Virtual Browser: a Virtualized Browser to Sandbox Third-party JavaScripts with Enhanced Security

Yinzhi Cao
Northwestern University
Evanston, IL
yinzhi.cao@eecs.northwestern.edu

Zhichun Li
NEC Research Labs
Princeton, NJ
zhichun@nec-labs.com

Vaibhav Rastogi
Northwestern University
Evanston, IL
vrastogi@u.northwestern.edu

Yan Chen
Northwestern University
Evanston, IL
ychen@northwestern.edu

Xitao Wen
Northwestern University
Evanston, IL
xitaowen2015@u.northwestern.edu

ABSTRACT

Third party JavaScripts not only offer much richer features to the web and its applications but also introduce new threats. These scripts cannot be completely trusted and executed with the privileges given to host web sites. Due to incomplete virtualization and lack of tracking all the data flows, all existing approaches without native sandbox support can secure only a subset of third party JavaScripts, and they are vulnerable to attacks encoded in non-standard HTML/-JavaScript (browser quirks) as these approaches will parse third party JavaScripts independently at server side without considering client-side non-standard parsing quirks. At the same time, native sandboxes are vulnerable to attacks based on unknown native JavaScript engine bugs.

In this paper, we propose Virtual Browser, a full browser-level virtualized environment within existing browsers for executing untrusted third party code. Our approach supports more complete JavaScript language features including those hard-to-secure functions, such as *with* and *eval*. Since Virtual Browser does not rely on native browser parsing behavior, there is no possibility of attacks being executed through browser quirks. Moreover, given the third-party Javascripts are running in Virtual Browser instead of native browsers, it is harder for the attackers to exploit unknown vulnerabilities in the native JavaScript engine. In our design, we first completely isolate Virtual Browser from the native browser components and then introduce communication by adding data flows carefully examined for security. The evaluation of the Virtual Browser prototype shows that our execution speed is the same as Microsoft Web Sandbox[27], a state of the art runtime web-level sandbox. In addition, Virtual Browser is more secure and supports more complete JavaScript for third party JavaScript development.

ASIACCS '12, May 2–4, 2012, Seoul, Korea.
Copyright 2012 ACM 978-1-4503-0564-8/11/03 ...$10.00.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Invasive software (e.g., viruses, worms, Trojan horses); Information flow controls

General Terms

Security, Design, Language

Keywords

Third-party JavaScript, Web Security, Virtualization

1. INTRODUCTION

Modern web sites often use third party JavaScripts to enrich user experiences. Web mashups combine services from different parties to provide complex web applications. For example, a web site may use JavaScript games from other party to attract users, include JavaScript code from targeted advisement companies for increasing revenue, embed a third party counter to record the number of visited users, and enable third party widgets for richer functionalities. In each of these cases, some third party JavaScripts, which are not developed by the web site, have the same privileges as the JavaScript code from the web site itself. Although these third party JavaScripts enrich the functionalities of the web site, malicious third party Javascripts can potentially fully subvert the security policy of the web site, and launch all kinds of attacks.

Therefore, in this paper, we propose Virtual Browser, a virtualized browser built on top of a native browser to sandbox third-party JavaScript. The idea of Virtual Browser is comparable to that of virtual machines. It is written in a language that a native browser supports, such as JavaScript, so that no browser modification is required. Virtual Browser has its own HTML parser, CSS parser, and JavaScript interpreter, which are independent from the native browser. Third party JavaScripts are parsed only once in Virtual Browser and run on top of the virtual JavaScript interpreter. The untrusted third party JavaScripts are isolated from the trusted JavaScripts of the web site by design. Virtual Browser introduces only the necessary communications between the JavaScripts from the web site and the third party JavaScripts.

Existing works such as Microsoft Web Sandbox [27] and Google Caja [20] may also be thought of as employing vir-

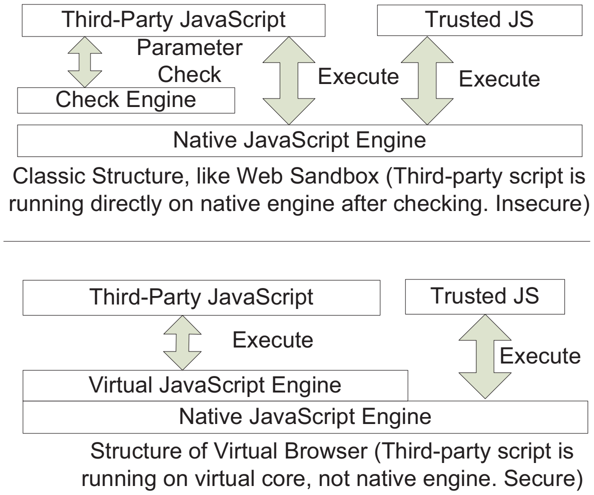
Third-Party JavaScript
Trusted JS
Parameter Check
Check Engine
Execute
Execute
Native JavaScript Engine
Classic Structure, like Web Sandbox (Third-party script is running directly on native engine after checking. Insecure)
Third-Party JavaScript
Trusted JS
Execute
Execute
Virtual JavaScript Engine
Native JavaScript Engine
Structure of Virtual Browser (Third-party script is running on virtual core, not native engine. Secure)

Figure 1: Classical Structure vs. Virtual Browser

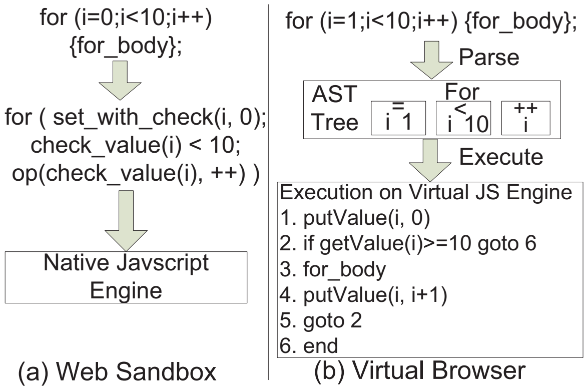
for (i=0;i<10;i++) {for_body};
for (set_with_check(i, 0); check_value(i) < 10; op(check_value(i), ++))
Native Javscript Engine
(a) Web Sandbox
for (i=1;i<10;i++) {for_body};
Parse
AST Tree
=
i 1
For
<
i 10
++
i
Execute
Execution on Virtual JS Engine
1. putValue(i, 0)
2. if getValue(i)>=10 goto 6
3. for_body
4. putValue(i, i+1)
5. goto 2
6. end
(b) Virtual Browser

Figure 2: Comparison of Web Sandbox and Virtual Browser when Executing For-loop

tualization but our technique is significantly different. The key difference is whether third party JavaScripts are directly running on a native JavaScript engine. Figure 1 illustrates the difference. Virtual Browser executes third party JavaScripts on a virtualized JavaScript engine; on the other hand, existing approaches check the parameters of each third party JavaScript expression and then let them execute directly on the native JavaScript engine. Web Sandbox[27] makes a big step toward virtualization. It provides a virtualized environment for native execution of third party JavaScripts, but its execution is still restricted by the parameter checking model. As shown in Figure 2(a), it provides a virtualized environment for *for* loop. All the variables are fetched from the virtualized environment. However, the *for* loop itself is still running on a native JavaScript engine[1]. As shown in Section 2, this is the reason why they are vulnerable to unknown native JavaScript engine vulnerabilities and it is hard for them to handle dynamic JavaScript features like *eval* and *with*.

Security is the key property of our design. In order to make Virtual Browser secure, we need to prevent third party JavaScripts from directly running on a native JavaScript engine. Two methods are used here to achieve our design: *avoidance* and *redirection*. Avoidance means that we avoid using some dangerous functions in the native browser when implementing Virtual Browser. The dangerous functions are functions that potentially lead a JavaScript string to be parsed and executed on the native JavaScript engine. For example, native *eval* in JavaScript can execute a string. If *eval* is not used appropriately, third party scripts, which are input to Virtual Browser as a string, can flow to the native *eval* and get executed. Hence, we do not use the native *eval* function when implementing the virtual browser. This ensures that there is no way for third party JavaScripts to exploit the Virtual Browser to access the native *eval* function. Redirection means we redirect data flows to a place that is ensured to be secure. For example, third party JavaScripts may leak to the native JavaScript engine through calls to the function *setTimeout*. Instead of letting the flow go into the native JavaScript engine, we redirect it back to the virtual JavaScript engine. We will present the details about how we use these two methods in Section 4.

The execution speed of our system is similar to that of Web Sandbox [27] and is fast enough to sandbox reasonable lengths of third party JavaScripts as discussed in Section 6.1. We find that even animation scripts, which trigger events much more frequently than average scripts, can work well with Virtual Browser.

Contributions: We make the following contributions.

- **Virtualization.** We propose the concept of Virtual Browser and adopt virtualization to provide security. Although earlier works may be conceived of as implementing some sort of virtualization, it is incomplete; we are the first to provide browser-level virtualization.
- **Enhanced Security for Isolation[2].** As shown in Section 2, compared to native sandboxes [26, 35] that purely rely on *iframe* for isolation, Virtual Browser is more robust to unknown native JavaScript engine vulnerabilities.
- **Securing even traditionally Unsafe Features in JavaScript.** A recent measurement study [40] reveals that 44.4% of their measured web sites use *eval*. In addition, Guarnieri et al. [21] shows that 9.4% of widgets use *with*. Thus, it is important to secure these functionalities.

2. MOTIVATION AND RELATED WORK

Related work in securing third-party JavaScript can be classified into two categories: approaches using JavaScript features and approaches using native browser features.

Securing Third-party JavaScript by Using JavaScript Features. Existing works about securing third-party JavaScript by using JavaScript features can be classified into three sub-categories: static, runtime, and mixed approaches.

- *Static Methods.* Many approaches, such as ADSafe [1], FBJS [4], CoreScript [39], and Maffeis et al. [24], restrict JavaScript to a subset and perform static check and enforcement upon third-party JavaScript.

[1] Microsoft Web Sandbox actually transforms the for loop into $for(e(b, "i", 0); c(b, "i") < 10; i(b, "i", 1))$ for conciseness. Since the sentence is difficult to be read, we change its presentation to be a human-readable manner as shown in Figure 2.

[2] For securing communication between trusted and third party JavaScripts, Virtual Browser DOES NOT improve state-of-the-art.

	Approaches with native browser support			JavaScript level approaches			
	Modifying browsers [26, 12]	Approaches [18] Using NaCl	Using *iframe* [35, 16]	Static methods [1, 39, 24, 30]	Mixed methods [21, 22]	Runtime approaches	
						Others [20, 27]	Virtual Browser
Robust to browser quirks	Yes	Yes	Yes	No	No	No	Yes
Robust to drive-by-downloads	No	Yes	No	No	Partial	Partial	Yes
(1) Caused by unknown JavaScript engine vulnerabilities	No	Yes	No	No	No	No	Yes
(2) Caused by others	No	Yes	No	No	Yes	Yes	Yes
Dynamic JavaScript feature support (like eval and with)	Yes	Yes	Yes	No	No	No	Yes
Support by all browsers	No	No	Yes	Yes	Yes	Yes	Yes
Speed	Fast	Fast	Fast	Fast	Medium	Slow	Slow

Table 1: Comparing Virtual Browser with Existing Approaches

- *Runtime Methods.* Microsoft Web Sandbox[27] and Google Caja[20] rewrite JavaScript to wrap up every dangerous part and perform runtime check with its own libraries. BrowserShield[31] and Kikuchi[23] are middle box solutions put at a proxy or a gateway. They wrap JavaScript with checking code at the middle box and perform runtime check at client side.
- *Mixed Methods.* Gatekeeper[21], a mostly static method also contains some runtime methods. It performs a points-to analysis on third-party JavaScript code and deploys policies during runtime. Huang et al. also check information flow at client side and protect client at runtime[22].

Securing Third-party JavaScript by Native Browser Features. Approaches using native browser features can be classified into three sub-categories: modifying native browser, using plugin, and using *iframes.*

- *Browser Modification.* ConScript [26]/WebJail [12] modify the IE8/Firefox browser kernel to enforce the deployment of policies (advices). MashupOS [37] proposes a new HTML tag *Sandbox* to secure Mashups by modifying native browsers. OMash [16] modifies Firefox to adopt object abstraction and isolate Mashups.
- *Using Plugins.* AdSentry [18] executes third-party codes in a shadow JavaScript engine sandboxed by Native Client [38].
- *Using iframes.* AdJail [35] and SMash [17] proposes using iframes to isolate third-party JavaScript.

Comparing with Virtual Browser. We discuss four important points: (a) Robustness to browser quirks, (b) Robustness to unknown native JavaScript engine vulnerabilities, (c) Support of some dynamic language features, and (d) Support by all existing browsers. The comparison is shown in Table 1.

First, we show how browser quirks, non-standard HTML and JavaScript, influence present approaches. All existing web browsers support browser quirks, because web-programmers are humans, and mistakes are very likely during programming. Browser quirks have previously been well studied in BluePrint[36] and DSI [29]. For example, the server side filter on the Facebook server had such vulnerability [34]. A string *<img src="..." onload:=attackcode>* is interpreted as *<img src="..." onload = attackcode>* at browsers (Firefox 2.0.0.2 or lower) but as *<img src="...">* at the server-side filter.

All existing JavaScript level approaches [21, 1, 30, 27, 20] define a particular server side (or middle-box) interpretation which may be very different from the browsers' interpretation, and hence remain vulnerable to such attacks. In Virtual Browser, those attacks will not happen because scripts are parsed only once at client-side.

Second, existing native sandboxing approaches that purely rely on *iframe* for isolation, like AdJail [35] and SMash [17], and JavaScript level approaches, like Web Sandbox [27], are vulnerable to unknown native JavaScript engine vulnerabilities but Virtual Browser is more robust to unknown attacks.

We take the for loop as an example in Figure 2 again. Assume there is an unknown integer overflow in the for loop of native browser that is triggered by $for(i = a; i < b; i = i+c)$ when a, b, and c are certain values. For a native sandbox, the vulnerability can be directly triggered. Because the vulnerability is unknown, neither Web Sandbox nor BrowserShield [31] (from which Web Sandbox evolved) can check the parameters and spot the attack. The vulnerability can still be directly triggered because as shown in Figure 2(a), the for loop is running on native JavaScript engine.

In Virtual Browser, since it interprets the for loop, direct input of the for loop will not trigger the vulnerability. As shown in Figure 2(b), the for loop is not running directly on the native JavaScript engine. In order to compromise the native browser with this vulnerability, virtual browser source code needs to have the sentence with exactly the same pattern: a for-loop where a, b and c are all open inputs. Then attackers need to manipulate other inputs to let the sentence in virtual browser source code get the certain values that can trigger the attack. Either of the two conditions is not easy to satisfy, as evaluated in Section 6.3.

Third, some dynamic language features, such as *eval* and *with*, are not supported in present JavaScript level approaches. Developers however still use *eval* to parse JSON strings in old browsers with no native JSON support. Gatekeeper [21] reveals that about 9.4% of widgets use *with*.

As shown in Figure 1, the classical runtime approaches (such as the runtime part in GateKeeper [21]) employ a parameter checking model, implying that they cannot check the safety of *eval* and *setTimeout*, whose parameters contain JavaScript code. and need to be passed to the JavaScript parser. Web Sandbox [27] itself does not execute third party scripts, and therefore it is hard to switch execution contexts for *with* statement. Meanwhile, although it is possible to recursively transfer arguments of *eval* back to the server for further transforming, large client-server delays will occur and render the approach extremely slow. Therefore, in the implementation of Web Sandbox, *with* is not supported and the support of *eval* is incomplete.

Fourth, those approaches, which modify existing browsers or utilize plugins like Native Client [38], are not supported by all existing browsers. Mozilla publicly rejects adopting NaCl [8], and meanwhile there is not clue that IE and Opera will adopt NaCl either. Therefore, those approaches can protect only a limited number of users who deploy their approaches. Virtual browser uses only JavaScript features that are supported by all present browsers.

3. DESIGN

In Section 3.1, we first introduce the architecture of Virtual Browser. Then we give several JavaScript examples to show how exactly Virtual Browser works in Section 3.2.

3.1 Architecture

The architecture of Virtual Browser is shown in Figure 3. Virtual Browser is very similar to a native web browser except that it is written in JavaScript. We will introduce the interface, components and flows of Virtual Browser below.

3.1.1 Interface

Similar to Microsoft Web Sandbox, Virtual Browser takes a string, which contains the code of a third-party JavaScript program, as input. For example, we are using the following codes to include third-party JavaScripts.

```
<script> evaluate(str); </script>
```

str is a string that represents a third-party JavaScript code, which can be embedded inside host web pages. Maintenance of *str* is vulnerable to string injection attacks. In our approach, we leverage Base64 encoding, one of the many existing ways [36, 29] to prevent string injection attacks.

Virtual Browser also provides a file loading interface.

```
<script> run("http://www.a.com/JS/test.js"); </script>
```

An *XMLHTTPRequest* will be made to the same origin web server (where all third-party and trusted codes and Virtual Browser are fetched) first by Virtual Browser. The same origin web server will redirect the request to the real web server (`www.a.com`). Therefore received contents will be fed into the aforementioned *evaluate* interface.

3.1.2 Components and Data Objects

The functionality of these components and data objects in Virtual Browser is similar to their corresponding parts in native browser.

Components of Virtual Browser include a virtual JavaScript parser, a virtual JavaScript execution engine, a virtual HTML parser, a virtual CSS parser and so on.

- *Virtual JavaScript Parser:* It parses third-party JavaScript codes and outputs the parsed JavaScript AST tree.
- *Virtual JavaScript Execution Engine:* It executes the parsed JavaScript AST tree from virtual JavaScript parser. The interface of the JavaScript execution engine has three parts: *putValue*, *getValue* and *function call/return*. *putValue* is loaded every time an object is changed. Every modification to a private (defined by third-party) function/variable or a shared (from trusted scripts or third-party) function/variables goes through *putValue*. *GetValue* provides an interface for every read operation. *Function call/return* are used for calling shared functions from natively running code and private functions from third-party codes. Our design of the interface of the virtual JavaScript engine is similar to the one of the native JavaScript engine. Several works[10, 15] have details about the native JavaScript engine's interface.
- *Virtual CSS Parser:* It parses CSS codes and attaches the results to Virtual DOM.
- *Virtual HTML Parser:* It parses HTML codes provided by other components and output DOM tree.

Data objects of Virtual Browser include virtual DOM and other private objects.

- *Virtual DOM:* It is linked to the native DOM as an `iframe`. The link is purely for virtual DOM to be shown on the screen. JavaScript access to native DOM from third-party codes is forbidden by our virtualization technique as shown in Section 4.1. JavaScript access to virtual DOM from trusted codes is also forbidden by `iframe` isolation[3]. Meanwhile, the native DOM also transfers all the events generated automatically by native browsers back to Virtual Browser.
- *Private Objects:* Private data is used to store JavaScript objects that is only accessible to third party JavaScript in Virtual Browser. Section 4.1.2 gives the isolation details.

3.1.3 Flows

Flows inside Virtual Browser. When a third-party JavaScript code runs into Virtual Browser, the virtual JavaScript parser will first parse it to an AST tree and give the tree to the virtual JavaScript execution engine. The virtual JavaScript execution engine will execute the AST tree similar to a normal JavaScript interpreter does. When HTML content is found, virtual JavaScript execution engine will send it to the virtual HTML parser. Similarly, JavaScript codes and CSS style sheets will be sent to the virtual JavaScript and CSS parsers. Virtual HTML parser will parse HTML and will send scripts/style sheets to the virtual JavaScript/CSS parsers. All of these processes are shown in Figure 3. We will give a detailed analysis on these flows in Section 4.2.2.

Flows between Virtual Browser and Trusted Codes. Virtual Browser is isolated from trusted codes as analyzed in Section 4.1. The only flow left is a shared object[4] that connects trusted codes running upon a native browser and third-party codes running on a Virtual Browser.

3.2 Examples for Several JavaScript Operations

[3] Notice that the isolation provided by `iframe` is purely for preventing access to virtual DOM from trusted code by mistake so that privilege escalation can be minimized (Please refer to Section 4.1.2 for details). Virtual Browser is still more robust to unknown native JavaScript engine vulnerabilities than native sandbox approaches, like AdJail [35].

[4] Object here is an abstracted concept, which can also be a single value. According to some recent work done by Barth et al.[13], in some cases, values might be less error-prone than objects.

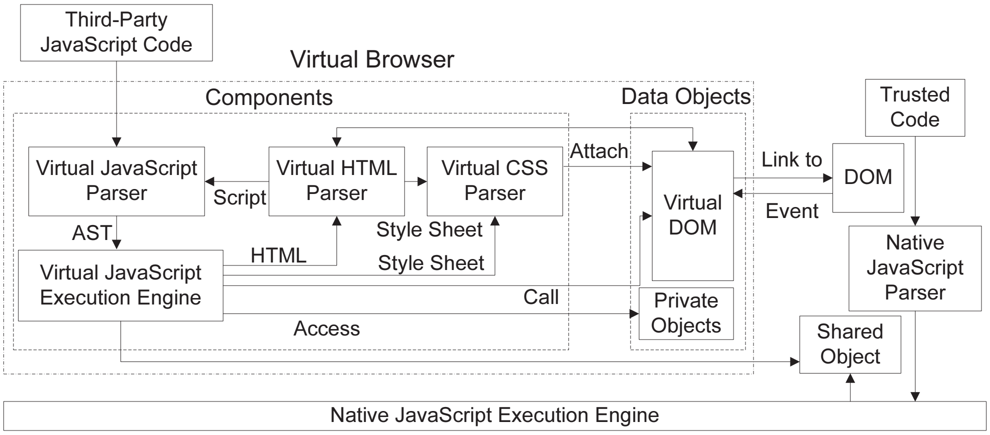

Figure 3: System Architecture

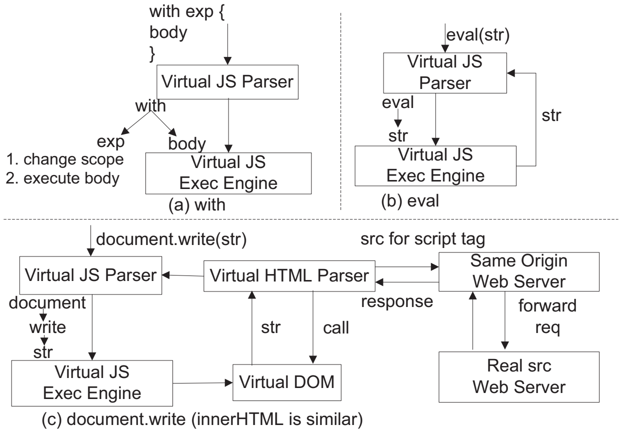

Figure 4: Securing Several Dynamic JavaScript Operations

In this section, we illustrate several JavaScript operations in Virtual Browser to show how Virtual Browser works. Some of them are not supported by previous approaches.

with. *with* is a notoriously hard problem in this area. None of the existing works can solve this problem. In our system, *with* becomes quite simple because Virtual Browser interprets JavaScript. For example, as shown in Figure 4(a), *with exp* in our system is just a switch of current context.

eval. *eval* is often disallowed by existing approaches totally or partially because it will introduce additional unpredictable JavaScript. As shown in Figure 4(b), Virtual Browser just needs to redirect contents inside *eval* back to our virtual JavaScript parser. No matter how many *eval*s are embedded, such as *eval(eval(...(alert('1'))...)*, JavaScript is still executing inside our virtual JavaScript engine.

document.write/innerHTML. *document.write* and *innerHTML* are related to the HTML parser. As shown in Figure 4(c), when virtual JavaScript execution engine encounters functions/variables like these, it will redirect them to the virtual HTML parser by calling methods in virtual DOM. If scripts like < *script src* = ".." > are found in those HTML codes, an XMLHTTPRequest like `http://www.foo.com/get.php?req=...` will be sent to the same origin web server (`www.foo.com`, where all third-party and trusted codes and Virtual Browser are fetched) due to same-origin policy restriction and then redirected to the real web server. JavaScript contents will be received and redirected back to virtual JavaScript parser.

arguments. *arguments* are implemented inside a function. Arguments of the current function are stored in the current running context. When the third party codes use *arguments*, Virtual Browser can fetch them directly.

4. SECURITY ANALYSIS

In this section, we analyze the security of Virtual Browser by making sure third-party JavaScripts and their data flows only inside Virtual Browser, and create necessary communication channels to the data and resources outside Virtual Browser. We adopt two methods to ensure security : avoidance and redirection. In Section 4.1, we sandbox all the components inside Virtual Browser by cutting off the inflows and outflows to and from the sandbox. We avoid using some of JavaScript's dangerous functions in the Virtual Browser implementation to achieve isolation. In Section 4.2, we enable shared objects and communications with security access control. Because we have already built an isolated sandbox, in the second part of the design, we mainly redirect dangerous flows within the third party code back to the sandbox to facilitate communication.

4.1 Isolation through Avoidance

As we mentioned before, we design our sandbox as another browser built on top of existing browsers, which we call the Virtual Browser, similar to the concept of a virtual machine. The right part of Figure 1 shows a Virtual Browser upon a native browser. The Virtual Browser and the trusted JavaScripts from the web site are all running on the native browser. The third-party JavaScript is running on the virtual browser. The method for building the Virtual Browser is similar to building a native browser. We need to build a JavaScript engine, HTML parser, CSS parser and so on. Those components are fully written in JavaScript. We will not focus on how to implement each of the components here, which are not very different from writing a native browser. What we are interested in is how to isolate the Virtual Browser from native browsers. Since we have not introduced communication yet, we only need to enforce isolation between the trusted JavaScripts from the web site and the third-party JavaScripts.

4.1.1 Cutting off Outflows of Virtual Browser

Cutting off outflows of Virtual Browser means that we want to prevent the third-party codes that run on Virtual Browser from running directly on the native browser. We ensure the third-party codes are trapped inside Virtual Browser. To achieve that, we have the following assumption.

Any JavaScript code has to be parsed in the native JavaScript parsers before it can be executed in native browser.

The Virtual Browser treats a third-party JavaScript as a string, and the string is the input for a virtualized browser instance. Virtual Browser calls the virtual JavaScript parser, which is part of the virtual JavaScript engine, to parse the string, and then executes the code on the virtual JavaScript interpreter. We need to prevent any part of the string from feeding into the native JavaScript, CSS, and HTML parsers. On native browsers, the operations that can cause JavaScript/CSS/HTML parsing are limited. Therefore, we have to avoid using all kinds of operations, such as *eval*, *document.write* and so on when implementing our system, so that the native parsers have no chance to be triggered on the string that contains the third-party JavaScript code.

The follow-up question is how we can find these operations that cause native parsing. We use two approaches: looking up references and looking at the source code of native browsers. Most browsers have their own references and many browsers obey certain standards such as DOM, CSS, and JavaScript. We look at those manuals to figure out which functions trigger native parsing.

However, those manuals may be erroneous and may be lacking details. Call graph analysis on the source code of native browsers is another option. We examine parts of the call graph in which only functions which call the *parse* function will be listed (the functions, which indirectly call the *parse* function through other functions, are also included). We avoid using all of the functions that are direct or indirect callers to native parsing when implementing Virtual Browser.

Cutting off Outflows to the Native JavaScript Parser.

We need to make sure the third-party codes running in the Virtual Browser cannot leak to the native JavaScript parser. First, we looked up the JavaScript reference[6] and verified that only *eval* and function construction from string, such as *new Function(string)* can result in calling the JavaScript parser. We also did a static call graph analysis of the WebKit[5] JavaScript Core using Doxygen[3].

In WebKit, we found that the following functions could call the JavaScript parser directly or indirectly. We trace functions only in the JavaScript Core.

- *opcode op_call_eval.* When *eval* is evaluated, *op_call_eval* is called, and the JavaScript parser is invoked to parse the parameter of *eval*.
- *evaluate.* This function is used by JSC (a test module in the JavaScript Core, not used in real browsers) and the DOM to parse <script> tag, which is outside the JavaScript Core.
- *constructFunction.* It is the function constructor. When feeding a string into the function constructor, JavaScript parsing is triggered. It is used by JSEventlistener which binds a JavaScript function as an event handler for a specific event, and also is used by JSObjectMakeFunction, an open API provided by the JavaScript Core.
- *functions in JavaScript Debugger.* Virtual Browser is not using debugging mode.
- *functions in JavaScript Exception Handling.* A JavaScript string may be reparsed during exception handling in order to get information such as line number, exception id, etc. In the reparsing function, it will reparse exactly the same string as before.
- *numericCompareFunction.* It uses the JavaScript parser to parse a numeric constant string. Numeric constants can be considered as safe, so this is not an issue.

Because the code of Virtual Browser uses only the basic functions in the native JavaScript engine, to avoid using *eval* and function construction through string is enough to prevent third-party JavaScripts in Virtual Browser from leaking out to the native JavaScript parser.

Cutting off Outflows to the Native HTML Parser and the CSS Parser.

Similar to cutting off flows to the native JavaScript parser discussed above, we need to make sure that third-party HTML or CSS do not leak to the native parsers. We found that the native core JavaScript engine does not call the native HTML or CSS parsers at all, so we can be sure that the third-party HTML or CSS is not leaked to the native parsers from Virtual Browser.[6]

In conclusion, outflows of Virtual Browser are cut off.

4.1.2 Cutting off Inflows of Virtual Browser

Cutting off inflows of Virtual Browser means preventing the trusted JavaScripts of the web site from accessing the objects in Virtual Browser directly. Although the JavaScripts from the web site may not have intentionally malicious behavior, it may influence the virtualized browser by mistake. For virtual DOM, we link data in virtual DOM to an *iframe* tag to prevent trusted codes from the web site to access it by mistake. For other objects, we perform an encapsulation of the *virtualized browser* based on the encapsulation of *Object* in object-oriented languages. We provide only a limited API as the interface and put all other variables and objects as private objects inside Virtual Browser. We also use anonymous objects to prevent inheritance and misuse of Virtual Browser. An example follows.

```
(function(){ this.evaluate= function (){codes} other codes})();
```

From the perspective of the native JavaScripts, they can only see *evaluate* but no other private objects inside the virtual JavaScript engine. The web developer (writing trusted scripts owned by the web site) would be required to avoid overwriting this narrow interface of Virtual Browser.

[5] Some undocumented and non-standard functions that can cause parsing may be used in close-source browsers, like IE, and new features may also be introduced in future version of those browsers. However, because we do not even know those new, undocumented, or non-standard functions, they are definitely not used in the source codes of Virtual Browser.

[6] Note that functions such as *document.write* belong to DOM and not the JavaScript engine.

4.2 Communication through Redirection

In Section 4.2.1, we will discuss data security that makes sure general data is secured, and script security that is more complicated to handle. Then, in Section 4.2.2, we give a detailed analysis on each component of Virtual Browser to show how data and scripts are flowing.

4.2.1 Data Security

Data security consists of general data security and security of special data—script.

General Data Security.

Data in our system is classified into two categories: private data and shared data. Private data refers to virtualized objects and functions in Virtual Browser, which are generated by third-party codes. Shared data refers to some shared objects for communication between the third-party and the trusted codes. Private data in Virtual Browser is safe because of encapsulation of our sandbox in Section 4.1.2. We however need to consider the security of shared objects. Data in shared objects can be secured in various ways. We illustrate two methods: mirrored objects and access control.

- *Mirrored Object.* Some objects, such as *String*, *Date*, and *Math* are not necessarily shared. We can create a copy of such an object, which means we turn them into private data. Below is a simplified example copied from Mozilla Narcissus JavaScript engine[28].

```
function String(s) {
        s = arguments.length ? "" + s : "";
        if (this instanceof String) {
            this.value = s;
            return this;
        }
        return s;
    }
```

  When third-party scripts try to access a property of *String*, third-party scripts are accessing the mirrored *String* defined in Virtual Browser. Similarly, *Object* may be mirrored to prevent using prototype chain to access the original (unique) *Object* object. Notice that not only *Object* is virtualized here but also the whole process of accessing. Because each time third-party codes call the property *prototype*, they will go through *getValue* interface of Virtual JavaScript engine. Virtual Browser will fetch virtualized contents for it. So the whole accessing process is secured.
- *Access Control.* Virtual Browser uses access control to secure must-share data, which means third-party scripts can access an object only when they have the right to do that. There are many existing works about how to facilitate communications between different entities, such as methods in postMessage channel [14], and Object Views [25]. They employ different policy engines. We can adopt either of them. Object Views [25], which can also be deployed on Google Caja [20], is a good choice for us.

Script Security.

Scripts are also a special kind of data, but they can be executed. Execution of third-party scripts outside of the virtual JavaScript engine may cause the third-party JavaScript to escalate its privileges. Based on the assumption about native execution we made earlier, i.e., scripts have to be parsed before execution, we need to prevent third-party data from flowing into the native parser. Therefore, our task is to track flows that might go out of the virtual JavaScript engine. Scripts in Virtual Browser are classified into two categories: confirmed scripts and potential scripts.

First, some types of data that we know are JavaScript codes, such as the data assigned to *innerHTML*, the parameter of *setTimeout*, etc. For this type of scripts, we use redirection to redirect these kinds of data back into Virtual Browser. There are two categories.

- *Scripts that need immediate execution.* For example, when some JavaScript strings passed to *eval*, they will be added to the JavaScript execution queue immediately. When processing *eval*, we directly put these codes back into our system. For example, *eval(str)* will be interpreted as *evaluate(str)* in which *evaluate* is part of our system.
- *Scripts that need delayed execution.* Some scripts' execution is triggered by certain events or functions. For example, when *setTimeout(”alert(1)”,100)* is executed, the code inside will be executed 100 milliseconds later. We adopt a pseudo-function pointer here. Virtual Browser first parses scripts and put it into an anonymous function. In this function, Virtual Browser executes parsed scripts with correct scope and registers this anonymous function with events that will trigger the original function. Therefore, when an event triggers this function or a function calls this function, this function will execute the parsed data inside our system. We still ensure third-party scripts are running in the sandbox. For the aforementioned example, it will look like *setTimeout(function(){execute(parsed_node, exe_context)},100).* We call this method the *pseudo-function pointer* method.

Second, we do not know if some data is a script or not. This data is therefore a potential script. Potential scripts exist because the trusted JavaScripts running on the native JavaScript engine, plugins, etc. have higher privileges and we do not have control over them. They may get data and execute it as a script, which is the well-known privilege escalation problem. Trusted JavaScripts from the web site will not intentionally behave maliciously, but they may unintentionally grant escalated privileges to the third-party JavaScripts. Next, we will analyze the potential privilege escalation possibilities in our system, which can be classified into three categories.

- *Access Shared Object.* Trusted JavaScript code may access shared objects, which are contaminated by third-party JavaScript code. Therefore, trusted JavaScript code might unintentionally grant escalated privileges to a third-party JavaScript.
- *Access Sandbox.* Trusted JavaScript codes may access the sandbox directly. We have already discussed this issue in Section 4.1.2.
- *Access Third-Party JavaScript.* Trusted JavaScript code may want to invoke some third-party JavaScript code. Because of our encapsulation of Virtual Browser, direct access is prohibited. However, if a function call is necessary, Virtual Browser can use pseudo-function pointers discussed earlier to let trusted JavaScript invoke third-party code while ensuring security at the same time.

Because the last two types of privilege escalation are solved by encapsulation, only the first one remains. This type of privilege escalation is not the focus of our paper. We can leverage the solutions from previous works. Finifter et al.

[19] propose a solution by limiting interface of shared objects and third-party programs in ADSafe. The interface of Virtual Browser is similarly designed to be narrow in order to control the flow.

4.2.2 Security Analysis of Data Flows in Virtual Browser Components

Now, we discuss the flow of data in Virtual Browser. We analyze data flows and script flows among three components, which are the virtual JavaScript engine, the virtual HTML and CSS parsers.

Securing Data Flows of the Virtual JavaScript Engine.

A narrow interface will help limit data flows and ease data flow examination. As mentioned in Section 3.1.2, the interface of the virtual JavaScript Engine has three parts: *putValue*, *getValue* and *function call/return*. Every flow from the virtual JavaScript engine needs to go through this interface.

We show all possible data and script flows of the virtual JavaScript engine in Figure 5. Every flow that goes into the native JavaScript engine is conservatively considered malicious because we have no control over the native JavaScript engine. All the flows in Figure 5 are presented below.

- *Redirection of Possible Accesses (Flows 1 and 2).* Flow 1 exists because JavaScript can generate JavaScript. For example, *eval* can execute a string as JavaScript. Flow 2 is caused by the fact that some objects, functions and properties may lead to HTML, JavaScript, and CSS parsing. For example, the *document.write* function and *innerHTML* property cause HTML parsing. The *onClick* property causes JavaScript parsing. We use redirection to redirect these functions to corresponding components in our system. For example, a modification of *innerHTML* in virtual DOM node will use the virtual HTML parser in Virtual Browser to parse it.
- *Privilege Escalation (Flows 4, 5, and 6).* Because trusted JavaScript has higher privileges, privilege escalation can happen. It can do anything as we mentioned in Section 4.2.1. Flow 4 is to access shared objects. Flow 5 is to access third-party JavaScript codes. Flow 6 is to access Virtual Browser itself. Flow 5 is secured by *pseudo-function pointers*. Flow 6 is secured by encapsulation. Flow 4 is hard to secure. We illustrate some methods in Section 4.2.1. In this regard, we are on par with state-of-the-art approaches, such as ADSafe [1], Web Sandbox [27] and Google Caja [20].
- *Hidden Access to Native Code (Flow 3).* Flow 3 is a hidden flow. In common cases, JavaScript running on the virtual JavaScript engine cannot access native JavaScripts and through them access the native JavaScript Engine. However, the hidden flow can be triggered by two conditions. First, the third-party JavaScripts running on the virtual engine modify shared objects that belong to flow 7 (part of flow 2). Second, native JavaScripts use a shared object that belongs to flow 4. We can break either of these two conditions to prevent this kind of privilege escalation. We have discussed how to break the second condition in Section 4.2.1. We will discuss about cutting the first here. If a mirrored object is used, flow 7 is blocked automatically. If access control mechanism is used, a *write* privilege will have to be carefully given to a third-party program. Usually, a *read* privilege is sufficient.

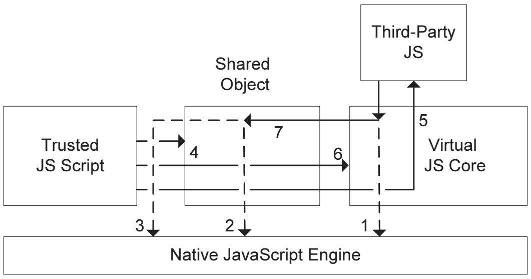

Figure 5: Securing Data/Scripts Flows (I)

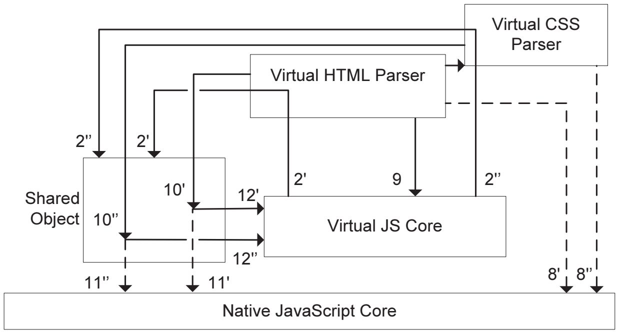

Figure 6: Securing Data/Scripts Flows (II)

Securing Data Flows of the Virtual HTML and CSS Parsers.

The interfaces to the virtual HTML and CSS parsers are narrow. The HTML parser and the CSS parser take a string as input and give parsed results as an output. The virtual HTML and CSS parsers can use each other as well as the virtual JavaScript parser. They do not have other interfaces.

In Figure 6, we show all the possible flows of the virtual HTML and CSS parsers. We ignore flows that come into the virtual JavaScript engine because they have already been discussed.

Flows $2'$ and $2''$ are redirected from flow 2 in Figure 5. Any possible parsing of HTML and CSS is done inside the sandbox. Flow $8'$ and $8''$ are outflows of HTML and CSS parsers, discussed in Section 4.1.1. Flow $11'$ and $11''$ are similar to Flow 2 in Figure 5. We use *pseudo function pointers* to redirect them to the virtual JavaScript engine (Flows $12'$ and $12''$). Other flows are inside our system and do not cause any security issues. They facilitate communication among components in Virtual Browser.

5. IMPLEMENTATION

We have implemented a prototype of Virtual Browser. Our virtualized browser contains a virtual JavaScript parser and a virtual JavaScript execution engine, a virtual HTML parser and a virtual CSS parser. We reused and modified the existing JavaScript implementations of the HTML parser[32] and the CSS parser[33]. For our JavaScript engine, we modified Mozilla Narcissus metacircular JavaScript Engine[28], implemented over JavaScript itself. Moreover, we implemented a simplified version of virtual DOM. Only basic functionality is supported. For example, we support *innerHTML*, *outerHTML*, *innerText*, *document.write*, *document.writeln* and so on in Flow $2'$ (Figure 6). As we have already discussed, if some flows are not introduced, it will result only in reduced functionality but no security problems. A production level implementation would also include other components such as an XML parser.

Except the ECMAScript standard, different browsers may implement different non-standard features of the JavaScript language. Our Virtual Browser needs to be compatible with those non-standard features in order to third-party JavaScripts that rely on those features. Two methods are used here. First, we support the standard features and non-standard features which has been implemented by most browsers. For example, we support *try...catch (e) if exp1 ... if exp2 ...* instead of non-standard *try ... catch (e if exp1) ... catch (e if exp2) ....* Second, for important features, we detect browser vendors and versions when necessary. For example, we use *ActiveXObject("Microsoft.XMLHTTP")* in IE but *XMLHttpRequest* in other browsers.

At the same time, sometimes, Virtual Browser needs to provide non-supported features than the underlying native browser. In this case, we need to mimic those non-supported features. For example, IE does not support the keyword *const*; so we use *var* and give it a tag if it is a constant to make it appear like a *const*.

6. EVALUATION

This section is organized as follows. In Section 6.1, we evaluate the performance of Virtual Browser prototype, memory usage, and parsing latency. In Section 6.2, 6.3, and 6.4, we evaluate Virtual Browser prototype with existing browser quirks and native JavaScript engine bugs, and completeness of our prototype.

6.1 Performance Evaluation

We measure the execution speed of Virtual Browser with mircrobenchmarks and macrobenchmark, and follow with discussion.

Microbenchmarks.

We compare the execution speed of Virtual Browser and Microsoft Web Sandbox [27], a state-of-the-art runtime approach for sandboxing third-party applications. Microsoft Web Sandbox[27] implemented by Microsoft Live Labs as a web-level runtime sandbox written in JavaScript. The idea was derived from BrowserShield[31] project, which rewrites dynamic web content and inserts runtime security checks. Web Sandbox has some problems with the virtualization of local variables[7]. Therefore, we only use global variables in our benchmarks for the comparison with Web Sandbox. We however note that since Virtual Browser virtualizes all variables, including both global and local, there are no performance impacts if we use local variables instead of global variables in our approach.

Operations	Time
New Game	50ms
Drop a Piece	18ms
Mouse Move	1ms
Game Over	8ms

Figure 7: Delays of Operations in Connect Four

Our experiments are performed on Firefox 3.6. The results of the execution speed of Web Sandbox and Virtual Browser on JavaScript microbenchmarks is shown in Table 2. In this experiment, we measure each important atomic operation 10K times and report the cumulative delay. Our system and Web Sandbox achieve nearly the same performance. Microbenchmarks 1-4 are some basic JavaScript operations. Virtual Browser is faster than Web Sandbox for array operations while for arithmetic and functional operations, the two perform nearly the same. Microbenchmarks 5-7 are object operations and have comparable performance on these two. Microbenchmarks 8-9 are string operations. Virtual browser uses mirrored string object which makes it slightly slower. Microbenchmarks 10-11 are DOM operations. Because Virtual Browser enforces a check on each element passed to or returned by DOM functions, it is slightly slower. Microbenchmark 12 and 13 evaluates the *with* and *eval* statements which are not supported by Web Sandbox.

Macrobenchmark.

We measure Connect Four, a JavaScript game from the front page of a popular JavaScript game web site [5], which is listed as the first site by Google search when searching for JavaScript games. The task of Connect Four is to connect four pieces of the same color. Delays of operations in Connect Four are shown in Figure 7. Virtual Browser does not cause any user visible delay. Due to the fact that *eval* is used, Connect Four cannot be run on Microsoft Web Sandbox without modification.

Discussion.

While Virtual Browser is as slow as Web Sandbox, it is much more secure and complete. Although slowness may limit applicability of this approach, our system remains well suited for securing third party JavaScripts. On average, a single JavaScript operation of a parsed AST tree in our system costs 0.03-0.05ms. For a common JavaScript program without mouse movement events (mouse move events are equivalent to the heavy animation in the later discussion), a user may trigger an event every 500ms, which means one can still write 5000 lines of code (1-2 operations per line). This is long enough to implement a decent program. For a heavy animation script, an image may need to be updated every 50 ms, allowing developers to perform 500 operations to deal with one event, which is long enough for most usages. For example, the *bounce game* in Figure 8 is a medium-sized game (800 lines of codes). From the user's perspective, she

[7] For example, in the following JavaScript code,
var tempReturn; for (var i = 0;i<10000; i++) tempReturn = fncTest();
Web Sandbox will rewrite the code as follows.
var tempReturn; for(var i = 0; i < 1e4; i++) h(); tempReturn = j.fncTest()
Web Sandbox does not wrap up *i* in the above code. To the best of our knowledge, Web Sandbox tries to avoid wrapping local variables to improve performance. Local variables are supposed to be put in their virtualized environment (Virtualizing local variables should be achieved in virtualized environment but not during virtualized execution). Such relaxations are dangerous because it exposes local variables directly to the native JavaScript engine and because the server and client may have different interpretation of local variables due to browser quirks as we have already mentioned in Section 2.

Operation	Web Sandbox	Virtual Browser
1. Arithmetic Operation	492ms	480ms
2. Invoke Function	515ms	525ms
3. Populate Array	1019ms	621ms
4. Read Array	1610ms	1217ms
5. Get member on user object	589ms	416ms
6. Set member on user object	566ms	500ms
7. Invoke member on user object	605ms	523ms
8. String Splitting	466ms	532ms
9. String Replacing	475ms	577ms
10. DOM Operation getElementById	707ms	757ms
11. DOM Operation createElement	673ms	822ms
12. With Statement	N/A	333ms
13. Eval Statement with Simple String	N/A	2867ms

Table 2: 10K Atomic Operations' Speed Comparison of Web Sandbox and Virtual Browser

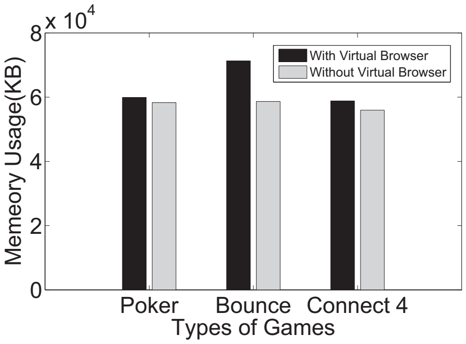

Figure 8: Memory Usage

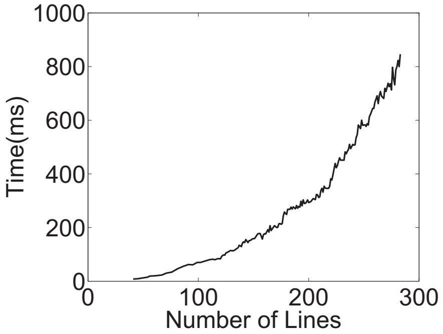

Figure 9: Latency of JavaScript Parsing

can feel only a little additional delay. Our system will not be able to support larger animation JavaScripts. However, for common third-party JavaScripts which do not involve very frequent events and are also not very large, the comparative slowness of our system should not pose a problem at all. We believe the situation is similar to Java and Python being preferred for some applications than C and C++ owing to their ease of programming despite their not being efficient enough. Virtual Browser provides an easy way to handle all kinds of third-party scripts, even those using complex and unsafe functions like *eval* and yet without any security glitches.

6.1.1 Memory Usage

Figure 8 compares memory usage of third-party JavaScript applications running in Virtual Browser to that when they are running in a native browser. We choose third-party JavaScript games from a popular web site[5]. Connect 4 and poker are two games with user interaction. Users need to move and click the mouse to play these games. These two games' memory usage with Virtual Browser is about 2% higher than that without Virtual Browser. Bounce is a game with substantial animation and user interaction in which the user can see a ball bouncing in the screen. Bounce takes about 20% more memory when running in Virtual Browser. This is because high animation programs require more resource from Virtual Browser.

6.1.2 Parsing Latency

We measure parsing latency of the virtual HTML and JavaScript parsers in Figure 9 and Figure 10 by parsing a game web site [5] and a JavaScript game, *Connect 4* from it. Both of these parsers are written in JavaScript. In Figure 9, JavaScript parsing rates decrease slightly when the number of lines increases (as shown in [9], top-down parsing is a polynomial time process). As seen in Figure 10, HTML parsing is faster than JavaScript parsing because JavaScript language with more types of AST nodes is more complex than HTML.

The JavaScript parser written in JavaScript is not very fast. An alternative of this virtual JavaScript parser would be to pre-parse first-order JavaScript and HTML code at server side, generate a JavaScript Object (parsing results) that our execution engine requires, and transmit JSON format to the client. JSON parsing speed is fast enough at about 600K/s. However, this may be vulnerable because the JSON generator at the server side and the JSON parser at the client side may interpret JSON differently. Due to the simplicity and well-formatted-ness of the JSON protocol, the chance of different interpretations is low; so we can still consider this approach as an alternative. We still execute third-party JavaScript on the virtual JavaScript engine (and not on the native JavaScript engine) and even if we adopt this alternative, our approach will still be more secure than other runtime approaches.

Unlike first-order JavaScript, high-order JavaScript and HTML (generated by scripts dynamically), like the scripts introduced by *innerHTML* and *document.write*, have to be parsed in the parser written in JavaScript. Compared to first-order JavaScript, the amount of these kinds of codes is relatively small. Thus, using our parser will not introduce too much delay and the JSON approach remains viable.

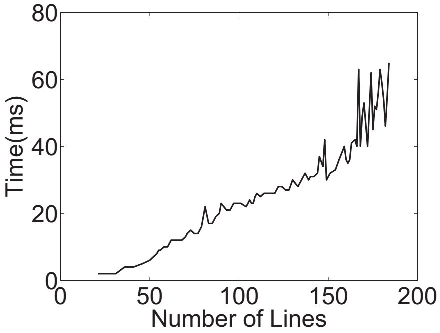

Figure 10: Latency of HTML Parsing

6.2 Browser Quirks Compatibility

Since we parse scripts only once at the virtual JavaScript engine, and do not rely on the native parsers, we are not vulnerable to browser quirks. Additionally, scripts cannot leak from our system to the native JavaScript engine. We evaluate our system with 113 browser quirks' examples listed in the XSS Cheat Sheet [11]. The XSS Cheat Sheet contains mostly examples of XSS attacks that are caused by browser quirks. The results show that none of the 113 browser quirks lets third-party JavaScript codes bypass the virtual JavaScript engine, regardless whether the language features are supported by Virtual Browser.

6.3 Robustness to Unknown Native JavaScript Engine Bugs

We evaluate the robustness of Virtual Browser to unknown native JavaScript engine bugs. In this experiment, we use an old version (before 2009) of Firefox. All the bugs (14 in totals) recorded in CVE [2] related to SpiderMonkey JavaScript engine in the year of 2010 and 2009 are evaluated. The results are the same as we discussed in Section 2. Running of example exploits in the database does not trigger those vulnerabilities in Virtual Browser.

Furthermore, our implementation of Virtual Browser does not satisfy the preconditions for triggering any of those vulnerabilities. We illustrate three as examples. Others are similar.

- **CVE-2010-0165:** Native *eval* is required to trigger this vulnerability. Virtual browser source code does not use native *eval* statement.
- **CVE-2009-2466:** The vulnerability is triggered by the statement $with(document.all)\{\}$. Virtual browser source code does not use *with* statement.
- **CVE-2009-1833:** The prototype of an object is set to be null inside the prototype function. Virtual browser source code does not use prototype in this way.

6.3.1 Discussion

How does Virtual Browser deal with bugs in Virtual JavaScript engine? The experiment of this section is performed to prove the hardness of circumventing virtual JavaScript engine to exploit native JavaScript engine. The security of virtual engine is fully analyzed in Section 4. Moreover, virtual JavaScript engine of Virtual browser is written in JavaScript, a type safe language that does not have vulnerabilities like buffer overflow in native JavaScript engine.

Where does the robustness come from? Virtual Browser is implemented by a type safe language, which gradually reduces the number of possible vulnerabilities. However, the enhanced security does not only come from type safe language but also the virtualization technique.

Browser virtualization adds another layer that increases security assurance. In particular, Virtual Browser utilizes virtualization to isolate JavaScript codes. Attackers need to break Virtual Browser first and then native browser to steal information. Similar to a virtual machine that has higher security assurance than other native sandboxes, Virtual Browser with another virtualization layer has its advantages.

6.4 Completeness of Virtual Browser Implementation

We evaluate the completeness of our prototype of Virtual Browser in order to show that our other experimental results are convincing. We use test cases of ECMA-262 Edition 1 from Mozilla [7]. The results show that we can pass 96% of the test cases. For some categories, such as *JavaScript Statement*, *String*, *Expressions*, *Types*, etc., we can pass 100% of the test cases. The worst of all is Object Objects, for which we can only pass 72% test cases. The incomplete implementation will only affect the functionality of Virtual Browser but not the security of Virtual Browser because we introduce the data flows after isolation. Security is always the most important concern in Virtual Browser.

When we are trying to run the same test in Web Sandbox [27], Web Sandbox cannot even run the test itself because Web Sandbox does not have full support of *eval*. However, *eval* is required in the driver of these test cases. This also proves the importance of *eval* without which we cannot perform the tests. Therefore, we have to give up running the test for Web Sandbox. Our manual check shows that they do not have complete implementation either.

7. CONCLUSIONS

In this paper, we propose the concept of browser virtualization, and we designed and implemented a prototype of Virtual Browser in JavaScript. With Virtual Browser, we allow *unmodified full-featured* third-party JavaScripts to run on Virtual Browser and enforce secure constraints on its privileges and communication with the JavaScripts from the web site. In our design, we first build an isolated virtual browser layer without any possible communication channel with the native browser resources and with the JavaScript from the web site. Then, we introduce the necessary data flows explicitly. Compared with existing efforts with similar goal, such as Web Sandbox, our scheme can allow full-featured JavaScript execution instead of limiting a subset of JavaScript functionalities. Fundamentally, it is much easier for any unmodified third-party JavaScript code to run on our scheme directly. Performance evaluation shows our prototype has a similar performance as Web Sandbox.

8. ACKNOWLEDGEMENTS

This work was supported by US NSF CNS-0831508. Opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of the funding sources.

9. REFERENCES

[1] AD Safe. `http://www.adsafe.org/`.

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

[3] Doxygen. `http://www.stack.nl/~dimitri/doxygen/`.

[4] FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`.

[5] Javascript game site. `http://javascript.internet.com/games/`.

[6] JavaScript reference. `https://developer.mozilla.org/en/Core_Javascript_1.5_Reference`.

[7] Javascript test cases. `http://mxr.mozilla.org/mozilla/source/js/tests/ecma/`.

[8] Mozilla rejects native code approach of chrome's nacl. `http://css.dzone.com/articles/mozilla-rejects-native-code`.

[9] Parsing time complexity. `http://en.wikipedia.org/wiki/Top-down_parsing`.

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

[11] XSS Cheat Sheet. `http://ha.ckers.org/xss.html`.

[12] Acker, S. V., Ryck, P. D., Desmet, L., Piessens, F., and Joosen, W. Webjail: Least-privilege integration of third-party components in web mashups. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[13] Barth, A., Jackson, C., and Li, W. Attacks on javascript mashup communication. In *W2SP: Web 2.0 Security and Privacy* (2009).

[14] Barth, A., Jackson, C., and Mitchell, J. C. Securing frame communication in browsers. In *17th USENIX Security Symposium* (2008).

[15] Barth, A., Weinberger, J., and Song, D. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *18th USENIX Security Symposium* (2009).

[16] Crites, S., Hsu, F., and Chen, H. Omash: enabling secure web mashups via object abstractions. In *CCS: Conference on Computer and Communication Security* (2008).

[17] De Keukelaere, F., Bhola, S., Steiner, M., Chari, S., and Yoshihama, S. Smash: secure component model for cross-domain mashups on unmodified browsers. In *WWW: Conference on World Wide Web* (2008).

[18] Dong, X., Tran, M., Liang, Z., and Jiang, X. Adsentry: Comprehensive and flexible confinement of javascript-based advertisements. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[19] Finifter, M., Weinberger, J., and Barth, A. Preventing capability leaks in secure javascript subsets. In *NDSS: Network and Distributed System Security Symposium* (2010).

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

[21] Guarnieri, S., and Livshits, B. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *18th USENIX Security Symposium* (2009).

[22] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Securing web application code by static analysis and runtime protection. In *WWW: Conference on World Wide Web* (2004).

[23] Kikuchi, H., Yu, D., Chander, A., Inamura, H., and Serikov, I. Javascript instrumentation in practice. In *APLAS: Asian Symposium on Programming Languages and Systems* (2008).

[24] Maffeis, S., Mitchell, J., and Taly, A. Run-time enforcement of secure javascript subsets. In *W2SP: Web 2.0 Security and Privacy* (2009).

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

[26] Meyerovich, L., and Livshits, B. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy* (2010).

[27] Microsoft Live Labs. WebSandbox. `http://websandbox.livelabs.com/`.

[28] Mozilla. Narcissus javascript engine. http://mxr.mozilla.org/mozilla/source/js/narcissus/.

[29] Nadji, Y., Saxena, P., and Song, D. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS: Network and Distributed System Security Symposium* (2009).

[30] Politz, J. G., Eliopoulos, S. A., Guha, A., and Krishnamurthi, S. ADsafety: type-based verification of JavaScript sandboxing. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 12–12.

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

[32] Resig, J. HTML Parser written in JavaScript. `http://ejohn.org/blog/pure-javascript-html-parser/`.

[33] Senocular. CSSParser. `http://www.senocular.com/index.php?id=1.289`.

[34] Sotirov, A. Blackbox reversing of XSS filters. *RECON* (2008).

[35] Ter Louw, M., Ganesh, K. T., and Venkatakrishnan, V. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium* (2010).

[36] Ter Louw, M., and Venkatakrishnan, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *SP: 30th IEEE Symposium on Security and Privacy* (2009).

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

[38] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *SP: IEEE Symposium on Security and Privacy* (2009).

[39] Yu, D., Chander, A., Islam, N., and Serikov, I. Javascript instrumentation for browser security. In *POPL* (2007).

[40] Yue, C., and Wang, H. Characterizing insecure javascript practices on the web. In *WWW: Conference on the World wide web* (2009).