires a character search and replace in HTML source code. Our current implementation of this operation makes use of regular expression based textual transformation, which works well in practice, but may not be very precise under all circumstances. As the purpose of this substitution is to conserve ad impressions, a loss in precision here may affect compatibility with ads, but not security. If more precision is required, works on in-browser source-to-source HTML transformation [14, 34] can be leveraged, at the cost of additional overhead.

One exception we make to the above scheme is for `<script>` elements. Our interposition does not block the setting of `src` attributes for scripts, because our goal is to enable ad scripts to execute in the shadow page. Thus scripts are the only source of ad impressions from the shadow page. Since our policy enforcement mechanism prevents ad scripts in the real page, each script is created only once, thereby conserving the number of ad impressions.

4.3.2 Content mirroring

We mirror ad content from the shadow page to the real page using a 5-step process: (1) monitoring the shadow page for modifications by the ad script, (2) modeling the detected modifications, (3) sending the model to the real page, (4) enforcing policies on the model, and (5) modifying the real page to reflect the model.

1. Monitoring the shadow page for modifications We monitor the shadow page for dynamic modifications using DOM interposition logic (introduced in §4.3.1). In addition to APIs that affect element attributes, we also hook APIs that modify the document, such as *`element`*`.appendChild()`. Whenever ad script attaches a new DOM node using `appendChild()`, our monitoring code is invoked before the actual modification takes place. Alternatively, *DOM mutation events* [51] can be leveraged to perform the same monitoring function with lower complexity than DOM interposition. However, Internet Explorer does not yet support mutation events, which would result in decreased compatibility.

2. Modeling the detected modifications When modifications to the shadow page are detected, we encode those

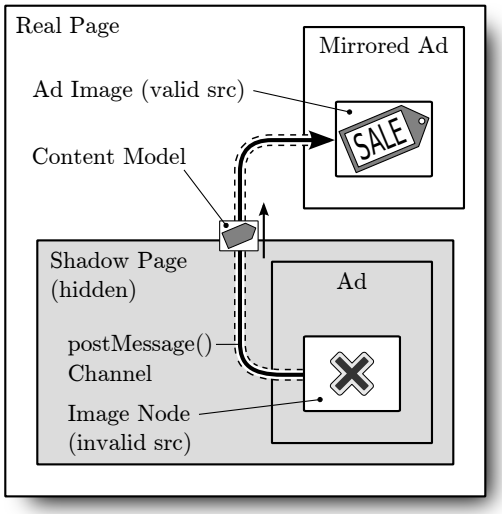

Figure 6: Rendering an ad image only on the real page so that just one impression is generated.

changes using the same model format described in §4.2 and depicted in Figure 5. However, when we find content that was substituted by our interposition (ref. §4.3.1), we model the ad script's *intended* content instead of the substituted content. Models are passed to the real page, where the modifications will be reflected to the extent allowed by policies.

3. Sending models to the real page The process of sending a model of an image element is depicted in Figure 6. In the shadow page, we serialize the model data structure to a JSON string. We send the serialized model from the shadow page to the real page using the `InsertNode()` message from Table 2b. (Other types of modifications use the additional `postMessage()` notifications listed in Table 2b.) On the receiving end (i.e., the real page), we deserialize the string to recover the model data structure.

4. Enforcing policies on the models Our policy enforcement code in the real page receives the model from the shadow page. The model is then checked for any content that violates the real page policy annotations. We trim all policy-violating content from the model. For instance, if the model describes an image to be added to the page where the `enable-images` permission is denied, then we remove the image from the model. If the model describes an ad that is 1000 pixels wide and the policy only allows the ad to be 600 px, we allow the ad but restrict its maximum width to 600 px.

5. Modifying the real page to reflect the modeled changes Finally we merge the changes represented by the model into the real page. We create or modify constructs using DOM APIs, such as `document.createElement()` and *element*`.setAttribute()`. To ensure scripts are not injected into the real page during this process, we leverage the techniques we developed in BLUEPRINT [46] to enforce a *no-script* policy over all merged changes. This entails protecting several script injection vectors, including `<script>` elements, event handler attributes, `javascript:` URI schemes, CSS expressions, and more.

Mirroring ad content on the real page has the side-effect of modifying the real page script execution environment. For instance, elements such as `<input name="query" ...>` can pollute the namespace by creating properties such as `document.elements.query`. A straightforward solution to this problem is disallowing `name` and `id` attributes on mirrored ad content; however, this may reduce compatibility with some ads.

4.3.3 Event forwarding

To prevent code injection attacks during content mirroring, our whitelist intentionally omits event handlers such as `onclick` and `onmouseover` that have been attached to ad content. In order to preserve event handler functionality in spite of this restriction, we perform event forwarding.

Event forwarding leverages our DOM interposition framework. We interpose on script operations used to register event handlers such as handler attributes and object properties (e.g., `onclick`, `onload`, etc.), using the same mechanism used for URI attributes and properties described in §4.3.1. Additionally, browser-specific APIs such as *element*`.addEventListener()` and *element*`.attachEvent()` are detected and interposed on when present.

When ad script uses any of these APIs to register an event handler on an element, and that element is also mirrored on the real page, we register our own handler for the same event on the mirrored element. Event handlers are registered on the real page when specified in content models (`InsertNode()` and `ReplaceChildren()` messages), or by sending the `WatchEvent()` message of Table 2. Whenever the event occurs on the real page, our handler is invoked and sends details of the event to the shadow page using the `DispatchEvent()` message (indicated by path ③ on Figure 3). On the shadow page we establish the appropriate JavaScript scope, then dispatch the event to the target element. This in turn invokes the ad script's original event handler. Effects caused by the ad script's handler are detected and mirrored back to the real page using the mechanism described in §4.3.2.

Ad clicks Unlike other user interface events, we do not forward click events on `<a>` (link) elements. Instead we click (i.e., activate) links on the real page, subject to enforcement of the `link-target` permission. This has the effect of bypassing any click event handlers the ad script may have registered on the activated link. Therefore there can be a compatibility trade-off in enforcing the

`link-target` permission if the ad script depends on such event handlers.

4.3.4 Position and style synchronization

Some ads mimic the appearance of a pop-up window by temporarily overlaying parts of the web page. Although the pop-up window can appear at variable locations on the page, typically it is positioned such that it is visible (given the portion of the page that is scrolled into view) and relative to some other content (such as a contextual keyword). The ad script contains logic to compute the pop-up location based on the above criteria. However, if content appears at a different location on the real page than it does on the shadow page, the pop-up will be positioned incorrectly when mirrored. For this reason we support synchronizing the visual aspect of both real and shadow pages, even though the shadow page remains hidden.

First, we keep the window sizes of each page synchronized by setting the shadow page size to 100% of the real page size. Second, we sync the scroll position of both pages by registering an event handler for the real page's `onscroll` event. Whenever the event fires, we send a `SetScrollPos()` message to the shadow page. Our code running in the shadow page receives this message and adjusts the shadow page vertical and horizontal scroll offsets to match the real page.

Next we have to ensure content on the shadow page occupies the same location and extent as the corresponding content on the real page. For example, consider the inline text ad (Figure 1, #3), which highlights keywords and makes a pop-up appear near a keyword when the user's mouse hovers over it. The precise location of the keyword depends on many things, such as the absolute coordinates of the element containing the text, height and width of the container element, font size of the text, dimensions / layout of other content in the container, and more. We synchronize these details by sending the absolute position, size and *computed style* of each mirrored element to the shadow page via the `SetStyle()` message. On the shadow page we apply these properties to content elements, while keeping record that these are not "authentic" properties that should be synchronized back to the real page during any future content mirroring operations.

This strategy works very well in practice but is not perfect. For instance, there may be text in the real page that flows around an image. If the policy in effect for the text content allows read access, and the image is not readable, then the image will not appear on the shadow page and thus the text will not flow in the same way. To resolve issues due to the layout becoming out of sync, the publisher can either make the image readable or customize the shadow page to more accurately reflect the real page.

5 Evaluation

We evaluated AdJail to assess performance in three major areas. In §5.1 we investigate the compatibility of our architecture with six popular ad networks, each of which serve a variety of ads. The security of our approach is tested in §5.2. We then measure ad display latencies in §5.3. Although many ad networks exist which were not tested, we believe the relatively small sample we evaluated offer good insights into the compatibility and performance of AdJail.

5.1 Compatibility

To evaluate how well AdJail works with existing ad scripts, we tested it on six popular ad networks: Yahoo! Network, Google AdSense, Microsoft Media Network, Federated Media Publishing, AdBrite and Clicksor. The first four used banner ads, while the latter two employed more complicated inline text ads. Yahoo!, Google and Microsoft were three of the top ten ad networks in terms of U.S. market reach in April 2009. With a total audience size of $192.8\,\text{million}$, Yahoo! reached 86.6% of the market, Google reached 85.3%, and Microsoft reached 72.4% [3].

Federated Media, AdBrite and Clicksor rank lower in terms of U.S. market reach (e.g., AdBrite ranked #21 with a reach of 47.2%), but were chosen as they represent the small publisher market and demonstrate unique functionality. They are not as pervasive, therefore they are more likely to exhibit compatibility problems and less tested features. In our experiments we focused on the following observations: whether the ad functioned correctly, the minimum permissions required to support the ad, and whether click and impression counts were affected by our approach.

Our prototype AdJail implementation is a sufficient proof-of-concept to demonstrate the feasibility of our approach. The prototype is designed and tested to work on recent releases of the Chrome, Firefox, Internet Explorer, Opera and Safari web browsers. It does not yet have the level of refinement that would be present in a production system, which exposes some compatibility limitations we describe below.

Correct functionality To evaluate correct functionality we embedded ad scripts from each ad network in a series of AdJail test pages, then compared the user experience to the same ad scripts when used without sandboxing. The four banner ad scripts (Yahoo!, Google, Microsoft and Federated) all made use of the default ad zone feature. In this experiment we observed two main types of ad banner: animated image and Flash.

All of the banner ads rendered on the real page without any noticeable differences from rendering the ad without AdJail. Interacting with Flash ads via the mouse and clicking on banners worked exactly the same as the

non-sandboxed ads. One minor issue we are aware of is that the contextual targeting approach used by Google AdSense does not work with our current implementation. This is because AdSense performs contextual targeting on the server, using an offline cached copy of the publisher's page. This limitation can be overcome by providing pre-computed shadow pages to ad networks who perform server-side contextual targeting, like AdSense.

For each of the inline text ad scripts (AdBrite and Clicksor), we annotated a news article with a full read and write access policy. The ad scripts identified keywords in the article and transformed them into interactive ads that "pop up" when the user hovers the mouse cursor over a keyword. This allowed us to evaluate the intricate synchronization capabilities of our architecture, such as ad script modifying existing page content and event forwarding. The pop-ups consisted of a decorative window border around the actual advertisement. AdBrite worked well in this experiment; its ads were simply `<iframe>`s wrapped by the decorative border. Clicksor also worked without any noticeable differences.

Minimum permissions For each tested ad network, we enabled the strictest set of permissions that would permit ads to function without impairment. These permissions are summarized in Table 3. To arrive at the set of permissions, we started with the base read and write access needed by the ad. We then enabled support in the content whitelist based on the needs of the ad. Finally, for fixed-size banner ads we set the maximum width and height policies.

Google AdSense was configured to serve text ads, so we were hoping to confine it with a strict text-only policy. Unfortunately the text ads were contained in an `<iframe>`, thus we had to set the `enable-iframe` permission.

AdBrite and Clicksor needed append write permission on the `<body>` element to create their pop-ups. Whitelist customization was also required for the pop-ups, as they contained custom HTML elements to prevent inheritance of publishers' CSS formatting rules [4]. AdBrite was easier to support as we only had to whitelist their custom `<ispan>` element. Clicksor used a randomly generated element tag name consisting of the word "span" followed by digits (e.g., `<span40110>`). To accommodate Clicksor we modified the whitelist to accept element tag names that matched the JavaScript regular expression `/^span[0-9]{5,7}$/`. Also we note that Clicksor was the only ad network to require `<form>` and `<input>` elements in its whitelist.

Click and impression counts To measure the number of clicks and impressions caused by ads, we configured our browser to route all traffic through a web proxy running the Squid proxy software. We rendered each ad script with and without sandboxing, and clicked on the displayed ads in each case. For this experiment, the web page hosting the ad script was completely blank except for a single paragraph of text, which was used for rendering inline text ads and contextual ad targeting.

A given ad script may show a different ad each time it is rendered. To ensure consistency in our evaluation, multiple renderings were sometimes performed for an ad network to ensure we clicked on the same advertisement with and without sandboxing. In between renderings, we cleared the browser's cache to ensure proxy access patterns were not affected by prior tests.

After performing the experiment, we analyzed the proxy's access logs. We discarded all log entries that referred back to our server hosting the test pages and ADJAIL source code. Comparing the remaining log entries, we did not find any differences in the HTTP requests generated by sandboxed versus non-sandboxed ads. Thus we conclude that in our experiment, ads using our sandbox environment did not impose any additional impressions or generate any additional clicks, thereby preserving traffic patterns crucial to the web advertising revenue model.

5.2 Security

To evaluate the security provided by ADJAIL we installed the RoundCube webmail v0.3.1 software on our web server. We integrated two ad network scripts on the main webmail interface: one ad script was included directly on the page, and the other was embedded using ADJAIL. A single trial consisted of replacing each of the two ad scripts with a malicious script designed to perform one specific attack or policy violation. We then observed if the malicious script functioned correctly in the non-sandboxed location, and whether the attack was prevented in the sandboxed location. Several trials were conducted to assess different attack vectors, and to determine the least restrictive policy required to defend each vector.

Our experiments were designed to support our claims in §1 of strong defense against several potent attack vectors to which ad publishers are routinely exposed. However, we did not evaluate the threats discussed in §2 that are beyond the scope of our current work: drive-by downloads, Flash exploits, privacy attacks, covert channels, and frame busting.

Results of the security evaluation are included on the right side in Table 3. With appropriate policies in effect, ADJAIL blocked all of the in-scope threats. We note that for each ad, write access was allowed for the subtree rooted at the `<div>` element designated for ad content. However, every ad policy denied write access (the default setting) for the rest of the document. A degree of leniency is required in our policies for compatibility with existing ads, which opens the door to some of the secondary attacks. However, every ad network we tested was protected from our primary threats: confidential data leaks and content integrity violations.

Ad Network	Element	Computed Policy (Annotated policy in bold)								Attack resistance						
		read access	*write access*	*enable images*	*enable iframe*	*enable flash*	*max width*	*max height*	*over-flow*	EX	CB	IV	CJ	UI	AP	OA
AdBrite	`<body>`	none	**append**	**allow**	**allow**	deny	none	none	deny	✓	✓	✓				
	Article `<div>`	**subtree**	**subtree**	deny	**deny**	deny	none	none	deny	✓	✓					
Clicksor	`<body>`	none	**append**	**allow**	deny	deny	none	none	deny	✓	✓	✓	✓			
	Article `<div>`	**subtree**	**subtree**	deny	deny	deny	none	none	deny	✓	✓		✓			
Federated Media	Ad `<div>`	none	**subtree**	**allow**	**allow**	**allow**	**90px**	**728px**	deny	✓	✓	✓			✓	✓
	Rest of page	none	none	deny	deny	deny	none	none	deny	✓	✓	✓	✓	✓	✓	✓
Google	Ad `<div>`	none	**subtree**	deny	**allow**	deny	**600px**	**160px**	deny	✓	✓	✓			✓	✓
	Rest of page	none	none	deny	deny	deny	none	none	deny	✓	✓	✓	✓	✓	✓	✓
Microsoft Media	Ad `<div>`	none	**subtree**	deny	**allow**	**allow**	**300px**	**250px**	deny	✓	✓	✓			✓	✓
	Rest of page	none	none	deny	deny	deny	none	none	deny	✓	✓	✓	✓	✓	✓	✓
Yahoo!	Ad `<div>`	none	**subtree**	deny	deny	**allow**	**90px**	**780px**	deny	✓	✓	✓	✓		✓	✓
	Rest of page	none	none	deny	deny	deny	none	none	deny	✓	✓	✓	✓	✓	✓	✓

Table 3: Policy annotations required to support several popular ad networks, and attacks prevented in policy enforcement regions. Attacks prevented are: EX: Execute arbitrary code in context of real page (non-XSS), CB: Data confidentiality breach, IV: Content integrity violation, CJ: Clickjacking, UI: UI spoofing, AP: Arbitrary ad position, OA: Oversized ad. Default `link-target` policy used for all.

Below we briefly describe our objectives and methodology for testing each attack.

Execute arbitrary code in context of real page In this attack we attempted to break out of the sandbox, by causing the browser to execute ad script code in context of the real page. This attack is critical because, if successful, malicious code can disable all policy enforcement logic in the real page and subsequently mount any of the other attacks. Specifically excluded from this vector is code injection by reflected, DOM0, and stored XSS attacks, which the web application can defend by other means.

We attempted to inject script code in the real page via DOM traversal, but this was blocked by the browser's SOP policy. Next, we evaluated 7 different real-world attacks sourced from the XSS Cheat Sheet [16]. Each attack demonstrated a unique code injection vector, such as embedded `script` element, event handler, `javascript:` URI, CSS expression, and more. These code injection attempts were blocked by enforcing a *no-script* policy on content models when constructing the mirrored ad in the real page, using the technique we developed in prior work [46].

To evaluate our defense against Flash-based script injection attacks, we created a Flash application that uses the `ExternalInterface` API to extract confidential data from the DOM. Flash regulates access to this API via the `allowScriptAccess` attribute of `<object>` elements, and `value` attribute of `<param>` elements when the `name` attribute is set to `allowScriptAccess`. Without ADJAIL, the ad network's script can create Flash objects on the real page with `allowScriptAccess` set to `always`. This setting permits Flash ActionScript code to fully access the real page's JavaScript environment, including sensitive page content via the DOM. Our defense blocks this attack vector by forcing the `allowScriptAccess` attribute to `never` on all `<object>` elements and relevant `<param>` elements. This action effectively disables the Flash `ExternalInterface` API.

All script injection attacks were prevented even with the most permissive policy that can be written using our policy language. Thus the script injection vector is defended for every possible policy configuration.

Confidential information leak For this attack we retrieved two items of confidential data from the real page: the user's session cookie and list of email contacts. Due to SOP restrictions, the sandboxed attack could not access the information by DOM traversal. (We note DOM traversal is also an ineffective strategy for all remaining evaluated attacks.) The only way the attack could access confidential data was when the data was given a policy granting full read access.

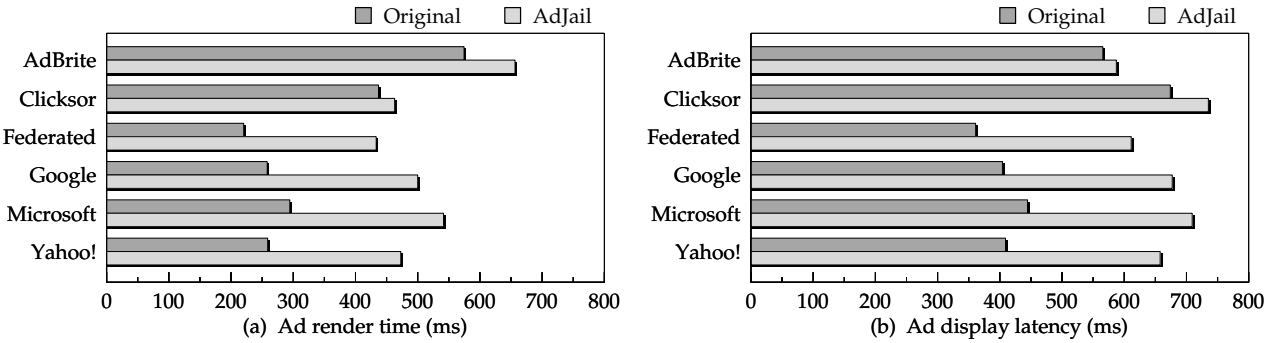

Figure 7: Rendering latencies: (a) time spent loading the ad, and (b) time from start of page load until ad appears.

Content integrity violation This attack tampers with trusted content on the real page: the user's email message headers. Specifically the attack makes all messages appear to be sent by prominent government officials. The sandboxed attack was unsuccessful except when the message headers were given a policy with full write access.

Clickjacking The clickjacking attack attempts to entice the user to unknowingly click on an `<iframe>` element. The attack script is based on detailed technical analysis of the vector [17, 54]. With a policy that disallows `<iframe>` elements, the sandboxed attack was unsuccessful because the policy prevents any `<iframe>` on the (hidden) shadow page from being brought up to the real page where the user can click it. Since any `<iframe>` embedded by the ad is unclickable to the end user, typical tricks to mask the clickjacking attack (e.g., hiding the `<iframe>` using transparency) are not a factor.

User interface spoofing We made an ad appear identical to trusted webmail user interface components in an attempt to lure users into interacting with the ad (i.e., an *interface spoofing* attack [26]). This attack was defeated by denying images, `<iframe>`s and Flash, and further constraining the ad with policies that disallow the ad from overlapping other parts of the trusted interface. Since the ad can still make use of textual elements, we note there exists a very small likelihood for an attacker to succeed through very nuanced UI spoofing attack using very small (single pixel) elements or text, such that images can be rendered in HTML one pixel at a time. Mitigating this threat may require advanced analysis of ad content or restricting the color palette available to ads.

Arbitrary ad position We made an ad appear on the real page outside of its write-accessible container element. This type of violation can be performed by setting an ad content display position that is outside the bounds of its container. With a policy that denies overflow, violations due to out-of-bounds display positioning are blocked. Position policies can also be violated by a *node splitting* attack, which may only succeed when there is no mechanism to provide hypertext markup isolation [41, 45]. Our content mirroring approach provides the necessary isolation by default to prevent node splitting attacks.

Oversized ad We made an ad larger than the publisher's expected ad size. The size violation was blocked by configuring a policy to limit the maximum height and width, and disallowing overflow.

5.3 Rendering overhead

To measure ad rendering latencies incurred by our policy enforcement mechanism, we placed each ad script on a typical blog page instrumented with benchmarking code. There were a total of 12 instances of the blog page: for each of the six ad networks evaluated in §5.1, one version of the blog page used the original ad, and a second version used ADJAIL to enforce the policies in Table 3. As the blog page is rendered, the ad script executes and scans for contextual data, requests a relevant ad from the ad network based on this data, and finally renders the ad. This experiment reflects the typical delays a end-user would experience when browsing publisher pages that integrate ADJAIL.

The test pages were rendered in Firefox v3.6.3 on an AMD Phenom X4 940 (3.0 GHz) workstation with 7.5 GB RAM. To resemble a typical browsing environment, the browser cache was enabled during the experiment. Each test page includes a link to our ADJAIL implementation source code (102 kB of JavaScript), which was cached by the web browser. The code is not optimized for space and contains much debug code. The memory overhead required by ADJAIL was reasonably consistent across ad networks, averaging 5.52% or roughly 3.06 MB.

Results of this experiment are shown in Figure 7. First we measured the time taken to render only the ad (Figure 7a). For AdBrite and Clicksor (inline text ads), this measurement consists of the time between the user triggering an ad pop-up and appearance of the pop-up. Although we do not separately report the latency incurred by forwarding events to the shadow page (ref. §4.3.3), this overhead is included in Figure 7. For this experiment, we stopped the benchmark after the ad's `<iframe>`

or `<object> onload` event was triggered, signaling the ad was complete. Without sandboxing, ads rendered in 374 ms on average. With ADJAIL, ad rendering averaged 532 ms, an additional latency of 158 ms.

To better understand the impact of ad rendering latency, we measured the time between when the page started loading until the ad completed rendering (Figure 7b). This is an important benchmark for ads, as many ad networks use a content distribution network (CDN) to improve performance in this regard [47]. For AdBrite, and Clicksor, we measured the time until inline text links finished rendering, although no ads are visible until the user triggers a pop-up. Without sandboxing, ads appear in 489 ms on average after the page begins to load. With ADJAIL, an additional 163 ms delay was incurred on average.

Optimizing performance is an important area for future work. A straightforward way to improve performance will be to optimize our prototype implementation. More significant gains may be achieved by adapting our approach to support pre-computing policies and shadow pages. It may be feasible to integrate caching of policies and shadow pages into web application templates and frameworks, to allow better performance without raising the publisher effort required to deploy ADJAIL.

6 Conclusion

In this paper, we presented ADJAIL, a solution for the problem of confinement of third-party advertisements to prevents attacks on confidentiality and integrity. A key benefit of ADJAIL is compatibility with the existing web usage models, requiring no changes to ad networks or browsers employed by end users. Our approach offers publishers a promising near term solution until web standards support for confinement of advertisements evolves to offer solutions agreeable to all parties.

Acknowledgements

We thank Rohini Krishnamurthi for many insightful discussions that helped to shape principal ideas of this work. Our sincere thanks to our shepherd Lucas Ballard, and the anonymous reviewers for their helpful and thorough feedback on drafts. This work was partially supported by National Science Foundation grants CNS-0716584, CNS-0551660, CNS-0845894 and CNS-0917229. The first author was additionally supported in part by a fellowship from the Armed Forces Communications and Electronics Association.

References

[1] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *17th USENIX Security Symposium*, San Jose, CA, USA, July 2008.

[2] Click Quality Team. How fictitious clicks occur in third-party click fraud audit reports. Technical report, Google, Inc., August 2006.

[3] comScore. April 2009 U.S. ranking of top 25 ad networks. `http://www.comscore.com/Press_Events/Press_Releases/2009/5/Top_25_US_Ad_Networks`, May 2009. Retrieved 19 Nov. 2009.

[4] Sean Conaty. Introducing the `<ispan>`. `http://nerdcereal.com/introducing-the-ispan/`, January 2008. Retrieved 1 Jun. 2010.

[5] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *19th International World Wide Web Conference*, Raleigh, NC, USA, April 2010.

[6] Douglas Crockford. ADsafe. `http://www.adsafe.org/`. Retrieved 1 Jun. 2010.

[7] Douglas Crockford. The application/json media type for JavaScript object notation (JSON). `http://tools.ietf.org/html/rfc4627`, July 2006. RFC 4627.

[8] Úlfar Erlingsson, V. Benjamin Livshits, and Yinglian Xie. End-to-end web application security. In *11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[9] Facebook Developers. Facebook JavaScript. `http://wiki.developers.facebook.com/index.php/FBJS`. Retrieved 8 Apr. 2010.

[10] Adrienne Felt, Pieter Hooimeijer, David Evans, and Westley Weimer. Talking to strangers without taking their candy: Isolating proxied content. In *1st International Workshop on Social Network Systems*, Glasgow, Scotland, April 2008.

[11] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *7th ACM Conference on Computer and Communications Security*, Athens, Greece, November 2000.

[12] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing capability leaks in secure JavaScript subsets. In *17th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, March 2010.

[13] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious Flash advertisements. In *25th Annual Computer Security Applications Conference*, Honolulu, HI, USA, December 2009.

[14] Google Caja. A source-to-source translator for securing JavaScript-based web content. `http://code.google.com/p/google-caja/`. Retrieved 1 Jun. 2010.

[15] Saikat Guha, Bin Cheng, Alexy Reznichenko, Hamed Haddadi, and Paul Francis. Privad: Rearchitecting online advertising for privacy. Technical Report MPI-SWS-2009-004, Max Planck Institute for Software Systems, Kaiserslautern-Saarbruecken, Germany, October 2009.

[16] Robert Hansen. XSS (cross site scripting) cheat sheet esp: for filter evasion. `http://ha.ckers.org/xss.html`, 2008. Retrieved 8 Apr. 2010.

[17] Robert Hansen and Jeremiah Grossman. Clickjacking. `http://www.sectheory.com/clickjacking.htm`, September 2008. Whitepaper.

[18] Interactive Advertising Bureau. Interactive audience measurement and advertising campaign reporting and audit guidelines. Global Version 6.0b, IAB, September 2004.

[19] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *15th International World Wide Web Conference*, Edinburgh, Scotland, May 2006.

[20] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for Web mashups. In *16th International World Wide Web Conference*, Banff, AB, Canada, May 2007.

[21] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *16th International World Wide Web Conference*, Banff, AB, Canada, May 2007.

[22] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. JavaScript instrumentation in practice. In *6th Asian Symposium on Programming Languages and Systems*, Bangalore, India, December 2008.

[23] Jeremy Kirk. Ad exploits Internet Explorer vulnerability to explose millions to adware. `http://www.infoworld.com/print/23520`, July 2006. Retrieved 23 Apr. 2010.

[24] Mary Landesman. ScanSafe: Weekend run of malvertisements. `http://blog.scansafe.com/journal/2009/9/24/weekend-run-of-malvertisements.html`, September 2009. Retrieved 23 Apr. 2010.

[25] Travis Leithead. Document Object Model prototypes, Part 1: Introduction. `http://msdn.microsoft.com/en-us/library/dd282900%28VS.85%29.aspx`, November 2008. Microsoft Corporation. Retrieved 22 May 2010.

[26] Elias Levy and Iván Arce. Interface illusions. *IEEE Security and Privacy*, 2:66–69, 2004.

[27] Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *19th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, December 2003. IEEE Computer Society.

[28] V. Benjamin Livshits and Salvatore Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[29] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Language-based isolation of untrusted JavaScript. In *22nd IEEE Computer Security Foundations Symposium*, Port Jefferson, NY, USA, July 2009.

[30] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Run-time enforcement of secure JavaScript subsets. In *3rd Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2009.

[31] Dahlia Malkhi and Michael K. Reiter. Secure execution of Java applets using a remote playground. *IEEE Transactions on Software Engineering*, 26(12):1197–1209, December 2000.

[32] Gervase Markham. Content restrictions. `http://www.gerv.net/security/content-restrictions/`, March 2007.

[33] Leo A. Meyerovich and V. Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2010.

[34] Microsoft Live Labs. Web Sandbox. `http://websandbox.livelabs.com`. Retrieved 1 Jun. 2010.

[35] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security*, Sydney, Australia, March 2009.

[36] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs point to us. In *17th USENIX Security Symposium*, San Jose, CA, USA, July 2008.

[37] C. Reis, J. Dunagan, Helen J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA, November 2006.

[38] Matthew Rogers. Facebook's response to uproar over ads. `http://endofweb.co.uk/2009/07/facebook_ads_2/`, July 2009. Retrieved 6 Apr. 2010.

[39] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: A study of clickjacking vulnerabilities on popular sites. In *4th Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2010.

[40] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *4th ACM Symposium on Operating Systems Principles*, Yorktown Heights, NY, USA, October 1973.

[41] Prateek Saxena, Dawn Song, and Yacin Nadji. Document structure integrity: A robust basis for cross-site scripting defense. In *16th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, February 2009.

[42] Barry Schnitt. Debunking rumors about advertising and photos. `http://blog.facebook.com/blog.php?post=110636457130`, November 2009. Retrieved 6 Apr. 2010.

[43] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the Web with content security policy. In *19th International World Wide Web Conference*, Raleigh, NC, USA, April 2010.

[44] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An efficient approach for realizing safe execution environments. In *12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2005.

[45] Mike Ter Louw, Prithvi Bisht, and V. N. Venkatakrishnan. Analysis of hypertext isolation techniques for cross-site scripting prevention. In *2nd Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2008.

[46] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009.

[47] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *17th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, March 2010.

[48] Ashlee Vance. Times Web ads show security breach. `http://www.nytimes.com/2009/09/15/technology/internet/15adco.html`, September 2009. NY Times. Retrieved 1 Jun. 2010.

[49] Yi-Min Wang, Doug Beck, Xuxian Jiang, and Roussi Roussev. Automated Web patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *13th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2006.

[50] Wikipedia contributors. Same origin policy. `http://en.wikipedia.org/w/index.php?title=Same_origin_policy&oldid=190222964`, February 2008.

[51] World Wide Web Consortium. Document object model (DOM) level 2 events specification. `http://www.w3.org/TR/DOM-Level-2-Events`, November 2000.

[52] Yankee Group. Yankee Group forecasts US online advertising market to reach $50 billion by 2011. `http://www.yankeegroup.com/pressReleaseDetail.do?actionType=getDetailPressRelease&ID=1805`, January 2008. Retrieved 6 Apr. 2010.

[53] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *34th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Nice, France, January 2007.

[54] Michal Zalewski. Browser security handbook. `http://code.google.com/p/browsersec/wiki/Main`, 2009. Retrieved 26 Jan. 2010.