AdSentry: Comprehensive and Flexible Confinement of JavaScript-based Advertisements

Xinshu Dong[†], Minh Tran[‡], Zhenkai Liang[†], Xuxian Jiang[‡]

[†] Department of Computer Science
National University of Singapore
{xdong, liangzk}@comp.nus.edu.sg

[‡] Department of Computer Science
North Carolina State University
{mqtran, xuxian_jiang}@ncsu.edu

ABSTRACT

Internet advertising is one of the most popular online business models. JavaScript-based advertisements (ads) are often directly embedded in a web publisher's page to display ads relevant to users (e.g., by checking the user's browser environment and page content). However, as third-party code, the ads pose a significant threat to user privacy. Worse, malicious ads can exploit browser vulnerabilities to compromise users' machines and install malware. To protect users from these threats, we propose AdSentry, a comprehensive confinement solution for JavaScript-based advertisements. The crux of our approach is to use a shadow JavaScript engine to sandbox untrusted ads. In addition, AdSentry enables flexible regulation on ad script behaviors by completely mediating its access to the web page (including its DOM) without limiting the JavaScript functionality exposed to the ads. Our solution allows both web publishers and end users to specify access control policies to confine ads' behaviors. We have implemented a proof-of-concept prototype of AdSentry that transparently supports the Mozilla Firefox browser. Our experiments with a number of ads-related attacks successfully demonstrate its practicality and effectiveness. The performance measurement indicates that our system incurs a small performance overhead.

1. INTRODUCTION

Internet advertising is one of the most popular business models of today's Internet companies. For example, more than 96% of Google's revenue is from Internet advertising [15]. In Internet advertising, web site owners or web publishers include advertisements (or "ads") from advertisers in their pages, and get paid by advertisers when users view and click on these ads.

To increase the likelihood for users to click on the ads, advertisers commonly use (JavaScript) code in ads to check a user's browser environment to select advertisements that are believed to be more attractive to the users. As third-party code, these ads unfortunately pose great security threats to both web applications and the underlying operating systems. For example, such ads require close integration with the displayed page contents, which may leak users' private data [28] and break web applications' integrity. Worse,

malicious ads can further exploit software vulnerabilities in web browsers to launch drive-by downloads and surreptitiously install malware on users' machines. A recent research shows that "about 1.3 million malicious ads are being viewed online everyday, most pushing drive-by downloads and fake security software" [33].

To mitigate the threats from untrusted ads, a number of solutions have been recently proposed. They address the threats to user privacy and web application integrity by sandboxing JavaScript ads through functionality restriction or isolation [2, 4, 10, 13, 17, 19, 23, 26, 27, 30, 35, 40, 45, 49]. However, they cannot block ads from triggering drive-by downloads, which have been "persistently" plaguing online users as one of the main attack mechanisms. In addition, these solutions are not flexible in controlling behaviors of JavaScript advertisements: the allowed access of an ad must be decided before it starts to run. In the face of these limitations, there is a need for an integrated solution that can not only flexibly regulate the ad access to various web contents, but also effectively block drive-by downloads from malicious ads.

In this paper, we present the design, implementation and evaluation of AdSentry, a comprehensive and flexible isolation framework to confine JavaScript-based advertisements. Instead of supporting only a subset of JavaScript functionality or isolating the ad execution through significant changes to web pages, AdSentry provides a *shadow JavaScript engine* for untrusted ad execution. The purpose of having a shadow JavaScript engine is to ensure that the ad will not affect the host web content without proper control, thus protecting user privacy and the integrity of web applications. More importantly, it provides the control in a transparent manner and still exposes the full spectrum of JavaScript functionality to the untrusted ads. Meanwhile, to block possible drive-by downloads and preserve the integrity of the host system, the shadow JavaScript engine is strictly sandboxed.

By design, AdSentry effectively mediates all accesses made by untrusted ads to the web application. We stress that the shadow JavaScript engine (for the ad execution) by default cannot access the original page DOM (Document Object Model). To accommodate legitimate accesses by ads to some part of the page content, AdSentry transparently interposes related DOM accesses from the ads. For every such access, AdSentry checks its legitimacy (according to a given access control policy) and, if benign, redirects it to the original page DOM to substantiate the access. Our framework is flexible in allowing both web publishers and end users to specify or customize the access control policies for ads, as well as allowing dynamically changing policy after an ad starts to execute.

We have implemented a proof-of-concept AdSentry prototype. The shadow JavaScript engine implementation is based on the open-source Mozilla SpiderMonkey. Its execution is strictly sandboxed with Native Client [48], which has demonstrated its effectiveness

in confining third-party code with high efficiency and reliability. Our development experience further indicates that AdSentry is a generic framework that can be conveniently implemented as a regular browser extension without requiring the modification of the browser code. Our evaluation results with a number of ad-related exploits show that AdSentry is effective in successfully blocking all of them. The performance evaluation shows that the protection is achieved with a low overhead.

Contributions.

We identified the critical need for a confinement solution to prevent ads from threatening the privacy and integrity of user data as well as the integrity of users' computing systems. To summarize, this paper makes the following contributions:

- AdSentry provides a comprehensive isolation framework to confine untrusted ads on web pages. Its comprehensiveness is achieved by not only regulating the ad access to the original web application (or web contents), but also effectively confining the ad execution in a strongly sandboxed environment to block possible drive-by downloads.

- Our solution preserves the original execution environment for third-party scripts. Unless otherwise specified by access control policies, AdSentry does not alter the execution order of different scripts on the page even after certain ad scripts are sandboxed by our solution. Sandboxed ads scripts also have full access to global JavaScript objects created or overwritten by other scripts outside the sandbox, if allowed by access control policies.

- Our solution allows for flexible mediation of each DOM access from untrusted ads. It is also flexible in allowing both web publishers and end users to specify access control policies for ads. We highlight that it is important to empower users with full control as they can choose how to protect the viewed web pages from ads (according to their own requirements and running environments) and remain confident in protecting the integrity of their host systems.

- We have implemented a prototype that transparently supports modern Mozilla Firefox browsers. Our prototyping and evaluation results with real-world examples demonstrate its practicality and effectiveness.

The rest of this paper is organized as follows: Section 2 provides an overview on Internet advertising. Sections 3 and 4 present our system design and implementation. Section 5 presents detailed evaluation results. Section 6 discusses limitations of our approach and suggests future improvement. Finally, Section 7 describes related work, and Section 8 concludes the paper.

2. PROBLEM OVERVIEW

In Internet advertising, advertisers pay web publishers directly to display their ads on these web sites, or more often, pay advertising networks to get their ads displayed on popular sites, easily reaching out to a large amount of audiences. Moreover, advertisers usually allow web sites or advertising networks to dynamically decide what kind of ads to display to their visitors, based on the web contents users are viewing. This behavior is called "targeting" of ads. It makes Internet advertising more relevant and presumably more helpful to visitors. The profit of web publishers hosting ads can be calculated by different revenue models, including measuring how many times the ads are displayed, how many visitors have seen
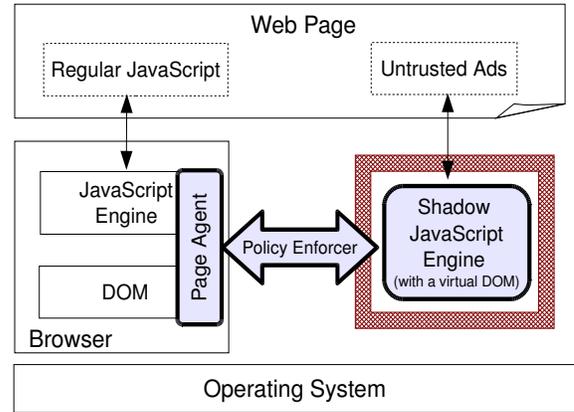
Figure 1: An architecture overview of AdSentry. The core components of AdSentry are highlighted in the figure.

the ads, or how many times the ads have been clicked by visitors, etc. [47]

Internet advertising brings in new challenges to web security and privacy. One possible way for publishers to include advertisements is to completely isolate them in separate iframes. However, such a complete isolation makes advertisement targeting impossible. As a result, third-party ads are often included in `<script>` elements, so they have the same privilege as other JavaScript on the web page.

In this paper, we focus on such third-party JavaScript-based advertisements (ads) that are deployed on web pages. These ads are hosted outside web publishers' servers, but included as JavaScript on the web pages. If some of them become malicious, they may abuse their privileges in accessing web application data for various purposes, such as leaking confidential user information and issuing unauthorized transactions. Moreover, they may exploit software vulnerabilities in browsers to take over the users' systems.

Our goal in this work is to comprehensively confine these untrusted JavaScript ads and effectively protect users' privacy and the integrity of both web applications and the users' computer systems.

3. SYSTEM DESIGN

To effectively confine untrusted ads, we have four design goals, i.e., *comprehensiveness*, *flexibility*, *transparency*, and *efficiency*. By comprehensiveness, we aim to provide an integrated scheme that not only regulates ad access to the host web page, but also contains malicious ads from launching drive-by downloads. The flexibility requirement allows both web publishers and end users to specify access control policy for ads. Users can also dynamically change access control decisions based on application runtime states. The transparency goal requires no modification to the browser for the support and preserves the timing of the JavaScript behaviors in the web applications. The transparency requirement ensures that the current billing model of ads is not affected. Actually, it is a stronger requirement than simply requiring no changes to ads billing. Also, the proposed solution needs to be efficient in introducing low performance or maintaining a similar level of user experience.

Following these design goals, we have developed a novel ad isolation framework called AdSentry, whose overall architecture is shown in Figure 1. In essence, AdSentry provides a shadow JavaScript engine to confine untrusted ads. This shadow JavaScript engine by default has no direct access to the original browser envi-

ronment and the operating system. Therefore, the ads can be fully confined, and the host web page and OS will remain intact even if the ads are malicious. To meet the transparency requirement, the shadow JavaScript engine can be seamlessly integrated into current browsers through the standard browser's extension application programming interfaces (APIs), i.e., no browser modification will be necessary.

With the introduction of a shadow JavaScript engine, AdSentry essentially works with two JavaScript engines: the untrusted ads run inside the shadow engine while the rest (normal) JavaScript in the web page runs as usual in the default engine. We point out that an ad may have legitimate reasons to access certain web content (e.g., for the purpose of advertisement targeting). To accommodate these requests, AdSentry provides a virtualized *DOM* to the shadow JavaScript engine. The virtual DOM has all the standard DOM interfaces, including XMLHttpRequest, so page accesses made by ads running in the shadow engine will be received by the virtual DOM. When the virtual DOM is being accessed, it will relay the access to the *page agent* in the browser through a *policy enforcer*. The policy enforcer will decide whether a page access is allowed by users' security policies. If yes, the page agent proceeds with the access request on behalf of the isolated ad, and returns the results back to the isolated ad through the virtual DOM. If not, the access will be blocked to protect the integrity of the web page.

Besides virtualizing the DOM access for untrusted ads, AdSentry also sandboxes the ad execution within the shadow engine, preventing them from compromising users' operating systems.

It is important to note that AdSentry is transparent to web pages by automatically dispatching ads to the shadow engine and seamlessly supporting their accesses. As a result, the billing model of ads is not affected. AdSentry preserves the original execution timings of all JavaScript in the web page, including ads scripts running in the shadow JavaScript engine. This is a key advantage of AdSentry over iframe-base isolation techniques, such as AdJail. This ensures that the behaviors of the applications and user experience will not be altered unless for security concerns, making AdSentry applicable to securing a wider class of untrusted JavaScript code in web applications.

3.1 Shadow JavaScript Engine

By introducing a shadow JavaScript engine to host untrusted ads, AdSentry allows us to achieve the comprehensiveness goal: resilience against exploits to browsers themselves and protection for the confidentiality and integrity of web application data. As mentioned earlier, the shadow JavaScript engine is executed inside a Native Client (NaCl) [48] sandbox. There are two reasons why we choose to build our system on top of NaCl. First, NaCl sandbox has been shown to be secure against code injection attacks with minor performance overhead. Second, NaCl has been supported on a number of platforms, such as x86, x86-64 and ARM [41], which can be very helpful for adopting AdSentry by end users, especially with the rising popularity of the Google Chrome browser and the Chrome OS that ships NaCl as one built-in component.

Like the normal JavaScript engine in the current browser, the shadow JavaScript engine is shared across web pages in the browser. To distinguish different ads from different pages, each ad will be assigned a unique identification number. With that in place, when an ad needs to be executed, AdSentry sends its JavaScript and the ad's identification number to the shadow engine. To preserve the original execution timing of the web page, the browser waits until the ad's JavaScript finishes in the shadow engine, in the same way that the original browser JavaScript engine handles its execution.

Specifically, once the sandboxed JavaScript engine receives a

JavaScript to execute, it creates a new JavaScript context for the ad with the associated identification number. A virtualized DOM will be initiated to contain a set of global objects, which are then made accessible to the JavaScript context. The virtual DOM has all standard DOM interfaces, but each interface is simply a stub that forwards the access to it to the page agent in the browser. After the initialization, the shadow JavaScript engine starts executing the received JavaScript. If the script accesses a particular DOM interface, the access is intercepted by the virtual DOM, which in turn communicates with the page agent to handle the access request.

3.2 Page Agent

The intercepted DOM access requests from the virtual DOM are forwarded to the page agent, which resides on the same page with the web application. After the verification from the policy enforcer, the page agent will perform the requested DOM access on behalf of the ad. If the request is to create or modify DOM element(s), the page agent takes special care to capture all resulting JavaScript executions and forwards them back to the shadow engine for processing. For instance, when a user clicks on a button created by an ad, the triggered `onclick()` function call will be captured and executed in the shadow engine.

The communication between the virtual DOM and the page agent is in the form of message passing. When the page agent receives a message requesting a DOM access from the shadow JavaScript engine, it extracts the access from the message, and processes it in the context of the original web page. The page agent then sends the result back to the virtual DOM, and in turn, to the shadow JavaScript engine, completing the access made by the confined ad.

AdSentry also naturally regulates access to HTTP requests from untrusted ads. Specifically, ads may initiate HTTP requests by either generating new DOM elements (that have already been controlled by the page agent), or by directly initiating XMLHttpRequest. As XMLHttpRequest is not part of the JavaScript engine, but is provided by the virtual DOM, the invocation to XMLHttpRequest is also regulated by our system.

In order to ensure that the relayed DOM access is transparent to the executing ad, we need to address a few issues – some of them come from innate JavaScript features.

Dynamically generated JavaScript.

Ads can insert a new piece of JavaScript into a web page. The new JavaScript must also be executed in the same shadow JavaScript engine. Otherwise, the newly generated JavaScript can escape the isolation of AdSentry.

There are several ways for ads to introduce new JavaScript into the original web page. Examples include abusing `document.write` or setting the `innerHTML` attribute of an element. Accordingly, whenever AdSentry receives a message from the shadow JavaScript engine requesting to invoke such DOM interfaces, the request is interpreted and all newly introduced JavaScript is properly flagged to ensure they will execute in the shadow JavaScript engine. We detail this solution in Section 4.

Timers and event listeners.

One interesting challenge comes from the support of asynchronous events, such as timers, where ads register callback routines to be executed later. Such callback routines need to be executed in the shadow JavaScript engine. To handle these asynchronous events, AdSentry dynamically creates a stub as the corresponding event handler in the web page. This stub will notify or invoke the true callback routine in the shadow JavaScript engine.

Unfortunately, as asynchronous events occur unexpectedly, the

notifying message sent to the shadow JavaScript engine may arrive in the middle of the execution of some other DOM accesses, which causes an undesirable race condition. To avoid that, the messages from asynchronous events will be separately marked and temporally buffered by the shadow JavaScript engine. These messages will then be processed after the ongoing DOM accesses are finished.

Anonymous functions.

The JavaScript language supports anonymous functions. For example, the following code snippet creates an anonymous function with a function body *alert(0)*.

```
window.addEventListener("click",
        function () { alert(0); },
        false);
```

As an ad may create these anonymous functions, we need to isolate them properly. Particularly, if these anonymous functions are being used as event listeners, they should be invoked within the shadow JavaScript engine when the corresponding events occur. Unfortunately, anonymous functions are represented as native function objects in the JavaScript engine, rather than strings of JavaScript code. Therefore, we cannot handle them in the same way as we do for JavaScript code on the page.

To address this problem, in our system, when the shadow JavaScript engine executes a statement that creates an anonymous function, we record the function's internal identification number, associate that number with the related DOM access, and forward it to the page agent. When the page agent receives the message at the real DOM side, it dynamically composes a new JavaScript function whose task is just to send a message containing the identification number of the anonymous function to the shadow JavaScript engine. After that, it assigns the newly composed function as the argument to the event listener. When the event occurs, the newly composed function will be invoked (at the real DOM side) to send a message to the shadow JavaScript engine and ask it to run the anonymous function with the specified identification number.

3.3 Policy Enforcer

By confining the shadow JavaScript engine within a sandboxed environment, our system effectively blocks possible drive-by downloads that target the underlying JavaScript engines (more concrete examples will be shown in Section 5). In the meantime, it is important to point out that the sandbox itself does not provide any guarantee on the confidentiality or integrity of the web application. As a result, it needs to work in concert with the policy enforcer to achieve this goal. Specifically, the policy enforcer checks the requests intercepted by the virtual DOM according to a given user security policy. Only if allowed by the enforced security policy, the request will then be forwarded to the page agent for processing.

As mentioned earlier, our system allows both web publishers and end users to customize the access policy for ads. Specifically, for web publishers, as they can simply change the web page content, they may choose to wrap the ad and confine its execution in the shadow JavaScript engine. For end users, our current system leverages Adblock Plus [34] to automatically identify ads and confine them with a customized JavaScript wrapper. We will present the details as well as the supported policies in the next section.

4. IMPLEMENTATION

We have implemented a proof-of-concept prototype of AdSentry based on the browser extension support of Firefox, and it is implemented and tested in Mozilla Firefox 3.5.8. Our implementation

of the shadow JavaScript engine is based on Mozilla SpiderMonkey version 1.8.0. The virtual DOM support is generated with a code generator of 770 SLOC in perl. On the browser side, the other two components (i.e., the policy enforcer and the page agent), are implemented entirely in the JavaScript language. These two components add about 3100 SLOC.

4.1 Specifying Advertisement Scripts

AdSentry is flexible in deployment. It allows both web publishers and end users to specify the scripts to be executed in the sandbox. The intuitive way is to associate an attribute with the script indicating it is an advertisement script. However, this solution requires the browser to be modified to recognize the attribute. In AdSentry, we provide a function sandboxAds. It takes the body of an ad or the URL of an ad script, and notifies AdSentry to execute it in isolation.

Therefore, to use AdSentry, web publishers can process the advertisement with the function sandboxAds, as illustrated by the following example, where the last argument indicates whether the first argument is the URL (true) or the body of an ad script (false).

```
<script>
  sandboxAds('http://ads.com/ad.js',
            id, true)
</script>
```

AdSentry also provides the option to end users by automatically identifying ads instances at the client side. It uses Adblock Plus [34] to identify ads and automatically processes them with sandboxAds.

4.2 Shadow JavaScript Runtime and Virtual DOM

We use NaCl to sandbox the SpiderMonkey JavaScript engine. We found this process relatively straightforward. However, the main challenge comes from the extension we make to the original JavaScript engine. Specifically, to enable ads running in the JavaScript environment to access related page content (e.g., for ad rendering), there is a need to provide virtual DOM objects. In our prototype, a virtual DOM is made available to the JavaScript engine in the form of a tree of objects. The root of this tree is called the global object. [1] In the case of a web page, the global object is the window object. This global object has a number of properties, including global JavaScript variables and functions, such as the document object, the location object, and the eval function. With this tree structure, all other virtual DOM objects are also properties of their parent objects.

We obtain a standard DOM structure from the standard DOM specifications [44], construct virtual DOM objects and expose them to the shadow JavaScript engine as host objects. More specifically, the virtual DOM for SpiderMonkey is generated in the following steps: 1) Create a new JSRuntime object and set up initial configurations and in runtime, create JSContexts for the execution of ads scripts. 2) Create a JSClass for each class of DOM objects. 3) Specify properties and member functions for each JSClass. 4) Implement property and function accessor methods for each JSClass, most of which will invoke one of the centralized access handling functions, respectively. 5) Implement the centralized access handling functions for virtual DOM accesses. These functions will then relay the access to the page agent (on the browser side). To relay the access, they also perform other tasks, such as preparing arguments to actual DOM function calls, looking

---

[1] Note that the concept of the global object here is different from that of global objects in a JavaScript program.

for anonymous functions, buffering event listener code for later execution, etc. These functions interpose each and every access from ads to the real DOM. 6) Create instances of standard objects from `JSClass` definitions, starting from the `window` global object. For non-global objects, we will specify their parent objects during the creation to form the tree structure.

Considering the large number of virtual DOM objects we need to construct and the associated tree structure we need to maintain, we have a code generator in place to automate the above steps for all virtual DOM objects. The code generator reads in an XML file that specifies the DOM tree objects and structures, and then generates an output file that embeds the JavaScript engine and sets up its host environment with the virtual DOM.

4.3 Page Agent

To facilitate the communication between the shadow JavaScript engine and the page agent, we define a simple message format for data exchange. The format is summarized as follows:

```
msg::= command data
command::=  script | callFunc | getProp
            | setProp | return
data::= <text>
```

Each message contains a *command* field and a related *data* field. Our prototype has defined five different commands: a `script` command is used to notify the page agent that an ad script needs to be sent to the shadow engine for execution. Upon receiving the message, the shadow engine will prepare the runtime environment and then start executing it. During execution, it will intercept any DOM access from the ad script and based on the type of access, translate it into three other types of messages to the page agent: `callFunc` for function invocations, `getProp` for property retrievals, and `setProp` for property (re)initialization. Finally, a `return` command carries the results in the message body, i.e., the *data* field.

The page agent extends the Firefox browser through its standard extension interfaces. We create a Firefox extension, which monitors the dispatched message events notified by `sandboxAds`. Specifically, following the above message format, if a *script* command is received, it parses the message stored in the event object, and communicates with the sandbox. We stress that web pages cannot directly communicate with the sandbox, and all communications are done via the Firefox extension. During the ad execution, if it needs to access a DOM object, the sandbox intercepts it and encapsulates the access by sending a message to the page agent requesting a DOM access. Here, the DOM access is meant for the access of the real web page and the extension cannot evaluate it in its own execution environment.

There are two possible approaches for our extension to evaluate the intended DOM access in the web page context. The first approach is straightforward: simply posting a message (or dispatching a custom event) to the web page. After receiving it, the web page can then evaluate the requested DOM access (encoded in the message or event). However, message passing is asynchronous, which allows other JavaScript on the same web page to preempt the execution of the current ad script. This kind of preemption may cause serious problems as it alters the original execution order of different scripts on the page. For example, scripts may have dependency on each other, and a premature execution of a later script may fail if the dependent script has not been executed. As another example, `document.write` is normally executed before a web page is loaded. If it's executed after a page is loaded, it creates a new page, completely eliminating the original one. To execute

sandboxed scripts normally, AdSentry should not alter the original execution order of scripts on the web page, so this first approach is not suitable here.

The second approach is to implement the communication between the web page and the extension like a function call. Mozilla Firefox provides a mechanism for extensions to evaluate JavaScript code in web pages' privileges, called `evalInSandbox` [32]. In our prototype, we leverage this method to call a function in the context of the web page that contains the ad script, which in turn evaluates the DOM access being requested, and returns the result to the extension. After that, the extension sends it back to the sandbox via a pipe. By doing so, when an ad script is being executed, we can ensure the JavaScript engine in the original browser environment is always in one of the three states: a) waiting for messages from the sandbox; b) executing our script in the extension; or c) executing the message processing function in the host web page while our extension is waiting for the return. As a result, no other scripts on the web page could preempt the current execution of ads script.

Consequently, our implementation is based on the second communication approach. More details are discussed below.

Concurrent ads scripts.

AdSentry supports processing multiple ads scripts concurrently. To avoid mix-ups of ad scripts from different web pages, our browser extension maintains a message queue to ensure that only one ad script is being processed at any point of time. Each message sent to the shadow engine is marked with an identification number, enabling the engine to evaluate each ad script in its own JavaScript context. When evaluating DOM accesses requested by the sandbox, AdSentry also makes sure the accesses will be evaluated in the same page that originally contains the ad script being executed.

Object maps.

The communication mechanisms implemented in AdSentry are text-based, but in some cases we need to pass objects as parameter or return values. This is achieved by maintaining object maps at both the page agent and the shadow JavaScript engine, and only communicating the objects' indices in the messages. Before a JavaScript object is to be communicated to the other end, it is checked against the local object map. If it already exists in the map, its index is returned; otherwise, it is inserted into the map with its new index returned. Then in the message sent, the index of the object is included, rather than the object's real data. Next time when a message is received from the other end containing an object index, the object is restored by querying its index from the local object map.

Parameter buffering.

AdSentry enforces security policies on the result of JavaScript actions, which will be described in Subsection 4.4. One possible way to bypass our access control policy enforcement is to insert content into the web page piece by piece. For example, instead of calling

```
document.write("<scr" + "ipt> some script <"
                + "/scr" + "ipt>");
```

malicious ad script may attempt to avoid being detected by inserting a `script` element like the following

```
document.write("<scr");
document.write("ipt> some script <");
document.write("/scr");
document.write("ipt>");
```

This way, checks on parameters to each individual DOM function call would not detect that a new `script` element is being inserted.

To prevent such misuses, AdSentry buffers such consecutive function calls by not sending them one by one to the shadow JavaScript engine for execution, but finally replace them with a single call with the entire piece of content being inserted, which is checked by the access control policy enforcer as normal.

4.4 Access Control Policy Enforcement

To regulate the communication between the host web page and the confined ad script, our policy enforcer acts as a moderator. Any communication between the two parties needs to be approved according to a given policy. AdSentry is flexible in allowing both web publishers and end users to specify the access control policies for ads.

AdSentry has a default policy. The default policy disallows any JavaScript code originated from ads to run in the host web page. In other words, all untrusted scripts will be guaranteed to be only executed inside the shadow JavaScript engine. To enforce that, we examine all incoming messages from the sandbox, distinguish page updates containing dynamic JavaScript content versus static HTML, and then handle them accordingly.

Specifically, for the static HTML content, our system first normalizes the HTML into the corresponding XML format and then serializes the XML back to HTML before processing. The HTML code is widely known as badly formed, to the point that badly written code is often called "tag soup" [6]. Also, all major browsers have permissive parsing behaviors by supporting a rendering mode called "quirks mode" beside the "standards mode" [5]. These browser quirks have many negative implications, one of which is that malicious attacker can embed JavaScript code inside a malformed fragment of HTML code. To strive a balance between security and the support of potential browser quirks, we took three phases for parsing HTML code. First, we attempt to reformat the code by correcting popular mistakes in web authoring. For instance, we close all open tags and correct all improperly nested tags. Second, we leverage the XML parser in the web browser to parse this reformatted code into a XML model. Note that a malformed HTML is considered dangerous and will be rejected by our parser. Since XML parser is strictly standard-compliant, any surviving formation will bear no ambiguity. Finally, we serialize this XML model back to HTML code before handing to the page agent for further processing.

For the JavaScript dynamically generated by ads scripts, we install wrappers that request the sandbox to run the dynamic JavaScript code. In other words, all untrusted scripts are guaranteed to execute inside the shadow page, not the real page. In our prototype, we apply the code wrapping based on the above XML model. Specifically, we leverage the XML XPath facility available in most browsers to traverse the XML model tree and inspect the enclosed nodes. We first query for patterns of dynamic code on the model. These patterns of dynamic code include event handlers such as `onclick()`, as well as related JavaScript functions such as `addEventListener()`, `setTimeout()` and `setInterval()`. The resulting node set will then be properly wrapped or transformed. In our prototype, we have installed wrappers on all 32 possible vectors of dynamic code and ensure that no potentially malicious code will ever be injected to the real page. As an example, the following code snippet

```
setTimeout(' slideAd(10,100); ', slideDelay);
```

will be transformed into the following code fragment:

```
setTimeout(' sandboxAds(" slideAd(10,
      100);", id, false); ', slideDelay);
```

To further ensure the privacy of sensitive user data in the web page, we allow users to configure the data to be shared with the script. As mentioned earlier, we do not copy all the content of the real DOM to the virtual DOM. Instead, we choose to interpose on every access to the virtual DOM from the untrusted ad and subject it for policy verification. As such, users can decide to be extremely cautious with certain kind of ads, and block any read access from the ad to the entire page. On the other extreme, a user might want to trust certain ads, and allow free accesses to the real DOM content. In addition to the above two policies, a user is also allowed to specify a policy that blocks accesses to the `document.cookie` object or mandates that ad can only read from its own elements and not the surrounding content. Moreover, an ad can be prohibited from appearing outside of the allocated region of the web page (by stating the allowed values of `width`, `height` and `overflow` property of ad elements). This is helpful to thwart some types of phishing attacks. In fact, as a comprehensive isolation framework, our system provides a mediation capability that can accommodate existing access control polices [23] for ads. And both web publishers and end users can take the advantage of the same capability to enforce security policy on ad behaviors.

AdSentry also enables end users to dynamically specify access control policy with tools during the execution of web applications. In addition, AdSentry leverages a customized version of Adblock Plus [34] to automatically identify and wrap ads scripts on web pages.

5. EVALUATION

In this section, we evaluate the functionality and performance of AdSentry. In particular, we have conducted four sets of experiments. The first one is based on real-world browser exploits to evaluate AdSentry's defense against drive-by download attacks. The second one is to test its resilience against malicious attempts that inject JavaScript into web applications. The third one is to evaluate AdSentry's protection of privacy against rogue information-stealing ads; The fourth one is to measure the performance overhead. Our experiments were conducted on a Dell E8400 workstation with a Core 2 Duo CPU (3GHz 6 MB L2 Cache) and 4GB of RAM. The system runs Ubuntu 9.10 and we its the default web browser – Mozilla Firefox 3.5.8 – for our experiments.

5.1 Browser Exploits

To evaluate the effectiveness of AdSentry in sandboxing ads, we conducted experiments with a few real-world exploitations, obtained from existing research work [25] as well as vulnerability databases [1, 31]. All the exploits we tested with caused the vulnerable versions of the Firefox browser to crash during our experiments. They are all marked as critical by Mozilla developers, and can be further crafted to launch severe attacks such as drive-by download.

Our experiments are summarized in Table 1. The eight examples exploit the vulnerabilities in the SpiderMonkey JavaScript engine. Most of them are various instances of buffer overflow or memory corruption attacks, and they could lead to arbitrary code execution. With AdSentry installed in the vulnerable versions of the Firefox browser (in our experiments, we used Firefox 3.0 and Firefox 3.5 for corresponding exploits), each of the exploits was successfully contained by the shadow JavaScript engine. This confirmed and demonstrated one of our design goals that we would like to run untrusted ads scripts in an isolated environment so that even in the worst case, they would not crash the entire web browser. As AdSentry sandboxes the JavaScript engine, so any memory attack against vulnerabilities in the JavaScript engine would be contained by the sandbox.

Bugzilla ID	Attack Behavior	Outcome
426520	Browser crashed by memory corruption with crafted XML namespace	Contained by shadow JS engine
454704	Browser crashed by exploiting a vulnerability of XPCSafeJSObjectWrapper	Contained by shadow JS engine
465980	Browser crashed by pushing to an array of length exceeding limit	Contained by shadow JS engine
493281	Browser crashed by stack corruption starting at unknown symbol	Contained by shadow JS engine
503286	Browser crashed by exploiting a vulnerability of Escape()'s return value	Contained by shadow JS engine
507292	Browser crashed by incorrect upvar access on trace involving top-level scripts	Contained by shadow JS engine
561031	Browser crashed by overwriting jump offset	Contained by shadow JS engine
615657	Browser crashed by buffer overflow due to incorrect copying of upvarMap.vector	Contained by shadow JS engine

Table 1: AdSentry evaluation using browser exploits

Scenario	Attack Vector	Attack Behavior	Outcome	Description
1	Direct code injection	Inject script	Blocked	Denied by the default policy
2	Browser parsing quirk	Malformed $<img>$ tag	Blocked	Rejected by message normalization
3	Browser parsing quirk	Malformed $<script>$ tag	Blocked	Rejected by message normalization
4	Browser parsing quirk	Malformed $<script>$ tag	Blocked	Rejected by message normalization
5	Browser parsing quirk	Malformed $<b>$ tag	Blocked	Rejected by message normalization
6	Browser parsing quirk	Malformed $<script>$ tag	Blocked	Rejected by message normalization
7	Browser parsing quirk	Malformed $<iframe>$ tag	Blocked	Rejected by message normalization

Table 2: AdSentry evaluation using JavaScript injection attacks

5.2   Script Injection by Ads

In our second experiment, we evaluated the effectiveness of our default policy in preventing untrusted code from being injected from the ad to the web page. In particular, we examined the XSS Cheat Sheet [7] and identified a number of cases that can successfully result in injecting JavaScript from the ad into the web page for execution. We confirmed the successful injection and execution in the default Firefox without AdSentry being installed. During our experiments, we explicitly cleared the browser's cache between each step.

Our results are shown in Table 2. The first one is a direct attempt to include an external JavaScript to execute in the web page while the other six exploit numerous parsing quirks [7]. Such attacks are created to execute a simple script that displays a message box "hacked!" The use of browser parsing quirks reflects the current trend [24] in part because they are much harder to repair without breaking compatibilities with legacy web applications. This was blocked by the default policy in AdSentry that direct injection of scripts into the web page is disallowed.

For the rest examples, we use the second scenario as the representative. Specifically, in the second scenario, the attempt is to exploit a parsing quirk by embedding a `<script>` tag as literal text inside a `<img>` tag, which will cause the browser to interpret the text string as JavaScript code, thus causing an injection:

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>">
```

The related code snippet is shown above. It contains three pairs of double-quotes, encapsulating different parts of the text. If a parser were properly implemented, there would be three literal strings: an empty string `""`, the second string `"><SCRIPT>alert("` and the last string `")</SCRIPT>"`. These three strings are orphaned as they are not assigned to any property of the tag and therefore should be discarded. As such, the entire tag should simply collapse to `<IMG XSS>`, which can also be disregarded. However, this is not the case in most modern browsers. In fact, existing browsers tend to be very permissive in their parsing behavior [5]. For instance, we observed that Firefox interpreted `<IMG """>` as the first tag and `<SCRIPT>alert("XSS")</SCRIPT>` as the second tag; the remaining `">` was accepted as plain text and displayed as is. As a result, the "malicious" code `alert("XSS")` was executed. This attempt was blocked because of the normalization

through the standard-compliant XML in our system. We successfully detected this malformed HTML content and substituted it with the benign static text "Script Injection Blocked."

5.3   Privacy Protection

In our third set of experiments, we test our system from the privacy perspective. In particular, it has been known that third-party JavaScript can violate user privacy in various ways. Examples include cookie stealing, location hijacking, history sniffing, and behavior tracking [20]. In our experiments, we evaluated AdSentry with a synthesized ad that simulates the above information-stealing behaviors.

In particular, the synthesized ad is developed to perform all these four types of behaviors: The cookie stealing is implemented to access the `cookie` property of `document` object; The location is hijacked by setting the `location` property of `window` (or `document`); The previously browsed URLs are sniffed by obtaining the color of the populated hyperlinks, which can be done by invoking the `getPropertyValue` function of the `ComputedCSSStyleDeclaration` object (with the argument "color") [2]; Behavior tracking is achieved by registering related event listeners of interested elements, such as `onclick`, `onmouseover`, etc.

AdSentry successfully detected each of the above four types of behaviors. For the first two types, our system simply denies the read access to the `document.cookie` and the write access to the `window.location` and `document.location`. For the third type of ad behavior, it is detected by monitoring any invocations to the related `getPropertyValue` function. For behavior tracking, AdSentry refused the registration of callback routines of those elements if the ad does not own them.

We stress that our privacy protection enforcement does not suffer from JavaScript object and property aliasing problems. This is because the access is intercepted by the virtual DOM that, when invoked, has already resolved all object and property aliasing, if any.

Moreover, we also evaluated the user experience of AdSentry using 15 popular website with ads, shown in Table 3. The embedded ads are automatically recognized by the Adblock Plus extension

---

[2]Recent browsers return the same computed styles for visited and unvisited links.

Web site	Properties of ads
www.msn.com	Ads on different domain of same company
www.aol.com	Ads on content distribution network (CDN)
www.livejournal.com	Ad network DoubleClick
espn.go.com	Ad network DoubleClick
www.cnet.com	Ads on different domain of same company
imageshark.us	Ad network Google
www.nytimes.com	Ad network Checkm8
www.ehow.com	Ad network YieldManager
sourceforge.net	Ad network DoubleClick
www.reference.com	Ad network DoubleClick
www.dailymail.co.uk	Ad network DoubleClick
www.guardian.co.uk	Ad network Google
www.gmx.net	Ad network Uimserv
yfrog.com	Ad network Rubicon Project
www.comcast.net	Ad network Yahoo!

Table 3: Web sites used in user experience evaluation

and then transparently confined with AdSentry. To allow users to interactively specify security policies, we integrate a Firefox extension called Firebug [3] and extend it with a pop-up menu that can be triggered with a right mouse click. Specifically, we use the Firebug to visually capture available screen regions and for a selected region, a right mouse click will activate the pop-up menu. From the menu, a user will be shown the list of ads (grouped by domains) currently embedded in the current page and can then choose which ad can have a read access to the chosen screen region or can register call-back routines (e.g., event listeners). By default, these ads are only allowed to read their own elements, not the surrounding areas. Users can also specify new policies during run time, which will overwrite existing ones if necessary.

Our experiments did not find any suspicious information-stealing behavior for these websites.

5.4 Performance Evaluation

In order to assess the performance overhead, we conducted experiments to measure the page load overhead. We picked up four typical ads, one from each of the top four ad networks. We created a test page for each ad and ran the test page with and without AdSentry. Each experiment was repeated for 20 times, and the average results were recorded.

Our results are shown in Table 4. Overall, AdSentry incurs small overhead. The relative overhead ranges from 3.03% in MSN Ad Network ad to 4.96% in Google Adsense ad. We observed that a typical ad might only infrequently access DOM namespace, which might attribute to the low overhead. From another perspective, the relative overhead can be low because ad content such as images are often dynamically loaded from a remote server, this process experiences network round trip delay that is typically much more significant than local computation time in web browsers. Also, to improve responsiveness, modern browsers typically start rendering any elements immediately once they are available. Therefore a user may not notice the difference in the speed of ad loading time at all. In other words, this pipelining of the rendering process contributes to masking the delay that may be experienced by any single element in a web page.

In addition to the above real ads, we also measure the time needed to initialize our sandbox. Our results show that it takes 31 ms to initialize and set up the sandbox. Though it is lightweight, we expect opportunities still remain to reduce the time by further optimizing the JavaScript engine and NaCl sandbox. Finally, we evaluate a round-trip communication delay for a virtual DOM access. Without our system, it typically took 0.001 ms for the ad to finish the

reading of a particular DOM property. When being confined, it will take 0.59 ms. This is expected as it needs to cross the sandbox boundary and go through the normalization for policy verification. Note that this overhead will be effectively amortized in real-world scenarios – as demonstrated in the four real ads.

6. DISCUSSION

In this section, we discuss the limitation of AdSentry and future work. First, our current work focuses on the JavaScript-based advertisements and has not yet explored the support of other types of advertisements. In particular, Flash technology is another popular way to write and display ads, which still remains to be investigated how flash-based ads can be supported.

Second, AdSentry protects the browsers from attacks exploiting vulnerabilities of the JavaScript engine, but it is not designed to prevent attacks to other browser components, such as the HTML rendering engine. If the malicious HTML segment is dynamically generated by JavaScript code, AdSentry's policy engine can mitigate the attack by the HTML normalization and signature-based attack blocking. A more general solution is to extend our solution to isolate other components of the browser.

Finally, we will continue to work on improving AdSentry's compatibility with JavaScript on a web page. Our prototype implementation is able to handle typical JavaScript advertisements, which has limited ways in accessing other parts of the web page. However, third-party JavaScript code in general has much tighter integration with the rest of the web page. As our future work, we will improve the support for transparently isolating a wider class of JavaScript code in web applications. It will also be interesting to investigate possible ways (e.g., in software testing) that automatically test AdSentry's compatibility with a broader set of web applications.

7. RELATED WORK

In this section, we discuss existing work that mitigates threats from untrusted web content embedded into web applications, including those compromising user data, web application integrity as well as users' operating systems.

Drive-by download prevention.

Drive-by downloads are serious threats to web and host security [37,38]. BLADE [25] proposes a detection system for drive-by download exploits. This type of attacks has recently received lots of attention. For example, heap-spraying attacks can pre-populate a large heap space with attack code and a software bug can be exploited to redirect execution flow to the heap sprays (with attack code). In addition, several systems [11, 12, 39] have been proposed to leverage specific memory characteristics of these attacks to identify them and prevent browsers from being exploited. WebShield [21] proposes a middlebox framework that processes page contents in a shadow browser, and transforms DOM updates to the client browser to reflect DOM changes there. As a result, drive-by downloads can be detected at the middlebox without affecting the client browser. Other existing sandbox and isolation solutions [14, 22] can also be used to protect the operating system against drive-by download attacks. Compared to AdSentry, solutions in this category are not designed to protect user privacy and web application integrity from malicious JavaScript ads.

Isolation in web browsers.

Several recent research projects [9,16,46] attempt to achieve better browser security architecture by running different browser components in isolated environments. The Google Chrome browser

Performance Test	with AdSentry (ms)	without AdSentry (ms)	Overhead ($\%$)
Google Adsense Rendering	381	363	4.96
DoubleClick Ad Rendering	601	578	3.98
MSN Ad Rendering	1224	1188	3.03
Yahoo Ad Rendering	1539	1475	4.34

Table 4: Runtime page load overhead of AdSentry

also uses a sandbox to isolate browser components and protect the operating system [8, 29]. The IBOS [42] system steps further by designing a secure architecture for both the operating system and the web browser altogether, minimizing default sharing and trust between software components. However, they do not support isolating JavaScript ads from the rest of web applications, while AdSentry executes untrusted ads scripts in a separate and sandboxed environment from trusted scripts, mediating every access from ads to web applications.

Web application integrity protection.

To prevent tightly-integrated third-party JavaScript from affecting the integrity of web application, one type of solutions [2,10,13, 17,26,27] restricts the "dangerous" functionality of JavaScript. For example, ADsafe [10] only allows ads to use a safe subset of the JavaScript functionality. It removes dangerous JavaScript features, such as global variables, `eval`, `this`, and `with`. ADsafety [36] proposes a lightweight and efficient verification for JavaScript sandboxes, and has been successfully applied to ADsafe. Another line of solutions [4,19,35,40,49] protects web application against JavaScript ads through code transformation, enforcing policies against malicious JavaScript at runtime. Similarly, ConScript [30] introduces aspect into JavaScript language to enforce users' security rules. MashupOS [45] proposes new script integration primitives reflecting different trust relationships between the integrator and the mashup content provider. Besides enabling web publishers to protect their web applications, AdSentry also allows end users to flexibly specify access control policies according to their own requirements.

AdJail [23] addresses the privacy and web application integrity threat from ads by isolating them into an iframe-based sandbox. Using a separate origin in the sandbox, AdJail leverages browser's native origin-based protection to isolate ads. It is a solution for publishers to isolated third-party ads. Compared to AdSentry, AdJail assumes the ads on a web page are relatively independent and do not have tight dependencies with the page environment. For example, ad scripts cannot access global JavaScript objects defined or overwritten by other trusted scripts in the same hosting page. AdSentry transparently supports tight dependency between ads and the host page, without significant modification of the web page. It also provides flexible control of behaviors of JavaScript ads.

In addition, solutions in this category cannot prevent malicious ads from exploiting browser vulnerabilities.

Privacy protection.

One of users' major concern about JavaScript ads is privacy. Privad [18] proposes a solution to protect users' privacy by making users anonymous to the advertisers and publishers, but it does not prevent users' data from being used by the ad script, which may implicitly leak our user data. Adnostic [43] uses a browser extension to perform ad targeting, selecting ads to display from a larger set of ads sent by the advertisement network. Compared to AdSentry, both solutions only focus on protecting users' privacy, and do not address the ad's threat to integrity of web applications and the underlying operating system.

8. CONCLUSION

JavaScript-based advertisements are ubiquitous on the Internet. They pose threats to the privacy and integrity of web applications, as well as security of operating systems. In this paper, we present the design, implementation, and evaluation of AdSentry, a comprehensive and flexible framework to confine untrusted JavaScript advertisements. AdSentry not only separates the untrusted ad execution in a shadow JavaScript engine, but also mediates their access to the main page with access control policies, which can be specified by both web publishers and end users. We have implemented a Linux-based prototype of AdSentry that supports current Firefox browsers. Our experiments with a number of ad-related exploits show that AdSentry is effective in blocking these attacks. Our performance evaluation shows that the comprehensive protection is achieved with a small performance overhead.

Acknowledgments

9. REFERENCES

[1] Common vulnerabilities and exposures. http://cve.mitre.org/.

[2] FBJS (Facebook JavaScript). http://developers.facebook.com/docs/fbjs/.

[3] Firebug. Web Development Evolved. http://getfirebug.com/.

[4] Google Caja. http://code.google.com/p/google-caja/.

[5] Quirk Mode. http://en.wikipedia.org/wiki/Quirk_mode.

[6] Tag Soup. http://en.wikipedia.org/wiki/Tag_soup.

[7] XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html.

[8] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. The security architecture of the chromium browser. http://seclab.stanford.edu/websec/chromium/.

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

[10] D. Crockford. ADsafe. http://www.adsafe.org/.

[11] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap Taichi: Exploiting Memory Allocation Granularity In Heap-Spraying Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[12] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browser against drive-by downloads: Mitigating heap-srpaying code injection

attacks. In *Proceedings of the 6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.

[13] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure javascript subsets. In *Proc. of Network and Distributed System Security Symposium, 2010.*

[14] Goldberg, Wagner, Thomas, and Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 5th USENIX Security Symposium*, 1996.

[15] Google Inc. Google Fiscal Year 2010 Results, 2010. http://investor.google.com/earnings/2010/Q4_google_earnings.html.

[16] C. Grier, S. Tang, and S. King. Secure web browsing with the op web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[17] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[18] S. Guha, B. Cheng, A. Reznichenko, H. Haddadi, and P. Francis. Privad: Rearchitecting Online Advertising for Privacy. Technical Report MPI-SWS-2009-004, Max Planck Institute for Software Systems, Germany, 2009.

[19] S. Isaacs and D. Manolescu. WebSandbox - Microsoft Live Labs. http://websandbox.livelabs.com/, 2009.

[20] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[21] Z. Li, T. Yi, Y. Cao, V. Rastogi, Y. Chen, B. Liu, and C. Sbisa. WebShield: Enabling various web defense techniques without client side modifications. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2011.

[22] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, 2003.

[23] M. T. Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[24] M. T. Louw and V. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[25] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[26] S. Maffeis, J. Mitchell, and A. Taly. Run-time enforcement of secure javascript subsets. In *Proc of W2SP'09*. IEEE, 2009.

[27] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91, Washington, DC, USA, 2009. IEEE Computer Society.

[28] Matthew. Facebook's response to uproar over ads. http://endofweb.co.uk/2009/07/facebook_ads_2/.

[29] S. McCloud. The Chrome Comic Book, 2008. http://www.google.com/googlebooks/chrome/index.html.

[30] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[31] Mozilla. Bugzilla@mozilla. https://bugzilla.mozilla.org/.

[32] Mozilla. Components.utils.evalInSandbox. https://developer.mozilla.org/en/Components.utils.evalInSandbox.

[33] R. Naraine. Research: 1.3 Million Malicious Ads Viewed Daily. http://threatpost.com/en_us/blogs/research-13-million-malicious-ads-viewed-daily-051910.

[34] W. Palant. Adblock Plus. https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/.

[35] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, 2009.

[36] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX Security Symposium*, San Francisco, CA, USA, 2011.

[37] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th USENIX Security Symposium*, pages 1–15, 2008.

[38] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[39] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[40] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[41] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[42] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[43] V. Toubiana, A. Narayanan, and D. Boneh. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[44] W3C. Document Object Model (DOM) Specifications. http://www.w3.org/DOM/DOMTR.

[45] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashups. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 1–16, New York, NY, USA, 2007. ACM.

[46] 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 *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 417–432, Berkeley, CA, USA, 2009. USENIX Association.

[47] Wikipedia. Online advertising - Revenue models. http://en.wikipedia.org/wiki/Online_advertising#Revenue_models.

[48] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[49] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2007.