WebShield: Enabling Various Web Defense Techniques without Client Side Modifications

Zhichun Li[§] Tang Yi[†] Yinzhi Cao[‡] Vaibhav Rastogi[‡] Yan Chen[‡] Bin Liu[†] Clint Sbisa[‡]
[§]NEC Laboratories America [‡]Northwestern University [†]Tsinghua University, China
[§]zhichun@nec-labs.com [†]tangy05@mails.tsinghua.edu.cn [†]bliu@tsinghua.edu.cn
[‡]{yinzhicao2013,vrastogi,c-sbisa}@u.northwestern.edu [‡]ychen@northwestern.edu

Abstract

Today, web attacks are increasing in frequency, severity and sophistication. Existing solutions are either host-based which suffer deployment problems or middlebox approaches that can only accommodate certain security protection mechanisms with limited protection. In this paper, we propose four design principles for general middlebox frameworks of web protection, and apply these principles to design WebShield*, which can enable various host-based security mechanisms. In particular, we run* all *the JavaScript from remote web servers* only *at* shadow browser *instances inside the middlebox, and only run our trusted JavaScript rendering agent at client browsers. The trusted rendering agent turns browsers into a thin web terminal by reconstructing the encoded DOM of a webpage.*

We implement a prototype of WebShield*. Evaluation demonstrates that a general JavaScript rendering agent can render webpages precisely and be just slightly slower than direct access. We further demonstrate that our design can work well with interactive web applications such as JavaScript games.* WebShield *can detect attacks deeply embedded in dynamic HTML pages including the ones in complex Web 2.0 applications, and can also detect both known and unknown vulnerabilities. We further show that* WebShield *is scalable for deployment.*

1 Introduction

1.1 Motivation

Today, the web has become a primary attack target due to its popularity. The complexity of web systems further creates a lot of different kinds of vulnerabilities and attacks, such as drive-by-downloads, cross site scripting (XSS), cross origin JavaScript capability leaks, cross site request forgery, etc. As a result, many web attack defense mechanisms have been proposed [12, 14, 16, 17, 19, 26, 30, 34, 36]. For most of them, enhanced browsers, virtual machines (VM) or other defending programs need to be deployed on the client side. However, users are slow in adopting new technologies. Many users do not have any motivation to switch to new software, and are afraid of potential problems caused by new software. Therefore, almost all major attacks such as worms and botnets *successfully exploit existing vulnerabilities after the patches have been released for months or even years*. Moreover, the host environments on the clients are inherently heterogeneous and fragile. It is difficult to ensure those defense mechanisms do indeed work in such environments without extensive testing and high maintenance costs, which will further slow down the deployment. Even though users may initially agree to use new software, it is still difficult to persuade them to keep updating, especially if they have to restart their machines or browsers.

Researchers have realized the limitations of client-side deployment and proposed the use of middlebox-based approaches. In Table 1, we list the benefits of deploying security protection at a middlebox instead of at the client side. However, existing work [23, 31] mainly focuses on the design of special purpose middleboxes for very specific security protection mechanisms rather than a general framework encompassing various mechanisms.

Rewriting is one such mechanism. The seminal work BrowserShield [31] takes advantage of a lightweight middlebox to prevent the exploitation of browser vulnerabilities. Although rewriting adds special policies at the HTML/JavaScript level, it cannot enable detection/protection approaches that require internal states of the browser or underlying OS. The other middlebox work SpyProxy [23] proposes an execution-based approach. It renders and examines the active web content before the content reaches an user's browser. One major limitation, as admitted in their paper, is that the approach cannot cope with the non-determinism of web content and user inputs.

Client	Middlebox
heterogeneous & co-exist with other software high maintenance overhead user voluntary update	clean installation centralized control easy update and VM management

Table 1. Comparison between the client-side deployment and the middlebox-based deployment

Fundamentally, if a JavaScript program `P` in the webpage gets executed twice (once at the middlebox and the other at the client side), it is impossible to ensure the two executions will be exactly the same; thus, the security check can be bypassed. Many reasons can lead to different outcomes, such as randomness, and different parameters such as the current time or the number of CPU clock ticks. It is easy for attackers to design a JavaScript attack which behaves normally on SpyProxy and still attacks the client browser, as we show in Section 2.1.

1.2 Proposed Solution and Contributions

In this paper, we aim to design a general middlebox framework that can enable different security protection mechanisms. Our first contribution is to propose the following four design principles and, based on those, to design WebShield, a general middlebox framework.

Principle I is that a general framework should enable various protection mechanisms to protect clients from as many attacks as possible.

Principle II is that we should avoid deploying any additional programs on clients.

Principle III is that we should not allow any *untrusted* script execution at the client side without proper containment. In general, all scripts from web servers are treated as untrusted, since even well-known websites may have compromised webpages [8]. Moreover, JavaScript is very powerful for launching attacks. For instance, malicious JavaScript can employ heap-spraying [30] to easily exploit the browser vulnerability.

Principle IV is that the user's experience should not be sacrificed, *i.e.*, users should notice little change while benefiting from the middlebox approach.

We believe a general middlebox framework needs to consider all the four principles. Examining the existing middlebox design, BrowserShield violates *principle I* whereas SpyProxy does not abide by *principle III*, which limits the applicability of other security protection mechanisms.

In this paper, we propose WebShield, a general secure proxy for enabling different security protection mechanisms. Based on the four principles above, we make two design choices. (i) We take a conservative approach. We prohibit untrusted scripts from executing on the client, which is more conservative than *principle III*. Disallowing JavaScript execution of untrusted scripts greatly limits what the attackers can do at the client side, even if they have bypassed the detection on the middlebox. (ii) We would like to leverage the client side browser as little as possible (*i.e.*, a thin browser) because the complexity of browsers makes them more vulnerable. The idea is similar to thin clients vs. fat clients. At the client side, we would like to convert the full-featured fat web browsers to web terminals, which only handle input and output, and move the real browser logic into the middlebox.

Attacks	Defense Schemes
Drive-by-download	Nozzle [30], HoneyMonkey [36], Tahoma [16] and OP Browser [17]
Cross Site Scripting	DSI [26] and Javascript Taint [34]
Cross Site Request Forgery	[14]
Cross-Origin Javascript Capability Leaks	[12]

Table 2. Examples of web defense approaches that can be deployed with WebShield.

Table 2 gives a list of some browser security mechanisms that require client-side modifications. With WebShield, we can deploy these approaches at the middlebox (proxy) instead and achieve similar protection. To demonstrate our design in this paper, we mainly focus on detecting drive-by-download attacks.

In particular, we make the following additional contributions.

- We propose to run *all* JavaScript from remote web servers *only* at *shadow browser* instances inside the middlebox, and only run our trusted JavaScript rendering agent at client browsers. The trusted rendering agent turns browsers into a thin web terminal by reconstructing the encoded DOM of a webpage. Evaluation demonstrates that a general JavaScript rendering agent can render webpages precisely and be just slightly slower than direct access. We further demonstrate that our design can work well with interactive web applications such as

JavaScript games.

- We design an object pairing mechanism that strictly masks the URI requests introduced by the middlebox, which guarantees the correctness of web application logic. Existing works such as SpyProxy break the application logic in some cases (see Section 3.3).

We implement a prototype of WebShield and demonstrate that this architecture can incorporate different drive-by-download detection engines easily. Evaluation results suggest that WebShield with drive-by-download detection add-ons can accurately detect and filter drive-by-downloads, and the user-perceived slowdown due to WebShield is quite low. For the incremental rendering version on Chrome, the median increase of the rendering starting delay is 134 milliseconds and the median increase to the page load time is only 531 milliseconds (25% increase). These performance are also comparable to SpyProxy and BrowserShield. In the scalability evaluation, we show that a single machine with 16 GB of memory can support 70 active users. With the same machine, if we use lightweight SELinux-based sandboxes, the creation speed is about 28 sandboxes per second. The results show that, with moderate resources, the administrators of an enterprise can feasibly deploy WebShield to prevent web attacks.

2 Overview

2.1 Comparison with Existing Middlebox Approaches

Both SpyProxy and BrowerShield mainly target the drive-by-download attacks which compromise the host machines through browser vulnerabilities. With proper policy engines and/or behavior engines, WebShield can detect drive-by-download attacks as well, including the cases that cannot be detected by SpyProxy and BrowserShield, as shown in the example in Figure 1. When an attack targets an unknown vulnerability, vulnerability details are not available, so the policy-based approaches, such as BrowserShield, cannot be applied. An attack can also employ user events to bypass the detection of SpyProxy, since it only checks the initial rendering process. To trigger the attack, the code can require certain user input patterns, which is hard to predict beforehand, as shown in Figure 1.

Furthermore, WebShield aims to provide a general framework for deploying host-based defense schemes (examples shown in Table 2) without requiring browser/client modifications. Some defense mechanisms such as those for cross site scripting or cross site request forgery may require both client and server modifications. WebShield at least help eliminate the needs for direct client browser modifications, which we argue is hard to deploy.

```
var attackcalled=false;
function attackX() {
// exploit an unknown vulnerability,
//so BrowserShield cannot be applied

}
function loadAttack() {
    var el=document.getElementById(Evil);
    //use user events to bypass SpyProxy
    el.addEventListener(mouseover,
        checkMouse,false);
}
function checkMouse() {
    if (not attackcalled) {
        attackcalled=true;
        window.setTimeout(attackX,0);
    }
}
```

Figure 1. The attack code snippet that can circumvent both SpyProxy and BrowserShield.

2.2 Problem Definition

The research problem we target is how to avoid client-slide deployment while providing web security protection from a middlebox. The high-level idea is to reduce the browser to a web terminal, and to let most browser tasks execute on a secure proxy (middlebox) so that we can deploy security protection mechanisms at the proxy. This design also handles non-determinism and user-input triggered attacks. The key challenge is to maintain good performance and usability with this design, so that users will not notice any major difference while achieving high security protection.

2.3 Threat Model and Assumptions

Most web attacks are from malicious web content, mainly malicious JavaScript, in webpages. In our threat model, we assume any webpage going through the proxy is potentially malicious. Also, system administrators can define a whitelist of trusted webpages and a blacklist of webpages to block. We assume that the remaining webpages contain potentially malicious content, so we will apply WebShield to them.

WebShield is mainly designed to protect web users in enterprise networks, such as networks in companies, government agencies, schools, etc. We assume the round trip time (RTT) between any web user and the proxy is small. We verify the RTT in the campus network of Northwestern university. All RTTs of the hosts to our proxy server are within 2ms. We also assume that web users and the Web-

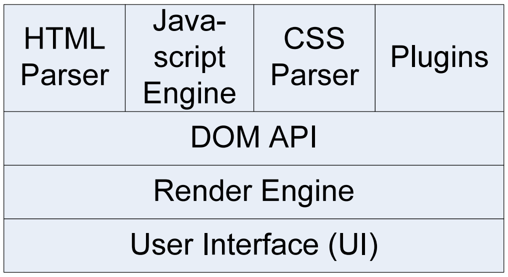


Figure 2. The abstract browser model.

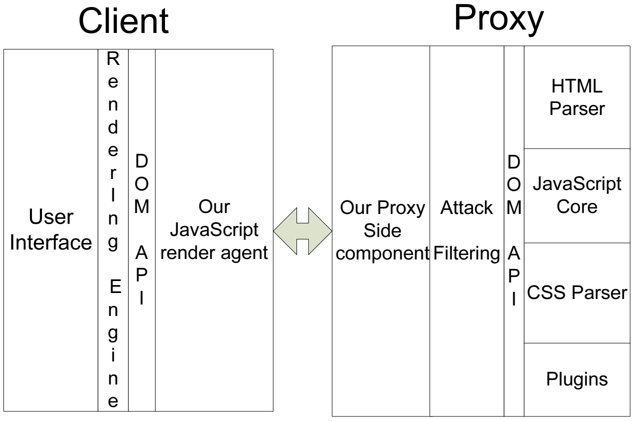


Figure 3. Illustration of the basic idea.

Shield proxy are connected to LAN and WLAN networks, so that the bandwidth to the proxy server is not a bottleneck.

Furthermore, it is assumed that the network administrator can shut down any malicious web user once detected. Finally, we assume that most web users are benign. In other words, we do not consider the possibility that a large number of web users will launch a DoS attack against the proxy.

To demonstrate that security protection mechanisms can be easily deployed with WebShield, we target to *detect drive-by-download attacks as an example*. In the paper we will show how both a behavior-based detection engine and a vulnerability filter-based detection engine can be easily incorporated into WebShield as add-ons.

2.4 Browser Model

In Figure 2, we show an abstract model of a browser. A browser has an HTML parser, a JavaScript engine, and a CSS parser. A browser may have one or more plugins. When the browser receives an HTML page, the HTML parser will parse the page and identify the JavaScript code in <script> tags. The identified JavaScript will be sent to the JavaScript engine. Through `innerHTML` or `document.write`, the JavaScript engine can also call the HTML parser. CSS content is identified by the HTML parser and is sent to the CSS parser. Similarly, JavaScript also has APIs to add CSS rules. The HTML parser, JavaScript engine and CSS parser call the DOM APIs to update the DOM data structures and render the webpage on the UI. According to the standardization organization W3C, "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents." [11]

We take a generalized DOM definition. We consider all the APIs with UI effects as belonging to the DOM. Therefore, under our definition, we consider the `Window` object of JavaScript, which represents the browser window, also as a part of the DOM. In a real browser, some non-standard APIs between the HTML parser (or the CSS parser) and the DOM data structures may be provided for optimization purposes, but they can be substituted by W3C standardized APIs as well. In other words, we can use W3C DOM APIs to fully reconstruct the DOM data structures, and thus to fully reconstruct a webpage.

2.5 Basic Scheme

Given the four principles listed in Section 1, we explore the design space of middleboxes. One possible solution is to work at the graphic rendering layer as in the case of X11 or VNC. The advantage of this solution is that it will run almost all browser modules at the middlebox. However, it is hard to maintain the same user experience, especially for video content embedded in webpages, which will introduce large graphic rendering and network overhead. Furthermore, it is hard for a user to upload/download files from/to their local machine directly because the browser session is entirely remote. We believe this solution has its value and may be beneficial in some circumstances.

However, in this paper, we propose an alternative approach. Our design works at the DOM data structure layer. The encoded DOM data structures are rendered at the client side by our JavaScript agent, while a *shadow browser* inside the middlebox takes care of the rest. Our design is based on the following observation. Bugs in different browser components enable attackers to execute malicious code, and almost all attacks require JavaScript execution for exploitation. Fully eliminating JavaScript execution of untrusted scripts at client browsers will not only make vulnerability exploitation harder, but also close the door for bypassing detection through non-determinism or user-input triggered execution.

Browser quirks are the parsing deviation from the W3C standard. Since attackers have already leveraged browser quirks in HTML and CSS parsing to inject malicious scripts [18, 33], to process HTML or CSS at client browsers might give attackers chances to circumvent our system and to inject malicious scripts. We handle the HTML parsing

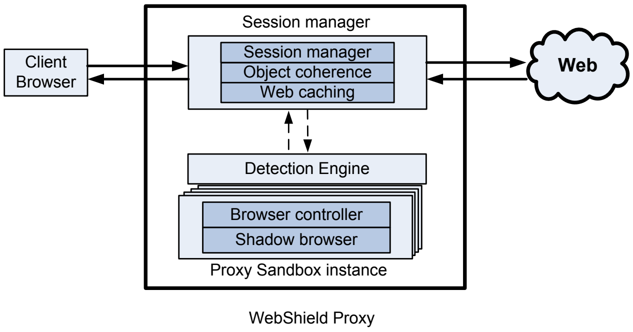


Figure 4. The framework of WebShield.

and the CSS parsing along with JavaScript execution together to the shadow browser. Although the DOM handling and rendering still occur on client browsers, they are deterministic and have less ambiguities. Therefore, If we make sure no attack is detected for the DOM handling and rendering on the shadow browser, the content should be safe to send to client browsers. W3C has standardized enough APIs that can be used to fully construct the DOM data structures using JavaScript. Given this fact, we leverage the client browser for DOM access, rendering engine and user interface. In some sense, we split the functionalities of a web browser. Figure 3 shows the design.

In summary, the reasons we choose DOM as the intercept layer are as follows. Although the lower layer the better, graphic rendering layer may impose higher overhead and be less user-friendly. HTML and CSS processing are at higher layer than DOM and can employ the DOM API to achieve. Moreover, direct HTML and CSS processing might not be very safe due to browser quirks [18, 33]. Even we include direct HTML and CSS processing, still we need the DOM API for dynamic contents.

3 WebShield Design

3.1 WebShield Architecture

To enable our basic design scheme, we propose the architecture shown in Figure 4. The *session manager* manages the web sessions, object coherence, and web caching. If a user sends a non-HTML request (decided based on content sniffing), the session manager will directly respond with the object to the user. For actual webpage requests, the session manager assigns a proxy sandbox instance for each client IP address. A webpage will also be rendered by the shadow browser in the proxy sandbox, and the detection engine will invoke the security protection mechanisms plugged in our system for security detection and prevention.

Next, in Section 3.2, we will first introduce content sniffing in WebShield which classifies contents from the web server into HTML contents and non-HTML contents. Then we will describe how we deal with HTML and non-HTML contents in Section 3.3 and 3.4 respectively. At last, we introduce our sandbox mechanism.

3.2 Content Sniffing

With the middlebox, we need to return transformed DOM objects for all of the HTML pages to the client browser, but directly return all the non-HTML objects such as images and videos. As a result, we need to exactly know which objects are HTML objects. Usually, the browser will determine the type of an object based on the MIME type specified in the HTTP Content-Type header, such as image/jpeg. To improve compatibility, browsers also leverage content sniffing [13] (*i.e.*, check the first n bytes of content) to further identify the MIME type for the object. Barth *et al.* [13] mention that different browsers may implement content sniffing differently and they have successfully extracted the content sniffing models for major browsers. Leveraging their research in our design, we first identify the versions of client browsers based on the User-Agent HTTP header and then apply the corresponding content sniffing models.

3.3 Processing HTML Content

There are two procedures for handling HTML contents: initial HTML page transforming and dynamic HTML support. We summarize how each of these two procedures works in Section 3.3.1. Given these two procedures are similar, and both can be described by a sequence of the same steps, in Section 3.3.2, we break down these two procedures into four steps, and introduce them one by one. Essentially, these steps form the basis of the two procedures.

3.3.1 Two Procedures of Processing HTML Content

When a user requests an HTML webpage, the rendering process has two major procedures: (a) the initial HTML page transforming, and (b) Dynamic HTML Interaction Support.

Initial HTML page transforming. Louw *et al.* [33] argue that HTML parsing and CSS parsing have parsing ambiguities (browser quirks), which can be easily abused to include malicious JavaScript code. In our design, we have decided to transfer encoded DOM data structures (generated by shadow browser execution) instead of transferring HTML or CSS sources directly or after encoding them.

The HTML page, embedded JavaScript files and embedded CSS files will be parsed and executed to create the DOM data structures of the webpage. The DOM data structures include the DOM tree, which defines the structure

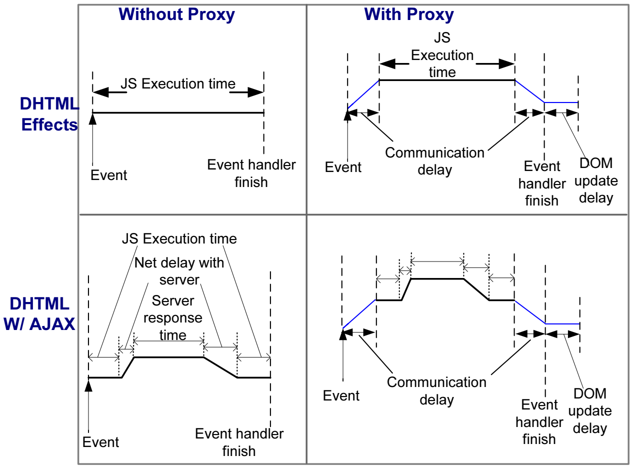


Figure 5. The delay overhead caused by WebShield**.**

of the document, and the CSS objects, which describe the layout and style of the document. We encode the DOM data structures as a list of transformed strings S. Instead of transferring the HTML page, JavaScript and CSS files to the client browser, we transfer an HTML page with our JavaScript rendering agent and S. Our JavaScript rendering agent will parse S and utilize the DOM data structures to render the page. Furthermore, we achieve incremental rendering by taking advantage of the fact that client browsers can incrementally render an HTML page. We show how we implement this and give an example in Figure 6. When we encode the DOM data structures, we remove all the content of the <script>, <style>, and <link rel=stylesheet > nodes; instead, we put empty nodes solely for maintaining the DOM tree structure. CSS is inserted using the DOM-CSS API. If the original HTML page is R, we call the new HTML page we transfer to the client browser $T(R)$. T is the transforming function that transforms the original HTML page as well as its embedded JavaScript and CSS files to a form that includes the JavaScript rendering agent, and the encoded DOM data structures of the original page rendering result.

Dynamic HTML Interaction Support. Dynamic HTML webpages use the event-driven programming model of JavaScript. The JavaScript code from the website may register a set of event handlers for different events. When an event fires, the corresponding event handler is called. In some cases, the JavaScript code may issue an AJAX request to the web server to get more data, and based on the data, update the DOM data structures. In our design, we do not take the risk of running JavaScript on the client browser. Instead, we substitute the event handler from the website to a default event handler written by us. In our event handler, we wrap the event object and transfer it to the shadow browser on the proxy. The same event will be injected to the page on the shadow browser, and the original JavaScript event handler from the website will be executed. After that, the changes to the DOM data structures will be transferred back, and our default event handler will update the DOM data structures on the client browser to achieve the same effect. In Figure 5, we show the extra delays caused by our approach. Mainly, we introduce additional *communication delays* to communicate with the proxy and additional *DOM update delays* to change the DOM data structures. We show two cases: the first row is the case without AJAX calls and the second row is the case with AJAX calls. The left side is the original timing graph, and the right side is the timing graph with WebShield.

3.3.2 Breakdown of the HTML Content Processing Procedures

Handling HTML contents can be divided into four steps: encoding DOM data structure, transmitting DOM updates, update DOM at client browser, and transmitting client events and DOM update back to shadow browser. The initial HTML page transforming requires the first three steps, and the dynamic HTML support requires all the four steps.

Step One: Encoding the DOM data structure. As we have mentioned, HTML and CSS parsing can potentially be abused by exploiting parsing quirks [18, 33], *i.e.*, by using unknown parsing behaviors of a browser to hide malicious JavaScript code. Because we do not want any JavaScript code from websites directly reaching the client's browser, we face the same problem as well. Louw *et al.* [33] propose to directly transfer the parsed DOM data structures instead of JavaScript. We take a similar approach. The major difference is that they only need to transfer small pieces of untrusted content blocks, but we need to transfer the entire DOM tree, all CSS style objects, and any dynamic updates to them– a much more challenging task.

We must transfer two pieces of the DOM data structures to the client browser: the DOM tree DT and the CSS style objects DC. For the dynamic HTML, we need also to transfer the changes to the DOM data structures: ΔDT and ΔDC.

From the client browser's point of view, the DOM tree and CSS style objects are all JavaScript objects. We need a way to serialize the JavaScript objects for interactions between the client browser and the shadow browser. Ideally, we want to use a simple serialization protocol, which itself will be subject to fewer parsing quirks than the HTML and CSS standards. With this goal in mind, we find JSON (JavaScript Object Notation) [5] to be a good candidate for our purpose. JSON is a standard for transferring structured data for web applications. JSON is very simple and

```
<!--eyJkYXRhIjp7fSwidHlwZSI6InN0eWxlU2h
lZXRzIiwiYWN0aW9uIjoiYWRkIiwibG9jYXRpb2
4iOltdfQ==--><script
id="DOM1"> __dp.apply("DOM1");</script>
```

Figure 6. An example of the blocks used for DOM updates

has much less ambiguity. Currently, Firefox 3.5, Chrome 3.0 and Safari 4 all have fast native JSON parsers, making JSON appealing in terms of performance. To avoid malicious content sent to the JSON parser, we also transform all the string properties of the DOM data structure. We choose the escape and unescape functions in JavaScript for this purpose. The functions escape all the JSON control characters to %XX form, and do so in a fast operation. Moreover, since both communication endpoints are controlled by us, it is much harder to subvert the protocol for malicious purposes.

DOM nodes are the internal representation of an HTML tag in the browser. For example, Element nodes can have child nodes. Essentially DOM nodes are objects, and the child relationships are the references between the nodes. Based on JSON, we define a protocol to serialize a DOM subtree constructed by DOM nodes. We have defined DOM subtree addition, deletion and updating primitives. The references to the objects have been changed to refer to the ID of nodes.

For CSS, after parsing a CSS rule, a browser will create a corresponding style object to represent the rule internally. JavaScript has an interface to create, read, and write the style object. In the client browser, we can create an empty rule (an empty style object) and then assign it properties. With this way, we can avoid parsing the CSS rule. We still use JSON to serialize the style objects for communication.

Next, we will introduce how the JSON encoded DOM updates are transferred during the initial page rendering and during dynamic HTML interactions.

Step Two: Transmitting DOM updates to client browser. We adopt two approaches to transmit DOM updates. In the initial page rending stage, we will encapsulate them in a HTML page. After that, we will use AJAX to transmit DOM updates.

- **Encapsulating the DOM updates in a HTML page during the initial page rendering:** The performance of the initial page rendering is important for web users. To take advantage of the incremental rendering available in all the major browsers, we embed the JSON encoded DOM updates in the return HTML page directly, instead of sending AJAX calls to retrieve them. However, JSON strings may interfere with the HTML parser. We need another layer of encoding to avoid this side effect. We use base64 encoding, because all the major browsers have implemented fast and native base64 encode functions (a2b, b2a), and base64 will not interfere with the HTML parser [33].

  When loading a page, the shadow browser will render the page incrementally. We monitor the DOM tree DT and CSS style object DC changes. Whenever a part of these data structures become available, we encode and transfer the ΔDT and ΔDC. In Figure 6 we show an example of an encoded block. When a block arrives at the client browser, a JavaScript function will be executed and the JavaScript function will delete the comment node and the <script> node, delete the DOM update, and replace it with the actual DOM data structures. This way, the client browser can also render the page incrementally.
- **Handling Dynamic HTML effects using AJAX:** Later on, after the initial page rendering, the DOM updates will be transferred through AJAX. Although the AJAX call is for the proxy, because of the Same Origin Policy (SOP) in modern browsers, we have to send the AJAX call to a URI destined on the original web server. AJAX XMLHTTPRequest allows us to add HTTP headers. We leverage this to add a special HTTP header with the session ID of the webpage to let the proxy know the AJAX request is an internal POST message. The proxy will not forward the request to the website; instead, the shadow browser will process and then reply to this request.

Step Three: Updating DOM at client side. As we have mentioned, all the HTML tags form a DOM tree, which is the internal representation of the HTML document. For a dynamic UI effect, parts of the tree are changed. Formally, a set of subtrees in the DOM tree may be changed. The change can be an addition, a deletion or an update. To sync the DOM trees between the shadow and client browser, we need to locate the root nodes of the subtrees in the DOM tree. We implement two solutions to resolve this issue.

For the first approach, we label each element DOM node with an ID attribute on the client browser. If the original DOM node on the shadow browser has an ID attribute, we just reuse the same value of the ID attribute. If the original DOM node does not have ID attribute, we create a unique value for the ID attribute. At the client browser side, some CSS rules may use the ID attribute, so maintaining the same value of the ID attribute as in the original DOM node is necessary. At the shadow browser side, we need to add another private DOM node property. We call it `MyID` because we do not want to add the ID attribute to the DOM nodes which did not have an ID attribute earlier, thus breaking the JavaScript from the website. The ID attribute in the client side DOM tree is mapped to the `MyID` property in the shadow browser side DOM tree. GetElementById and GetElementByMyId are used in the client browser and the shadow browser to locate the node. `MyID` and its related APIs are not visible in JavaScript on shadow browsers, so that webpages cannot use these to detect whether it is running inside a shadow browser.

Because, comment DOM nodes and text DOM nodes cannot have ID attributes, we must also use a second approach. A DOM node can be uniquely identified by its coordinates in the tree structure. In general a node in the DOM tree may have n child nodes. The index $i \in [0, n-1]$ can be used to identify a specific child node. We can do this recursively using a vector of the location index to identify the path from the root node to the specified node.

In our design, we use the location system which is the most convenient for the node at hand.

Step Four: Transmitting Events and DOM updates back to shadow browser.

For Dynamic HTML effects triggered by user events, we pack the event and changes in the DOM data structures at the client browser, and send that to the proxy through an AJAX post message. The shadow browser on the proxy will then inject the event into the page and apply the appropriate changes to the DOM data structures. The event will trigger JavaScript to run. Finally, the shadow browser will reply with the DOM updates made by the JavaScript code to the client browser. When using AJAX, we do not need to use base64 encoding for the JSON message in either side of messages, because the message is treated as a string. Furthermore, we do not need to use browser related parsers for such messages.

For every DOM node that accepts user inputs, such as <input> and <textarea>, we register an "onchange" event handler that stores the value of the user input in a global buffer in JavaScript. For the DOM nodes that have event handlers registered in the shadow browser, we register the default event handler that transfers the event and all the changes of DOM nodes in the global buffer to the shadow browser through an AJAX call. The shadow browser will reply with the changes to the DOM data structures triggered by the event.

3.4 Processing Non-HTML Content

The non HTML embedded content such as Flash, images and videos are returned to the client browser directly upon request from the client. The same content is still rendered at the shadow browser so as to detect any exploits that may appear while rendering these objects. Some non-HTML content may still be scriptable. It is better to transform them before sending them back to client browsers. Several techniques have been proposed to transform flash or Java applets [22, 25]. Working with these techniques together, WebShield will provide better security, as discussed in 8

When we render a webpage on the shadow browser, all non-HTML objects will be requested by the shadow browser. After we transfer $T(H)$ to the client browser, the client browser will also request the objects excluding the JavaScript and CSS files, because those files are part of $T(H)$. One problem that arises here is that an object e will be requested twice (once from the shadow browser and again from the client browser). This may have serious effects on the web application. Actually, all middlebox designs, including SpyProxy, which run browser instances, may encounter this problem.

A design trade-off we need to consider is whether the two browsers can request the same object e independently. For cacheable objects, with the web cache, the first request will go to the web server, and the second request will be returned by the web cache. This is actually the policy used by SpyProxy [23]. However, this may cause problems for dynamically generated, non-cacheable objects. In such cases, both requests will go to the remote web server. This can be harmful when the requests change the persistent state on the web server. A simple example is a visit counter image. Given that this can sometimes lead to serious problems, we enforce the rule that only one of the two requests for the same object can reach the web server. To achieve this, we have to accurately identify the pair of requests for the same object. Ideally, if each non-cacheable object has a globally unique URI, it will be easier to pair the requests. However, in practice this might not be true. For instance, it is hard to enforce the user to not open two identical windows to render the same page.

Because of the object coherence problem, we propose to add a unique identifier to every embedded URI. With the identifiers, we can separate the URIs that are directly typed by users from the embedded URIs, and we can also accurately identify the pair of requests for the same object. In our current design, we use a 256-bit identifier. The first 96 bits are a unique random string to avoid collision from other possible user-inputted URIs. The middle 128 bits are the web session ID. Finally, the last 32 bits are used to differentiate the embedded URIs in a page. The 256-bit identifier is encoded into a URI safe string and attached to the end of each URI. When constructing the DOM data structures on the shadow proxy, we rewrite the URIs to append the identifiers. At the session manager, the embedded URIs will be identified and paired up.

3.5 Sandbox of the Shadow Browser Instance

Similar to other security prevention schemes that require dynamic execution of the suspicious content, WebShield needs to use sandbox techniques to make sure that even when an attacker compromises the shadow browser, he still cannot compromise the physical host running the shadow browsers, let alone the client browser or the client machine.

For this purpose, any state-of-the-art sandbox techniques can potentially be applied with different tradeoffs on security, performance and stability. Our design is not tied to any particular sandbox technique. Potentially, we can apply the best available techniques, such as SELinux, Xen virtual machines (VM), VMware VM, etc. In our current implementation, we focus on SELinux.

4 Security Analysis

4.1 Subverting the Sandboxes

In an actual deployment setting, WebShield will have one session manager running on a separate host M and a set of hosts H for hosting the sandboxes. In our design, for any host in H, the system administrator can configure the switch and force it to only communicate with the proxy service on M. The hosts in H cannot communicate with each other, the Internet or the internal hosts of the enterprise network unless going through the session manager.

To compromise a host in H, the attacker needs to first compromise the shadow browser, which means he must bypass the detection of known vulnerabilities as well as the behavior detection engine (step I). Then, the attacker needs to exploit another vulnerability in the sandbox to escalate his privileges and take control of the host OS or VMM (step II). After step I, the attacker can control one shadow browser, and after step II, the attacker can control all of the sandboxes and shadow browsers in a physical machine. In both cases, he needs to exploit the session manger to take full control of the proxy. We believe the session manager should be written in a type-safe language, which will make the control flow hijacking exploitations much harder. This will also make the proxy safer.

By taking control of the shadow browsers, the attacker can also try to send malicious HTTP requests to the Internet through the proxy to compromise other web servers. Since the attacker can send such malicious requests without compromising the shadow browser, we do not believe this enhances the attacker's power.

After compromising a shadow browser, an attacker can also try to compromise enterprise web users by returning malicious content for DOM update requests. In our design, all of the JavaScript code in the returned webpage, *i.e.*, our JavaScript code and the <script> tags for each update block, is only added by the session manager, but not the shadow browser. Shadow browsers only provide encoded DOM updates in JSON. The attacker has to exploit a vulnerability in the JSON parser in the client browser to execute JavaScript directly by the JSON parser. After JSON parsing, but before adding parsed content to the DOM tree or CSS style objects, we ensure all <script> tags are empty, and no event handler attributes are present. Therefore, it is still nontrivial for attackers to compromise web users or even control the corresponding shadow browser.

4.2 Potential DoS Attack

In our threat model, we do not consider the case where web users will launch a DoS attack on the proxy. We limit the number of dynamic HTML webpages that can be concurrently opened from a single IP. (Note that the limitation is for a DoS attack. For a legitimate user, the number is large enough for him to use.) For static webpages, we do not maintain a long lived webpage in the shadow browser, reducing resource consumption. This will prevent a few users from overwhelming the proxy by opening a large number of pages.

Some pages may also open more webpages automatically using JavaScript. For this case, we will not create the window on the shadow browser directly. Instead, we will send the open window request to the user. Normally the client browser will block it and ask permission from the user. If the user allows the pop-up, an AJAX request will be sent to the proxy, and the shadow browser will allow the window to open and transfer the transformed content to the user. We also limit the number of such pop-up pages for a given web user (Similar methods has already been adopted in modern browsers, such as Firefox, which limit the number of pop-up windows).

4.3 Fingerprinting the Shadow Browser

Some methods can be adopted for the webpage running inside the shadow browser to fingerprint the environment. For example, the webpage can detect the browser's version number, support of functionalities, etc. However, it is still hard for attackers to decide whether the webpage is running in a shadow browser or in a client browser directly. Even if they can detect that the webpage runs in a shadow browser, they still cannot exploit the client directly. Moreover, they cannot probe the browser version of a client browser directly, because we do not allow any JavaScript from the webpage to run on the client browser. Therefore, it is non-trivial for the attackers to leverage on the possible version difference between the shadow browser and the client browser for attack.

4.4 Compatibility with Other Security Protection Mechanisms

In our design, we explicitly consider compatibility with other existing security protection mechanisms.

Same Origin Policy: In our design, we do not break the same origin policy. We keep the origin of each website unchanged. When we rewrite URIs, we do not alter the parts related to the origin.

Host Anti-virus Protector: Currently, many anti-virus systems add security plug-ins to the browser to enhance user protection. Since the DOM data structures are almost identical to the originals, WebShield will not influence anti-virus scanners.

5 Implementation

5.1 DOM Instrumentation

At the client browser side, we leverage JavaScript DOM APIs to update the DOM data structures. We use the DOM-CSS interface to add the CSS style objects for the CSS rules. We also use the `element.style` object to handle the inline CSS style rules.

To implement the shadow browser, we modify WebKit revision 41242 [10]. We instrument the DOM interface of WebKit in C++. Once there is an action in the DOM, e.g., appendChild, we detect such a change in the DOM and process it accordingly.

5.2 Session Manager Implementation

In our current prototype, the session manager is implemented in Python. We use the HTTP `client.py` and `server.py` files from Python 3.1. We implement web caching, object coherence and session management. For Web caching, we follow the cache related HTTP headers. For session management, we assign a session ID to each HTML object. Later, the session ID will be used to identify the AJAX call for the event proxy.

5.3 Sandbox Implementation

There are many possible choices for a sandbox environment. Generally speaking, we consider two factors, performance and security. Tahoma [16] uses Xen, a VMM which has good security protection but the overhead is quite large. OP Browser [17] and Google Chrome [32] use process level sandboxing, which has good performance, but weaker security. In our current implementation, we adopt SELinux, the same sandbox used by OP Browser.

In the TE model of SELinux [27], an *object*, for example a program, is assigned a *type*, which has limited access privileges to resources within itself and other objects. We assign a different type to every new sandbox we create. We then provide the minimum resources required by the sandbox. If the processes inside the sandbox tries to access a certain resource for which they does not have permissions, the event will be logged in our system, indicating contamination of this sandbox. For example, the processes in a sandbox tries to access a user file with type `user_t`, which requires having the privilege `user_t:file read`. Since the processes do not have the proper privileges, access will be denied. This denial will appear in SELinux logs giving us a means for detection.

5.4 Drive-By-Download Detection

To demonstrate the usefulness of the WebShield framework, we implement two types of detection engines for drive-by-download attacks: a policy-based engine to detect known vulnerabilities and a behavior-based engine for unknown vulnerability detection.

The policy engine for vulnerability filtering. BrowserShield [31] leverages HTML and JavaScript rewriting to add an interposition layer to check the invocation of DOM APIs and malicious HTML tags. Since we are able to modify the shadow browser, we directly insert a security checking layer between the DOM APIs and the HTML parser, CSS parser and JavaScript engine. Therefore, we can filter out all the DOM nodes or CSS style objects that will potentially trigger the vulnerabilities before encoding the DOM data structures and sending the DOM data structures to the client browser. This way, we "purify" the webpage and display the remaining safe parts to the end users. The end users can still access the important information in the the webpages without any problems.

For the policy engine, we primarily add a security checking layer at two places. The first place is the JavaScript API binding. Whenever JavaScript tries an API call, such as the DOM APIs, we will capture the invocation. We then check whether the parameters to the APIs will trigger any known vulnerabilities. The second place is the HTML and CSS parse trees. The vulnerability checkers can be written as a C++ function. We provide APIs for writing such checkers. Our present implementation is a regular expression checker, which checks each passed string using a signature library by regular expression.

The behavior engine for detecting unknown vulnerabilities. Usually, the goal of drive-by download attacks is to exploit the victim's browser, and let the attacker install and run arbitrary software on the victim's computer. In [23, 24, 28, 29, 36], a behavior based model is used to detect drive-by-download attacks toward unknown vulnerabilities. The basic idea is that any abnormal behaviors that violate the browser security model will be counted as attacks. In [23, 24], Moshchuk *et al.* give a list of abnormal behaviors, such as attempts to create a new process that does not belong to the browser, modifications to the file system other than the cache folder, browser/OS crashes, etc. We implement a similar behavior detection model on SELinux. We mainly rely on two mechanisms: the SELinux log and a process monitor. The SELinux log detects a potential normal profile violation, including attempts to execute a different binary to create a process, etc. The process monitor monitors whether the process has crashed, or uses too much memory, etc. When either of these two reports a problem, we will consider it as a vulnerability.

5.5 Implementation Summary

We add 6000 lines of C++ code to WebKit in order to construct the proxy-side sandbox, with 200 lines used to inject the DOM interface. Session Manager also contains 3700 lines of Python code. The client side program contains 722 lines of JavaScript code.

6 Evaluation

We evaluate WebShield with seven different metrics: (i) compatibility of representative webpages with our implementation, i.e., how accurately webpages through our proxy render at the client side, (ii) the latency overhead, (iii) the communication overhead, (iv) the memory overhead, (v) the interactive performance for dynamic HTML, (vi) scalability and (vii) accuracy of the detection engine. The first five metrics are simply to evaluate how transparent WebShield is to the user. Then, we discuss how well our system scales. Finally, as a demonstration, we show the accuracy of the detection engine plugins for detecting drive-by-download attacks. The proxy server was installed on a 2.5 GHz Intel Xeon server with 16 GB RAM running CentOS 5. For the client, we used a 2.66 GHz Intel Core2Duo machine with 3.25 GB of memory and running Windows XP SP3. For some tests using the Safari browser, we used a 2.2 GHz Intel Core2Duo based MacBook with Mac OS X version 10.6 and 4 GB memory as the client side machine. The client machines and the server were connected on Gigabit Ethernet. Unless indicated otherwise, the client browser used is Google Chrome. Next, we discuss the evaluation results.

6.1 Compatibility Tests

In this section, we evaluate whether webpages render correctly when viewed through the proxy. The notion of rendering correctness is relative to the rendering without going though our proxy. For all of the webpages, we manually test the visual correctness of the rendering.

We note that not all of the websites are actually entirely compatible with our current implementation of WebShield. 91 of the top 100 websites and 19 of the top 20 as given by Alexa were compatible, and the rest have some rendering issues. The webpage at `http://www.aol.com` cannot be rendered by the WebKit version we used for the WebShield implementation. For the remaining websites, the reasons are mainly implementation stability issues (crashes) and the websites using unsupported features such as iframes with the HTTPS protocol.

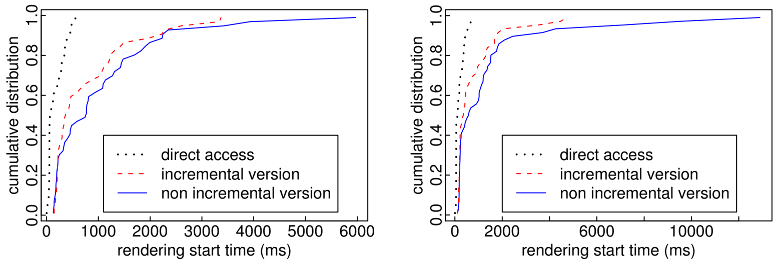
6.2 Latency Overhead

The timing overheads are computed as the latencies between the start and finish time of page rendering relative to the page request time. We selected the webpages passing the compatibility test and rendered each of them in Firefox, Chrome and Safari and measured the rendering start and finish times. Due to the limited space, we show only the results for Firefox and Chrome here; the results for Safari are similar. When accessing the pages through WebShield, we used JavaScript functions to get the start and end times. The browser may issue the onload event before the page response has already completed. So we report rendering finish time as the maximum of HTTP response end time and the onload event time. When directly accessing the webpage without using WebShield, we use the page response start and finish times to approximate the rendering time. Each URL is rendered five times and the medians of individual results of these runs are used as the rendering latencies for the URL. For this metric, we do a detailed evaluation for two versions of WebShield: incremental and non-incremental. The incremental version transmits a webpage part by part to the client browser once it has rendered these parts in the shadow browser. It does not wait for the entire webpage to be downloaded and rendered before transmitting it to the client. The non-incremental version is a simpler one, presented here only for the sake of a comparison. It transmits responses to the client, only after the webpage has been completely rendered at the proxy.

Figure 7 presents the cumulative distribution of rendering start times. The incremental version sends partial rendered contents to the client starting earlier than non-incremental one. As expected, the incremental version responds earlier than the non-incremental version for more than 50% of the pages.

Figure 8 shows us the cumulative distribution of rendering finish times. We note from the CDF that there is not much difference between the incremental and non-incremental versions. This is easily understood because the rendering end time depends on the last chunk of the page response which depends on the response from the original webserver; both the incremental and non-incremental version end up transmitting this last chunk at nearly the same times. Effectively, the incremental version only helps in improving the responsiveness of the webpage and not in the net load time.

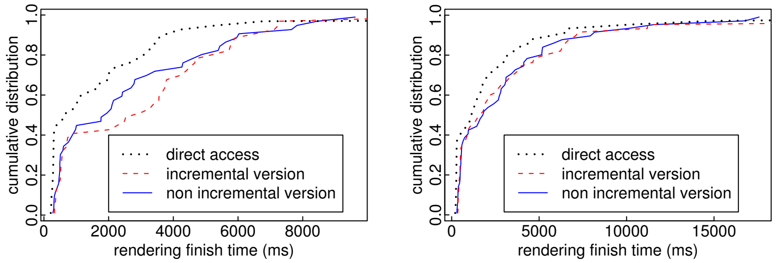
To summarize the comparison between access through the incremental version of WebShield and direct access, we present the following numbers. The median difference of rendering starting latency is 133.5 milliseconds. For 90%



(a) Firefox

(b) Chrome

Figure 7. Cumulative Distribution Function of Rendering Start Time



(a) Firefox

(b) Chrome

Figure 8. Cumulative Distribution Function of Rendering End Time

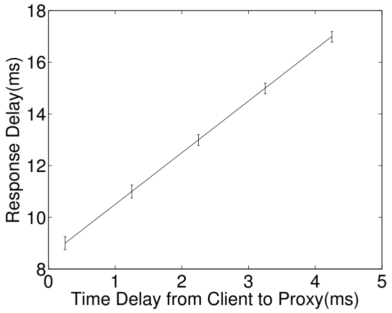


Figure 9. Communication Delay with Different Local Latencies

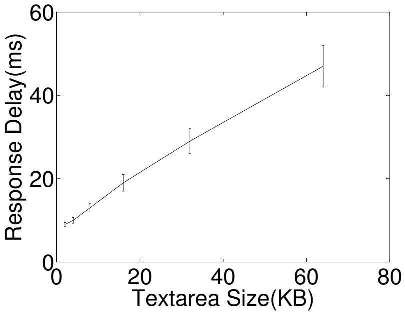


Figure 10. Communication Delay with Different Msg Sizes

of the cases the difference is less than 1.083 seconds. For the rendering finish times, the median difference is 382 milliseconds and the 90 percentile cutoff comes at 2.459 seconds. For a few rare cases, the difference is larger than five seconds; we attribute those rare cases to some implementation problems in our current prototype.

6.3 Communication Overhead

Communication overhead is calculated as the data transferred over the network to and from the client. For a webpage, we captured the network traces when accessing the webpage with and without our proxy and obtained the sizes of those traces.

Figure 11 depicts the communication overhead. WebShield does not always have larger communication overhead comparing with direct access. For some of the webpages, we see that the amount of data transferred to the client when going through WebShield is less than direct access. On the one hand, Direct access needs to transfer HTML content, JavaScript files, and CSS style sheets, whereas WebShield needs to transfer encoded DOM updates related to visual affect, such as a text area, visible elements, and so on. Our transfer of user-visible components is not as effective as that of direct access because we transfer parsed data in JSON, which is less dense than HTML notation, and during incremental rendering we need to provide location information and other tokens. On the other hand, however, we do not need to transfer JavaScripts, which contain application logic, because JavaScripts are executed by the JavaScript engine of shadow browsers in WebShield and only the final results are sent back to users. This reduce the transfer overhead. As seen in the Figure 11, there are mixed results when comparing WebShield with direct access. We incur more overhead on Youtube as compared to direct access because there are few script tags in Youtube, but for Google Maps, we achieve smaller overhead because there are many JavaScript scripts.

6.4 Memory Overhead

To evaluate memory overhead, we selected ten complex webpages. For every webpage, we report the memory overhead as the difference in memory usage of the browser before and after the page load. The initial and after the page load memory usages are gathered using the Windows Task

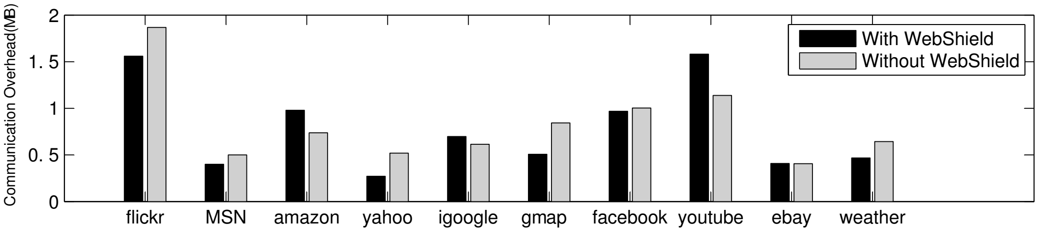


Figure 11. Communication Overhead

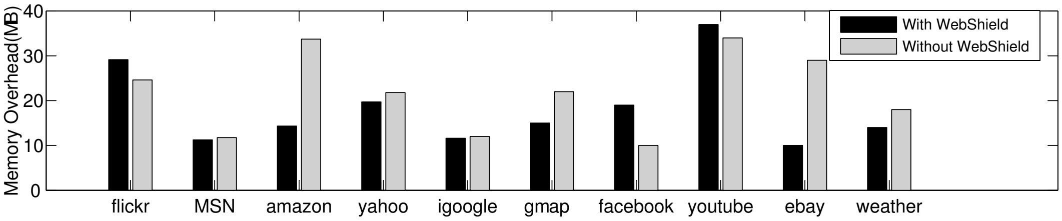


Figure 12. Memory Overhead

Manager. While this approach is only approximate in the sense that we do not gather the data exactly before page request and after page load, but wait for the Task Manager to show the results, we find this approach to be quite reasonable as we received consistent results over multiple loadings of a webpage.

In Figure 12, we compare the memory consumption of the transformed webpage with the corresponding native webpages. For our system, memory is consumed mostly by JavaScript and DOM, because we use JavaScript to reconstruct DOM. For a native webpage, there are many places, for example the HTML parser, CSS parser, and JavaScript engine, which consume memory. If a page is very large and contains many elements, parsing the native web pages can use a lot of memory. If a page is fairly small, our JavaScript program itself may take up a fairly large amount of memory compared to the webpage itself.

Figure 12 shows that our memory overhead is just a little higher than those of native webpages, which means the memory usage of our JavaScript program is nearly the same as native webpages. We look at some webpages as examples. We have more memory overhead on Facebook because the native Facebook page has a few elements for parsing and our JavaScript render agent on client side will take comparatively more memory. But, for Amazon, we have less memory overhead for the transformed webpage than the native page. This is because Amazon has many user effect elements and the browser will consume a lot of memory to parse the contents.

6.5 Interactive Performance for Dynamic HTML

As shown in Figure 5, the normal JavaScript event response time is the time for a browser to execute the JavaScript event handler bound to the special event. If the JavaScript event triggers an AJAX call, an AJAX connection to the remote web server will occur, and thus the network round trip time as well as the server response time need to be included in the respond time. This cost is inevitable in any client-server system. The extra time cost is the cost introduced by our secure proxy.

In this section, we present interactivity evaluation with microbenchmarks and a real-world JavaScript game. In the microbenchmarks, we will investigate the extra time cost. Regardless whether there is an AJAX connection to the remote server, the extra event response time introduced by WebShield can be classified as *communication delay* and *DOM update delay*. *Communication delay* is the time cost for transferring the triggered events from the client to the shadow browser in the secure proxy and transfering back the result (DOM updates) from the shadow browser to the client browser. *DOM update delay* is the time cost for updating the aforementioned shadow browser DOM updates into the webpage's DOM in the client browser. For the JavaScript game evaluation, from the user's perspective, we measure the time delay caused by the user's actions.

6.5.1 Microbenchmarks

With microbenchmarks, we measure the atomic event transmission delay. We write a test webpage to test the event response time. In the testing webpage, the user can enter

	Start Game	Move Mouse	Drop a Piece	Game Over
Additional Delay	41ms	7ms	10ms	7ms

Table 3. Time Delay in Game `Connect 4`

texts into a textarea. After the user clicks the button on the client browser, the text in the textarea will be updated to a text node inside a `<div>` node. With our security proxy, when the user clicks the button, the changes in textarea node and the "click" event will be packaged in JSON and transferred to the shadow browser in the proxy. The shadow browser updates a textnode with the receiving textarea content. Then, the changes of the DOM tree will be sent back to the client browser and the client-side DOM is updated.

Communication delay: The communication delay time is decided by two factors, the network delay and the data transfer size. Network delay will influence the *communication delay*. We use the network emulator `netem` [9] to emulate different network environments with latencies of 0, 1, 2, 3 and 4ms on the gateway. Here, we change the gateway latency, and trigger the click event with a 2 kilobyte textarea. As seen in Figure 9, the response delay increases linearly with the growth of local latency. The response delay is within 20ms when local latency is 4ms. So the user will not perceive this extra delay.

Then, we will test the communication delay with different DOM node sizes. As Figure 10 shows, we change the length of textarea in a test webpage, and trigger the click event while measuring the delay time with the Date() function. We can see that the response delay increases linearly with the growth of textarea size. Even as the textarea reaches 64 kilobytes, the communication delay is still within 50ms.

In an intra-network environment, the latency between the client browser and proxy is usually small (less than 5ms). As the above evaluation shows, the event response time is just tens of milliseconds, which will not affect the user's experience.

DOM update delay: In the above evaluation, we update one node for each event. Here, we evaluate the time cost for updating one DOM node. In our experiment, we repeatedly insert a textnode with a length of 2 kilobytes to the first child of body tag 10000 times, and use the Date() function to get the duration. The average time cost for updating a single node is 0.04ms. For state-of-the-art web applications, one event usually incur an update of less than 100 DOM nodes. Therefore, *DOM update delay* is negligible in our implementation.

6.5.2 Test on a Real Game

We chose a real-world high-interactive JavaScript game, Connect Four, to benchmark WebShield. We downloaded this game from the front page of the top website when querying Google for "javascript games". We measure the delay on the client side based on different user operations. Connect Four is a mid-size (about 7K) two-player JavaScript game we found at a popular JavaScript games site [4]. Each player can place a piece on a square. When a player has four connected pieces, he will win. In our experiment, we will measure time delay introduced by our proxy with various user actions taking effect on the user screen.

In this game, four kinds of activities are measured. First, we measure the time difference between when a user presses the button to start the game and when the game is started. Next, we measure the time difference between when a user moves a mouse and when the visual effect is shown on the screen. Third, time delay is the time difference between when the user drops a piece and when the piece is shown on the screen. Lastly, how long it takes for the game's end to be shown when someone has four connected pieces, It is generally acceptable that delays under 200ms are tolerable for an interactive application [15]; As shown in table 3, the delay incurred in our case is much smaller. Therefore, we believe our scheme is good for even high interactive web applications.

6.6 Scalability

WebShield is designed to be used in an enterprise network such as a company, a government agency or a university. We consider the sandbox creation speed and sandbox memory usage for the scalability of WebShield. Sandbox creation speed will influence how many new users that that can be supported in a short amount of time because a new sandbox is created for each new user. This depends on the startup time of a sandbox, which is 0.035 seconds per sandbox in our system. The total number of sandboxes will influence the total amount users we can support because each sandbox will take a certain amount of memory. In our system, one sandbox takes 100MB and one page takes 10-25MB. We use a sample of one day of traffic in Northwestern university to observe the scalability of our system.

At Northwestern university, there are 39 new web users on average in one second and at most 82 new web users in one second. With our sandbox creation speed, 28 sandboxes

Exploit	Description	Behavior Monitor	Signature Detection
1	CVE-2009-0945: SVG null-pointer dereference	Y	N
2	CVE-2009-1690: Use-after-free vulnerability	Y	Y
3	CVE-2009-1698: Incorrectly parsing `attr()`	N	Y
4	CVE-2009-1700: Local file theft	Y	Y
5	WebKit Bug 19588:Crash doing open on destroyed window	Y	N
6	Cross-site scripting across frames	Y	Y
7	Opening an arbitrary amount of windows	N	Y
8	Parsing arbitrarily large integer	Y	Y
	False Positive	0/500	1/500

Table 4. Exploit Description and Detection Results(Y means Can Detect, N means the Opposite)

are created per second. Using the maximum amount of new users, we need 3 computers to support that number of new users. For some margin, no more than five computers are needed to sustain the new user rate.

From our measurement, we find Northwestern University has 2720 live web users on average, and each user visits five websites per minute on average. We assume we use 16GB memory machines to support users. Assuming the OS uses 2GB of the memory, one sandbox uses 100MB of memory, and five webpages count for about 100MB, each machine can sustain 70 users(sandboxes). Thus, we will need 39 machines for a middle size enterprise network such as the campus network of Northwestern university.

Sandbox creation speed is not a big problem for our system because we use process-level sandboxes, but sandbox memory usage is the system bottleneck. However, we believe less than 40 machines are moderate for an enterprise network.

6.7 Drive-By-Download Evaluation

To demonstrate the usefulness of WebShield, we also evaluate the detection accuracy of two detection engines when pluging them in WebShield framework.

As there is no good source of publicly available or popular WebKit exploits, several exploits were written targeting both real, reported vulnerabilities (such as those with a CVE number) and fake one that were introduced into the WebKit engine by modifying its code. With the four real and four fake exploits, listed in Table 4, targeting cross-site scripting, denial of service, and other kinds of attacks, the detection engine was evaluated on its success in detecting the exploits.

There are two components to the detection engine: the *behavior monitor* and the *signature engine*. The behavior monitor keeps track of potentially suspicious behavior such as file accesses, new processes, and browser termination. The signature engine resides inside the web browser and uses regular expressions along with signatures to detect suspicious webpages. When either or both of the components report something, the webpage is considered to have an exploit.

The results of the evaluation are in Table 4. As seen in the table, the behavior monitor and the signature engine both detected several of the exploits. Though each component by itself was not able to detect all the exploits, considering the results from both components led to a full coverage of the exploits.

To test for false positives, the detection engine was tested on the index pages of the top 500 websites listed on Alexa [1], which are supposedly benign. The procedure to evaluate the effectiveness of the engine on the exploits was also used on these top ranked pages to determine if any of these were incorrectly determined to be malicious webpages. As seen in Table 4, the behavior monitor did not consider any of those pages suspicious and the signature engine considered only one of those pages as malicious. Furthermore, the signature engine may be improved by using better, more accurate signatures.

7 Related Work

7.1 Web Attack Defense Techniques

Host-based approaches: Many techniques have been proposed for defending web attacks. The majority of them are host-based approaches which require browser or client-side modification [12, 14, 16, 17, 19, 26, 30, 34, 36].

Several research efforts propose to modify the client browser architecture for better security protection [16, 17, 32, 35]. At a high level, they propose to use proper sandboxes to isolate different instances (browser modules, browser tabs, principals *etc.*), which require browser modification. Their design can be easily deployed in our shadow browsers, which will help improve the security of web users. WebShield is different in that it is not a new browser architecture, but rather a framework for deploying security defense techniques without client modification.

There are also many works proposed for different types of web attacks. HoneyMonkey [36] requires a BHO (browser helper object) and VM monitors for behavior-based drive-by-download detection. Barth *et al.* propose to add an origin header for preventing cross site request forgery [14]. In [12], a reference monitor is proposed to solve cross origin JavaScript capability leaks. Both DSI [26] and JavaScript Taint [34] require taint checking in JavaScript engines to prevent cross side scripting attacks. All these cases require client side deployment. WebShield can help deploy them at a middlebox without client side modification.

Middlebox Approaches: Middlebox approaches are an alternative to host based approached for web defense. Existing middleboxes focus on drive-by-download attack detection. SpyProxy [23] and BrowserShield [31] are two such examples. We have compared with these two in Section 2.1.

Malicious URI Labeling: Many industrial vendors, such as Google [28], McAfee [6], and BlueCoat [2], attempt to statically label URIs as either benign or malicious using [36] or similar approaches. Two major problems exist with this approach. First, similar to SpyProxy, the detection is static, and thus can easily be bypassed with non-determinism or user inputs. Second, attackers can leverage URI polymorphism to make URI based detection harder. We believe that WebShield and URI labeling approaches can potentially be combined together. The URI labeling can increase efficiency while our approach can improve the accuracy.

7.2 Remote Browsing

Malkhi *et al.* [22] propose to run Java Applets in a remote playground and proxy the visual effect back to client browsers. FlashProxy [25] aims to rewrite Flash to an AJAX JavaScript program to display the same visual effect without requiring flash support on client browsers. Our approach is similar to their in philosophy; it works by sending the visual effect in the form of DOM updates back to client browsers. However, to design a proxy for the visual effects of webpages has its own challenges, such as the aforementioned object coherence problem. Moreover, by incorporating these two approaches, we can have a more complete design which can even handle mobile code in plug-ins. Opera Mini [7] is one of the most widely deployed systems that have browsers inside middleboxes. Opera Mini partially renders a webpage and convert to a internal format (*e.g.*, OBML) in a middlebox before transferring it to the Opera Mini client on smartphones. Opera Mini is designed for a totally different purpose and requires client-side modifications. Ripley [20] executes JavaScript at the client and server sides in parallel in order to validate the correctness of the application. Their goal is different from ours in that they aim to prevent malicious users from tampering with the web application logic. Co-browsing [21] also investigates different options for synchronizing two different web browser instances. Due to an entirely different goal they choose a quite different approach by synchronizing the inputs to JavaScript functions. We also need to synchronize the shadow browser with the client browser. However, because we want to prohibit JavaScript execution on the client side, we choose a different way for synchronization.

8 Discussion

Privacy Issues: Similar to many security checking devices, WebShield needs to check the content related to users, primarily the webpages visited by users. We believe WebShield does not raise more privacy issues than traditional network based intrusion detection/prevention systems (NIPSes), which are currently deployed by most enterprise networks. Similar to NIPSes, WebShield examines the user related content through automated programs and filters out the attacks automatically. Therefore, we believe deploying WebShield in enterprise networks will not bring new privacy issues.

Dealing with User Scripts: Bookmarklets and Greasemonkey [3] allow the user to run custom JavaScript on a webpage to do some tasks or change the appearance of the webpage. Since these tools interact with the page's DOM in some way, and our approach keeps the original DOM intact, the behavior of user scripts remains unchanged.

Dealing with Multiple Brands and Versions of Browsers: Different users may favor different browsers. A deployed system should support all major brands of browsers. Therefore, WebShield need to have shadow browsers of major brands of browsers, such as IE, Firefox and Chrome. Most modules of shadow browsers are independent from the actual browser. The browser dependent module is mainly a DOM monitor which can access DOM nodes and events with they are created.

Note that only the most recent versions of browsers need to be supported in the shadow browser collection. Known vulnerabilities are detected by policy-based engines, which check the DOM updates and cannot be bypassed regardless of whether the versions of shadow browsers and client browsers match exactly. Behavior engines are mainly used to detect zero day vulnerabilities. It is very rare to have zero day vulnerabilities only in an older version but not in the current version of browsers. We therefore need to support only recent versions of a handful of major browsers.

Limitations: One major limitation of our approach is: if the event triggered DOM updates happen too frequently, our remote execution potentially cannot keep up with updating the webpage on time. Given the RTT of enterprise is low, only extremely high event streams will cause such problems. We examined top 100 sites from Alexa. None of them triggered such high event streams. Therefore, we believe for most web pages our scheme will not have a problem.

Moreover, our current implementation does not intercept the HTTPS protocol. Currently, some commercial NIPSes have already done that. In theory, if the enterprise can get the users' private keys, or the enterprise fakes the web server's key in the middle, this can be done. Therefore, we can work in a similar way as these NIPSes for HTTPS.

Our current prototype offers limited protection against malicious plugin content. All the attacks which target data-only plugins such as video codecs, can be detected accurately using WebShield since the data will be examined in the shadow browser. Both policy and behavior based approaches can work in such cases. However, some plugins such as Flash have their own script languages. With scripts, an attacker can potentially hide they malicious intent from shadow browsers by using non-determinism or user input. To counter this type of attacks, we need to intercept the flash content in a way similar to what we do for the webpages. Flash rewriting has been used by FlashProxy [25] in the past, though for a different purpose. it is possible to extend their technique to transfer visual effects of Flash back to the client browser and run ActionScript (Flash scripts) inside shadow browsers. Although in theory there are a lot of different plugins, the popular ones are limited. Covering popular plugins can already protect most web users. It is our future work to design scalable solutions for supporting a large number of plugins.

9 Conclusion

Detecting attacks in dynamic webpages has been a great challenge, especially when the web content is non-deterministic. Existing host-based solutions suffer from deployment problems due to slow user adoption, while the current middlebox approaches can only accommodate certain limited security protection mechanisms. In this paper, aiming to design a general middlebox framework that can enable different security protection mechanisms, we developed four design principles and, based on them, designed WebShield, a general secure proxy with shadow browsers. In WebShield, we ensure no untrusted scripts can be run on client browsers, and thus close the door for attacks that employ non-determinism or user input to bypass detection. This way, we can filter out malicious parts of a web page while rendering the rest of the page at the user-side. Evaluation shows that WebShield can be applied to an enterprise network to prevent attacks, even those with non-deterministic behavior or involving user interaction, and protect the end-user browsers from both known and unknown vulnerabilities.

10 Acknowledgements

We would like to thank Shamiq Islam for his contribution in the early stage of this project. This work was supported by US NSF CNS-0831508, China NSFC (60625201, 60873250, 61073171), China 973 project (2007CB310701), and Tsinghua University Initiative Scientific Research Program. Opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of the funding sources.

References

[1] Alexa Top Sites. http://www.alexa.com/topsites/global.

[2] Blue Coat Secure Web Gateway. http://www.bluecoat.com.

[3] Greasemokey Add-on for Firefox. https://addons.mozilla.org/en-US/firefox/addon/748/.

[4] JavaScript Games. http://javascript.internet.com/games/.

[5] JSON. http://json.org/.

[6] McAFee SiteAdvisor. http://www.siteadvisor.com/.

[7] Opera Mini. http://en.wikipedia.org/wiki/Opera_Mini.

[8] Research: 1.3 million malicious ads viewed daily. http://www.zdnet.com/blog/security/research-13-million-malicious-ads-viewed-daily/6466.

[9] TC. http://pupa.da.ru/tc/.

[10] The Webkit Open Source Project. http://webkit.org/.

[11] W3C Document Object Model. http://www.w3.org/DOM/.

[12] A. Barth, U. Berkeley, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *Proc. of USENIX Security Symposium*, August 2009.

[13] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proc of IEEE Symposium on Security and Privacy*, 2009.

[14] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proc of USENIX Security Symposium*, 2009.

[15] M. Claypool and K. Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006.

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

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

[18] R. Hansen. Xss (cross site scripting) cheat sheet esp: for filter evasion, 2008. `http://ha.ckers.org/xss.html`.

[19] J. Howell, C. Jackson, H. J. Wang, and X. Fan. Mashupos: operating system abstractions for client mashups. In *HotOS*, 2007.

[20] B. L. K.Vikram, Abhishek Prateek. Ripley: Automatically securing web 2.0 applications through replicated execution. In *CCS*, 2009.

[21] D. Lowet and D. Goergen. Co-browsing dynamic web pages. In *Proc of WWW*, 2009.

[22] D. Malkhi and M. K. Reiter. Secure execution of java applets using a remote playground. *IEEE Trans. on Software Engineering*, 26(12), 2000.

[23] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. Spyproxy: execution-based detection of malicious web content. In *USENIX Security*, 2007.

[24] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proc. of NDSS*, 2006.

[25] A. Moshchuk, S. D. Gribble, and H. M. Levy. Flashproxy: transparently enabling rich web content via remote execution. In *Proc. of ACM MobiSys*, 2008.

[26] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proc. of NDSS*, 2009.

[27] NSA. SELinux. `http://www.nsa.gov/research/selinux/`.

[28] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *USENIX Security*, 2008.

[29] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: analysis of web-based malware. In *USENIX HotBots*, 2007.

[30] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security*, 2009.

[31] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proc. of OSDI*, 2006.

[32] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proc. of Eurosys*, 2009.

[33] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *Proc. of IEEE Symposium on Security and Privacy*, 2009.

[34] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of NDSS*, 2007.

[35] 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 *Usenix Security*, 2009.

[36] Y.-M. Wang, D. Beck, X. Jiang, and R. Roussev. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proc. of NDSS*, 2006.